UNIVERSITÉ DU QUÉBEC À MONTRÉAL

AMÉLIORER L'EFFICACITÉ DE L'ALGORITHME CDCL : DÉCOMPOSITIONS ARBORESCENTES DE GRANDES INSTANCES, CDCL SANS SAUT ARRIÈRE ET CDCL À ORDRE PARTIEL

THÈSE

PRÉSENTÉE

COMME EXIGENCE PARTIELLE

DU DOCTORAT EN INFORMATIQUE

PAR

ANTHONY JEAN-LUC MONNET

AOÛT 2013

REMERCIEMENTS

Je voudrais tout d'abord remercier les enseignants qui au long de ma scolarité m'ont transmis leur passion pour leurs domaines d'expertise. Je pense particulièrement, mais pas exclusivement, à Daniel Singer, Dieter Kratsch et Loïc Colson.

Je remercie également la Fondation de l'UQAM et le département d'informatique de l'UQAM pour les différentes bourses qu'ils m'ont octroyées au cours de mon doctorat. Au-delà du support financier qui a contribué à me permettre de poursuivre mon programme à temps plein, je suis reconnaissant de la confiance en mon projet dont ces bourses témoignent.

Je remercie ma famille et mes amis, particulièrement Julie, pour leurs encouragements et leur support au long des mes études.

Enfin et surtout, je tiens à remercier mon directeur de thèse Roger Villemaire. Vous êtes un directeur exceptionnel, très présent et à l'écoute, et vous m'avez permis d'effectuer ce doctorat dans les meilleures conditions possibles. Si je suis parvenu au bout de ce processus, c'est en grande partie à vous que je le dois.

TABLE DES MATIÈRES

TABLE DES FIGURES

LISTE DES TABLEAUX

LISTE DES ALGORITHMES

LISTES DES ABRÉVIATIONS, SIGLES ET ACRONYMES

#CSP	Problème de comptage associé au problème CSP
#SAT	Problème de comptage associé au problème SAT
All-CSP	Problème d'énumération associé au problème CSP
All-SAT	Problème d'énumération associé au problème SAT
BJ	*Backjumping*, algorithme de résolution du problème CSP
BT	*Backtracking*, algorithme de résolution du problème CSP
BT	*Backtracking with Decompositions*, algorithme de résolution du problème CSP
CBJ	*Conflict-Directed Backjumping*, algorithmme de résolution du problème CSP
CDCL	*Conflict-Driven Clause Learning*, algorithme de résolution du problème SAT
CDCL-OP	CDCL à ordre partiel
CDCL-OP-Dépendances	CDCL-OP avec dépendances supplémentaires
CDCL-OP-Surveillance	CDCL-OP avec surveillance accrue
CDCL-SSA	CDCL sans saut arrière
CDCL-SSA-PropAltAnt	CDCL-SSA avec gestion anticipée des propagations alternatives
CDCL-SSA-PropAltTard	CDCL-SSA avec gestion tardive des propagations alternatives
CDCL-SSA-Conservation	CDCL-SSA-PropAltAnt avec conservation additionnelle d'instanciations
CDCL-SSP	CDCL sans sauvegarde de phase
Chron	Heuristique chronologique pour le choix du niveau d'assertion
CSP	*Constraint Satisfaction Problem*
DB	*Dynamic Backtracking*, généralisation de DBT
DBT	*Dynamic Backtracking*, algorithme de résolution du problème CSP

DP	Davis-Putnam, algorithme de résolution du problème SAT
DPLL	Davis-Putnam-Logemann-Loveland, algorithme de résolution du problème SAT
FC	*Forward Checking*, mécanisme d'inférence pour le problème CSP
GBJ	*Graph-based backjumping*, algorithme de résolution du problème CSP
GDBT	*Generalized Dynamic Backtracking*, algorithme de résolution du problème CSP
GPB	*Generalized Partial Order Backtracking*, algorithme de résolution du problème CSP
LB	Littéraux bloqués
MAC	*Maintaining Arc-Consistency*, mécanisme d'inférence pour le problème CSP
Max-CSP	Problème d'optimisation associé au problème CSP
Max-SAT	Problème d'optimisation associé au problème SAT
MaxDép	Heuristique de maximisation des dépendances supplémentaires pour le choix du niveau d'assertion
MaxDésinst	Heuristique de maximisation des désinstanciations pour le choix du niveau d'assertion
MinDép	Heuristique de minimisation des dépendances supplémentaires pour le choix du niveau d'assertion
MinDésinst	Heuristique de minimisation des désinstanciations pour le choix du niveau d'assertion
PIU	Point d'implication unique
PDB	*Partial Order Dynamic Backtracking*, algorithme de résolution du problème CSP
POB	*Partial Order Backtracking*, algorithme de résolution du problème CSP
QBF	*Quantified Boolean Formula*, problème de satisfaisabilité des formules booléennes quantifiées
SAT	Problème de satisfaisabilité d'une formule propositionnelle

RÉSUMÉ

Cette thèse s'intéresse à l'amélioration des performances pratiques de l'algorithme CDCL (*Conflict-Driven Clause Learning*) pour la résolution du problème de satisfaisabilité des formules propositionnelles, ou problème SAT. Plus particulièrement, nous cherchons à diminuer la destruction de l'instanciation courante lors des étapes de saut arrière, qui peuvent occasionner la désinstanciation de nombreuses variables n'ayant aucun rapport direct avec le conflit à résoudre.

Dans ce but, nous proposons trois approches différentes. La première est une amélioration de l'utilisabilité de la méthode déjà existante de décomposition implicite d'une instance SAT. Notre but principal est de permettre son application à des instances de plus grande taille possible, après avoir montré les limitations des implémentations existantes.

Nous développons également deux variations de l'algorithme CDCL, le CDCL sans saut arrière et le CDCL à ordre partiel. Si le premier supprime totalement la notion de saut arrière en permettant la propagation des clauses unitaires à des niveaux de décision quelconques, le second rend le saut arrière plus sélectif, en désinstanciant uniquement les niveaux de décision qui dépendent du niveau de retour du saut arrière.

Notre analyse est à la fois théorique, notamment par une analyse détaillée des propriétés de différentes variations des CDCL sans saut arrière et à ordre partiel, et pratique, puisque l'efficacité de nos contributions est évaluée en les implémentant comme modifications de solveurs SAT de l'état de l'art et en se servant de ces implémentations sur des instances SAT difficiles utilisées lors de compétitions internationales de solveurs.

Mots-clés : problème SAT, satisfaisabilité, formules propositionnelles, CDCL, décomposition arborescente, retour arrière, ordre partiel.

INTRODUCTION

Le problème de satisfaisabilité des formules propositionnelles, ou problème SAT, a une grande importance à la fois théorique et pratique en informatique. Son importance théorique tient au fait qu'il est le premier problème de décision à avoir été démontré NP-complet (Cook, 1971) ; SAT est donc un exemple emblématique de la classe des problèmes NP-complets, dont une réponse possible peut être vérifiée en temps polynomial mais pour lesquels il n'existe à ce jour aucun algorithme polynomial, sans que l'on sache si un tel algorithme polynomial peut exister ou non.

Le problème SAT est également très important en pratique, car il permet de modéliser intuitivement un grand nombre d'autres problèmes typiques de nombreux domaines de l'informatique, parmi lesquels la conception et la vérification de circuits électroniques (Goldberg, Prasad et Brayton, 2001; Nam, Sakallah et Rutenbar, 1999; Stephan, Brayton et Sangiovanni-Vincentelli, 1996), la vérification formelle (Biere et al., 2003; Velev et Bryant, 2003), l'intelligence artificielle (Kautz et Selman, 1992; Memik et Fallah, 2002; Grastien et al., 2007), la bioinformatique (Lynce et Marques-Silva, 2008) et la cryptanalyse (Mironov et Zhang, 2006; Eibach, Pilz et Völkel, 2008).

La conception de solveurs SAT efficaces a donc été grandement motivée par ces deux aspects complémentaires ; d'une part, ils fournissent un exemple intéressant de différentes techniques pour solutionner efficacement en pratique un problème NP-complet ; d'autre part, tout progrès dans la résolution du problème SAT se répercute également sur l'efficacité de résolution des différents problèmes modélisables par la satisfaisabilité d'une formule propositionnelle. De fait, les progrès des solveurs SAT et la diversité de leurs applications ont un effet d'entraînement réciproque, car des solveurs plus efficaces incitent plus de chercheurs de domaines variés à expérimenter la résolution pratique de divers problèmes par un encodage sous forme de problème SAT, et l'accroissement

des applications de SAT augmente à son tour l'intérêt d'améliorer davantage les performances des solveurs.

Depuis environ 15 ans, les solveurs SAT complets (qui sont capables de prouver aussi bien la satisfaisabilité que l'insatisfaisabilité d'une formule) les plus performants sont basés sur un algorithme de recherche en profondeur à saut arrière, nommé CDCL (*Conflict-Driven Clause Learning*, Marques-Silva et Sakallah, 1999). Cet algorithme a permis un gain d'efficacité important par rapport aux algorithmes de recherche en profondeur conventionnels, car son système de saut arrière combiné à un apprentissage de nouvelles clauses permet d'optimiser l'élagage de l'espace de recherche à chaque conflit rencontré. Le CDCL s'assure en fait qu'une nouvelle occurrence de ce conflit sera rendue impossible dans la plus grande partie possible de l'arbre de recherche ; pour cela, son saut arrière peut défaire plusieurs niveaux de décision à la fois (un niveau de décision étant constitué d'une décision et de ses conséquences), jusqu'à un certain niveau nommé le niveau d'assertion, alors que la recherche en profondeur simple, en comparaison, se contente d'effectuer un retour arrière, c'est-à-dire d'inverser la décision la plus récente possible, qu'elle ait ou non un rapport avec le conflit. Le CDCL améliore donc fortement les performances de la recherche en profondeur en évitant la découverte répétée de conflits identiques.

Le saut arrière du CDCL a cependant en contrepartie un inconvénient : en retournant plus loin en arrière pour propager la clause apprise le plus tôt possible, il peut également défaire un bien plus grand nombre de niveaux de décision. Or, par définition, aucun de ces niveaux défaits n'a de rapport direct avec le conflit qui a déclenché le saut arrière, hormis celui où le conflit a été détecté. Or, le CDCL progresse précisément en construisant petit à petit une instanciation partielle jusqu'à obtenir si possible un modèle pour la formule considérée. Comme le saut arrière défait potentiellement une plus grande partie de l'instanciation courante que le retour arrière, il invalide donc une plus grande partie de cette progression de l'algorithme, sans que ces désinstanciations supplémentaires soient nécessaires à la résolution du conflit. Nous parlerons pour décrire ce phénomène de destructivité des sauts arrière.

Différentes stratégies ont été proposées pour chercher à réduire cette destructivité dans le CDCL, ainsi que dans l'algorithme CBJ (*conflict-directed backjumping*, Prosser, 1993), un algorithme similaire pour la résolution du très proche problème de satisfaction de contraintes (CSP, ou *constraint satisfaction problem*). Parmi celles-ci, on peut citer la détection des composantes connexes (Biere et Sinz, 2006), qui empêche un retour arrière à l'intérieur d'une composante connexe de défaire des instanciations dans une autre composante, la décomposition implicite (Durairaj et Kalla, 2004; Huang et Darwiche, 2003; Bjesse et al., 2004; Li et van Beek, 2004; Jégou et Terrioux, 2003; Habbas, Amroun et Singer, 2011), qui provoque la séparation des instances en composantes connexes puis leur résolution indépendante en contraignant les heuristiques de choix de variables, l'heuristique de sauvegarde de phase (Pipatsrisawat et Darwiche, 2007), spécifique au problème SAT, ou encore les algorithmes de retour arrière dynamique ou à ordre partiel et leurs généralisations (Ginsberg, 1993; McAllester, 1993; Ginsberg et McAllester, 1994; Bliek, 1998), définis dans le cadre du problème CSP.

La présente thèse poursuit cet objectif d'améliorer l'efficacité en pratique de l'algorithme CDCL pour le problème SAT en réduisant la destructivité des sauts arrière. Pour cela, nous proposons d'améliorer l'utilisabilité d'une méthode existante, la décomposition implicite, et concevons deux nouvelles variantes de l'algorithme CDCL, respectivement le CDCL sans saut arrière et le CDCL à ordre partiel, pour lesquelles nous définissons et analysons diverses variations.

Notre contribution concernant l'application des méthodes de décomposition implicite au problème SAT (Monnet et Villemaire, 2010) est motivée par le constat que les implémentations disponibles ne sont pas adaptées à la taille généralement importante des instances de problèmes SAT difficiles. En effet, les méthodes de construction de décompositions utilisées, bien qu'heuristiques, sont trop lourdes pour traiter le graphe caractéristique de telles instances : elles nécessitent une quantité d'espace mémoire prohibitive ou un temps de calcul trop long pour permettre de globalement améliorer le temps de résolution de l'instance. Pour résoudre ce problème, nous proposons une méthode de construction de décomposition volontairement très simple et montrons expérimentale-

ment qu'elle permet de traiter des instances d'une taille conséquente.

Le CDCL sans saut arrière et le CDCL à ordre partiel constituent la contribution principale de cette thèse. Il s'agit de deux variantes du CDCL classique dont le but est de réduire la destructivité des sauts arrière conventionnels. Le CDCL sans saut arrière, comme son nom l'indique, propose d'abolir totalement le saut arrière : aucun niveau de décision n'est alors défait, à l'exception du niveau de conflit et de ses possibles conséquences dans d'autres niveaux. La clause de conflit peut alors être propagée au niveau d'assertion, comme dans un CDCL classique, mais sans avoir défait tous les niveaux de décision ultérieurs. Cette méthode permet bien de réduire la destructivité de la gestion des conflits, mais elle a pour effet de complexifier la gestion de l'algorithme : en particulier, les propagations unitaires et les conflits peuvent avoir lieu à un niveau de décision quelconque, alors que dans un CDCL classique ils ne peuvent survenir qu'au dernier niveau de décision créé.

Le CDCL à ordre partiel (Monnet et Villemaire, 2012a,b), quant à lui, conserve la notion de saut arrière, mais rend la destruction des niveaux de décision plus sélective : seuls sont détruits les niveaux qui dépendent, directement ou indirectement, du niveau d'assertion. Ces dépendances entre niveaux de décision définissent donc un ordre partiel entre eux. Le CDCL classique, en comparaison, détruit indistinctement tous les niveaux ultérieurs au niveau d'assertion, en considérant l'ordre total chronologique de leur création. La destructivité du CDCL à ordre partiel est donc plus sélective par rapport à celle du CDCL classique, sans être plus réduite dans tous les cas de figure, car un saut arrière partiel peut parfois défaire un niveau de décision chronologiquement antérieur au niveau d'assertion. L'ordre partiel sur les niveaux de décision a notamment la particularité d'ajouter une nouvelle dimension de liberté à l'algorithme : le niveau d'assertion peut souvent être choisi parmi plusieurs niveaux candidats, alors qu'il est défini de façon unique dans un CDCL classique. Cette caractéristique permet l'utilisation d'heuristiques pour le choix de ce paramètre additionnel.

Nous nous intéressons à la fois aux aspects théoriques et pratiques de ces contributions. D'un point de vue théorique, nous montrons par exemple que notre proposition de décomposition efficace de grandes instances SAT se formalise dans le cadre d'un concept qui généralise à la fois les définitions de décomposition arborescente et de *dtree*. En ce qui concerne nos variantes de l'algorithme CDCL, nous proposons un ensemble de nouvelles définitions de propriétés des algorithmes basés sur le CDCL, portant sur des aspects de l'algorithme comme la surveillance des clauses, l'exhaustivité de la détection des conflits et des clauses unitaires, ou encore sur la capacité des conflits à générer de nouvelles clauses inédites et donc à contribuer à l'élagation de l'espace de recherche. Nous montrons diverses relations entre ces propriétés et nous nous en servons pour comparer le CDCL classique, une de ses variantes, le CDCL de type GRASP, et nos propres contributions. Nous observons en particulier que le CDCL sans saut arrière et le CDCL à ordre partiel ne vérifient plus certaines des propriétés du CDCL classique et proposons diverses modifications pour les rétablir entièrement ou en partie. Nous remarquons au passage que le CDCL de type GRASP lui aussi ne vérifie pas autant de propriétés que le CDCL classique, ce qui contribue à la compréhension des différences d'efficacité observées entre les deux variantes.

Sur le plan pratique, toutes nos contributions sont évaluées expérimentalement en les implémentant par une modification d'un solveur SAT parmi les plus performants de l'état de l'art (MINISAT (Eén et Sörensson, 2004) dans le cadre des décompositions implicites, GLUCOSE (Audemard et Simon, 2009) dans le cas des CDCL sans saut arrière et à ordre partiel), puis les performances de ces modifications sont comparées avec celles des implémentations originales sur des instances difficiles de problèmes utilisées lors des compétitions internationales annuelles de solveurs SAT (Le Berre et al., 2012). Ces évaluations indiquent que les implémentations originales des solveurs sont si optimisées qu'il est difficile d'améliorer leur performance : d'une part, les modifications que nous apportons aux gestions des sauts arrière impliquent des tâches supplémentaires qui ralentissent l'exécution de parties fréquemment utilisées du codes telles que la vérification des clauses surveillées, alors que les solveurs originaux doivent en partie leur efficacité

à leur minimisation du temps passé dans ces parties fréquemment exécutées. D'autre part, même en faisant abstraction de la vitesse d'exécution en elle-même, il est difficile de réduire la longueur de la résolution en nombre d'étapes élémentaires de l'algorithme, car nos modifications provoquent des interactions complexes avec divers aspects de l'algorithme, qui sont optimisés pour le fonctionnement du saut arrière conventionnel. Nous verrons toutefois, notamment dans le cas du CDCL à ordre partiel, que sur certains types particuliers d'instances les avantages de nos méthodes surpassent leurs inconvénients, ce qui permet d'améliorer substantiellement les performances de résolution par rapport à l'implémentation originale du solveur.

Le contenu de la thèse est organisé de la façon suivante : le chapitre 1 résume les notions préliminaires nécessaires à sa lecture et fixe les conventions de notations que nous utiliserons, notamment dans le domaine de la théorie des graphes. Le chapitre 2 est une introduction globale à notre domaine d'études, c'est-à-dire à l'algorithme CDCL et plus généralement au problème SAT, et à sa résolution à l'aide d'algorithmes complets. Ce chapitre contient également une courte présentation du problème CSP et des algorithmes de résolution équivalents, notamment CBJ. Le chapitre 3 aborde notre problématique de la destructivité des sauts arrière dans les algorithmes CDCL et CBJ, et effectue une revue bibliographique des principales méthodes existantes qui contribuent à sa réduction. Le chapitre 4 motive et présente notre méthode de décomposition implicite de grandes instances SAT, et évalue son applicabilité ainsi que son influence sur les performances de résolution. Le chapitre 5 est un court chapitre consacré à la définition de diverses propriétés des algorithmes de type CDCL qui nous permettront de comparer théoriquement le CDCL classique aux CDCL sans saut arrière et à ordre partiel, ainsi qu'à la preuve de certaines relations entre ces propriétés. Nous y établissons également les propriétés du CDCL classique et du CDCL de type GRASP. Enfin, les chapitres 6 et 7 introduisent respectivement le CDCL sans saut arrière et le CDCL à ordre partiel. Pour chacun de ces algorithmes, nous motivons sa conception et expliquons son intuition, décrivons son fonctionnement en détail, évaluons ses propriétés et proposons diverses variantes implémentatives aux propriétés plus ou moins complètes. Nous verrons

notamment que le CDCL sans saut arrière et le CDCL à ordre partiel de base permettent la détection exhaustive des conflits à condition de désactiver les littéraux bloqués, mais pas des clauses unitaires, que des modifications des algorithmes permettent de rétablir la détection exhaustive des clauses unitaires, mais que seul le CDCL à ordre partiel permet de garantir que tout conflit produise une nouvelle clause précédemment inconnue. Toutes les variantes des CDCL sans saut arrière et à ordre partiel présentées sont évaluées expérimentalement en comparant leurs performances sur des instances difficiles à celles de l'implémentation originale du solveur modifié.

CHAPITRE I

NOTIONS PRÉLIMINAIRES

Ce chapitre présente diverses notions utilisées dans de la thèse sans être directement liées à son thème principal. Il permet notamment de fixer les conventions de notations ou de significations que nous emploierons par la suite. La section 1.1 traite les notions de base relatives à la théorie des ensembles. La section 1.2 s'intéresse aux définitions des relations, fonctions et ordres. Les sections 1.3 et 1.4 couvrent respectivement les domaines de la théorie des graphes et de la théorie de la complexité.

1.1 Théorie des ensembles

Un **ensemble** fini E est une collection non-ordonnée d'éléments distincts. $|E| \in \mathbb{N}$ est la **cardinalité** de cet ensemble, c'est-à-dire le nombre d'éléments qu'il contient. Si $|E| = 0$, alors E est l'**ensemble vide**, noté $\emptyset$. Si $|E| = 1$ ou $|E| = 2$, E est appelé respectivement un **singleton** ou une **paire**. L'ensemble formé par $n \in \mathbb{N}$ éléments distincts $e_1, e_2, \ldots, e_n$ est noté $\{e_1, e_2, \ldots, e_n\}$. Si l'élément e appartient à l'ensemble E, alors on note $e \in E$; dans le cas contraire, on note $e \notin E$. La notation $e_1, e_2 \in E$ signifie que e_1 et e_2 sont des éléments (possiblement identiques) de E. Si nous voulons préciser que e_1 et e_2 ne peuvent être identiques, nous utiliserons la notation $\{e_1, e_2\} \subseteq E$ (voir la notion de sous-ensemble ci-dessous).

Une collection ordonnée de n éléments (pas forcément distincts) est appelée un **n-uplet**. Un n-uplet formé des n éléments $e_1, e_2, \ldots, e_n$ dans cet ordre est noté

$(e_1, e_2, \ldots, e_n)$. Les n-uplets de 1, 2 ou 3 éléments sont appelés respectivement **singletons**, **couples** et **triplets**.

Pour deux ensembles E et F, $E \cup F = \{x \mid x \in E \text{ ou } x \in F\}$ est l'**union** de E et F, $E \cap F = \{x \mid x \in E \text{ et } x \in F\}$ est l'**intersection** de E et F et $E \setminus F = \{e \in E \mid e \notin F\}$ est la **différence** de E et F, soit l'ensemble des éléments de E qui n'appartiennent pas à F. $E \times F = \{(e, f) \mid e \in E, f \in F\}$ est le **produit cartésien** de E et F, c'est-à-dire l'ensemble des couples formés d'un élément de E et d'un élément de F. $\forall n \in \mathbb{N}^*$, nous utiliserons la notation E^n pour désigner le produit cartésien de n occurrences d'un ensemble E (un élément de E^n est donc un n-uplet d'éléments de E).

Nous noterons indifféremment $\subset$ ou $\subseteq$ une **inclusion large**, c'est-à-dire une inclusion pouvant aussi être une égalité : $E \subset F \Leftrightarrow E \subseteq F \Leftrightarrow \forall e \in E, e \in F$. L'**inclusion stricte** sera explicitement notée$\subsetneq$: $E \subsetneq F \Leftrightarrow \forall e \in E, e \in F$ et $\exists f \in F \mid f \notin E$. E est un **sous-ensemble** de F si et seulement si $E \subseteq F$.

Pour un ensemble E, nous noterons $\mathcal{P}(E) = \{E' \mid E' \subseteq E\}$ l'**ensemble des parties** de E, c'est-à-dire l'ensemble des sous-ensembles de E. Pour tout ensemble E, $\emptyset \in \mathcal{P}(E)$, $E \in \mathcal{P}(E)$ et la cardinalité de $\mathcal{P}(E)$ est $2^{|E|}$. Soit $P_E \subseteq \mathcal{P}(E)$ un sous-ensemble des parties de E. P_E est une **partition** de E s'il ne contient pas l'ensemble vide et si chaque élément de E est contenu dans un et un seul élément de P_E ; plus formellement, $\emptyset \notin P_E$, $\bigcup_{p \in P_E} p = E$ et $\forall \{p_1, p_2\} \in P_E$, $p_1 \cap p_2 = \emptyset$. $\{E\}$ est une partition triviale de E. Pour tout $i \in \{1, \ldots, |E|\}$, $\mathcal{P}_i(E) = \{p \in \mathcal{P}(E) \mid |p| = i\}$ est l'ensemble des parties de E de cardinalité i.

1.2 Relations, fonctions et ordres

Soient E et F deux ensembles. Une **relation** (binaire) $\mathcal{R}$ entre E et F est un sous-ensemble de $E \times F$. $\forall e \in E$, $f \in F$, si $(e, f) \in \mathcal{R}$, on dit que e est en relation avec f, que l'on note $e\mathcal{R}f$. Si e n'est pas en relation avec f, on note $\neg(e\mathcal{R}f)$. $\mathcal{R}$ est une **fonction (partielle)** de E dans F, notée $\mathcal{R} : E \nrightarrow F$, si chaque élément de E est en relation avec au plus un élément de F : $\forall e \in E$, $|\{f \in F \mid e\mathcal{R}f\}| \leq 1$.

Soit ϕ une fonction partielle de E dans F. Pour tout $e \in E$ et $f \in F$, la notation $e \overset{\phi}{\mapsto} f$ (ou $e \mapsto f$ en l'absence d'ambiguïté) est équivalente à $e\phi f$. Le **domaine de définition** de ϕ, noté $\mathcal{D}(\phi)$, est l'ensemble des éléments de E auxquels ϕ associe un élément de F. Pour tout $e \in \mathcal{D}(\phi)$, $\phi(e) \in F$ est l'élément de F associé à e par ϕ. Nous utiliserons la notation $\phi(e) =$ indéfini comme équivalente à $e \notin \mathcal{D}(\phi)$.

Par définition de ϕ, on a $\mathcal{D}(\phi) \subseteq E$. Si $\mathcal{D}(\phi) = E$, ce qui signifie que ϕ associe à tout élément de E un élément de F, alors ϕ est une **fonction totale** de E dans F, notée $\phi : E \to F$. Pour alléger le texte, dans la suite de cette thèse, nous considérerons toute fonction comme totale, sauf indication contraire.

Pour un ensemble E, $\mathrm{id}_E : E \to E$ est la **fonction identité** sur E, qui associe chaque élément de E à lui-même : $\forall e \in E$, $\mathrm{id}_E(e) = e$. Soient $\phi : E \to F$ et $E' \subseteq E$. La **restriction** de ϕ sur E', notée $\phi|_{E'} : E' \to F$, est la fonction qui à tout élément $e' \in E'$ associe l'élément $\phi(e') \in F$.

Soit E un ensemble, $E_1 \subseteq E$ un sous-ensemble de E et p une propriété pouvant être vraie ou fausse sur tout sous-ensemble de E; plus formellement, $p : \mathcal{P}(E) \to \mathcal{B}$ où $\mathcal{B} = \{\text{vrai}, \text{faux}\}$ est l'ensemble des valeurs booléennes. E_1 est dit **maximal** pour la propriété p si $p(E_1) =$ vrai et $\forall E_2 \subseteq E$, $E_1 \subsetneq E_2 \Rightarrow p(E_2) =$ faux. Inversement, E_1 est dit **minimal** pour la propriété p si $p(E_1) =$ vrai et $\forall E_2 \subseteq E$, $E_2 \subsetneq E_1 \Rightarrow p(E_2) =$ faux.

Dans nos descriptions d'algorithmes, nous manipulerons souvent les fonctions en tant que cas particuliers de relations binaires, c'est-à-dire de sous-ensembles du produit cartésien de leurs ensembles de départ et d'arrivée. Par exemple, pour une fonction $\phi : E \to F$ et des éléments $e \in E$ et $f \in F$, la notation $\phi(e) \leftarrow f$ signifie que l'on modifie ϕ en lui faisant associer f à e. Si ϕ associait initialement un autre élément de F à E, cette association est remplacée par la nouvelle. Cette notation est donc équivalente à $\phi \leftarrow (\phi \setminus \{(e, f') \mid f' \in F\}) \cup \{(e, f)\}$. De même, la notation $\phi(e) \leftarrow$ indéfini signifie que l'on retire l'association à l'élément e si elle existe. Elle est donc équivalente à $\phi \leftarrow \phi \setminus \{(e, f') \mid f' \in F\}$ ou encore à $\mathcal{D}(\phi) \leftarrow \mathcal{D}(\phi) \setminus \{e\}$.

Soit $\mathcal{R}$ une relation binaire sur un ensemble E, c'est-à-dire entre E et lui-même ($\mathcal{R} \subseteq E^2$). $\mathcal{R}$ est **réflexive** si $\forall e \in E, e\mathcal{R}e$ ou au contraire **irréflexive** si $\forall e \in E, \neg(e\mathcal{R}e)$. $\mathcal{R}$ est **antisymétrique** si $\forall e, e' \in E$, $e\mathcal{R}e'$ et $e'\mathcal{R}e \Rightarrow e = e'$; $\mathcal{R}$ est **asymétrique** si $\forall e, e' \in E$, $e\mathcal{R}e' \Rightarrow \neg(e'\mathcal{R}e)$. $\mathcal{R}$ est **transitive** si $\forall e_1, e_2, e_3 \in E$, $e_1\mathcal{R}e_2$ et $e_2\mathcal{R}e_3 \Rightarrow e_1\mathcal{R}e_3$. $\forall e, e' \in E$, e et e' sont dits **comparables** par $\mathcal{R}$ si $e\mathcal{R}e'$ ou $e'\mathcal{R}e$. La **fermeture réflexive** d'une relation $\mathcal{R}$ sur E, notée $\mathcal{R}^{\mathrm{id}}$, est la plus petite relation réflexive contenant $\mathcal{R}$, c'est-à-dire $\mathcal{R}^{\mathrm{id}} = \mathcal{R} \cup \mathrm{id}_E$. De même, la **fermeture transitive** de la relation $\mathcal{R}$ sur E, notée $\mathcal{R}^+$, est la plus petite relation transitive contenant $\mathcal{R}$. On peut définir $\mathcal{R}^+$ récursivement par $\mathcal{R}^+ = \mathcal{R} \cup \{(x, z) \in E^2 \mid (x, y) \in \mathcal{R}, (y, z) \in \mathcal{R}^+\}$.

Une relation sur E est un **ordre partiel strict**, noté $\prec$, si elle est à la fois irréflexive, asymétrique et transitive. $e_1 \succ e_2$ est une notation équivalente à $e_2 \prec e_1$ et $e_1 \not\prec e_2$ est équivalente à $\neg(e_1 \prec e_2)$. Une relation sur E est un **ordre partiel faible**, noté $\preceq$, si elle est à la fois réflexive, antisymétrique et transitive. L'ordre partiel faible associé à un ordre partiel strict $\prec$ est $\prec^{\mathrm{id}}$, la fermeture réflexive de $\prec$; inversement, l'ordre partiel strict associé à un ordre partiel faible $\preceq$ est la relation $\preceq \setminus \mathrm{id}_E$.

Un ordre sur E est dit **total** si toute paire d'éléments distincts est comparable ; on le note alors $<$ s'il est strict ou $\leq$ s'il est faible. L'ordre strict associé à un ordre faible total (respectivement partiel) est lui-même total (respectivement partiel) et vice versa. Pour alléger le texte, nous considérerons que tout ordre est total et strict, sauf indication contraire.

Soit E un ensemble et $\prec$ un ordre partiel (strict ou non) sur E. $\forall e \in E$, e est un **élément maximal** de E (selon $\prec$) si $\forall e' \in E \setminus \{e\}$, $e \not\prec e'$; inversement, e est un **élément minimal** de E si $\forall e' \in E \setminus \{e\}$, $e' \not\prec e$. Les éléments maximaux et minimaux selon un ordre partiel strict et son ordre partiel faible associé sont identiques. Pour tout ensemble E et tout ordre partiel $\prec$, il existe au moins un élément maximal et un élément minimal. Si $<$ est un ordre total sur E, alors E a exactement un élément maximal et un élément minimal selon $<$.

1.3 Théorie des graphes

Cette section présente différentes notions relatives aux graphes. Nous nous intéresserons successivement aux graphes non-orientés et orientés (sous-sections 1.3.1 et 1.3.2 respectivement), aux arbres (sous-section 1.3.3) et aux hypergraphes (sous-section 1.3.4).

1.3.1 Graphes non-orientés

Un **graphe** (non-orienté) $G(\mathcal{S}, \mathcal{A})$ est défini par un ensemble de **sommets** $\mathcal{S}$ et un ensemble d'**arêtes** $\mathcal{A} \subseteq \mathcal{P}_2(\mathcal{S})$ reliant des paires de sommets : $\forall a \in \mathcal{A}$, $a = \{s, s'\} \subseteq \mathcal{S}$. Deux sommets de G reliés par une arête sont dits **voisins**. Le nombre de voisins d'un sommet est appelé son **degré**.

Un **chemin** (fini) dans un graphe non-orienté $G(\mathcal{S}, \mathcal{A})$ est une suite de sommets $s_0, s_1, \ldots, s_k \in \mathcal{S}$, $k \in \mathbb{N}^*$ telle que $\forall i \in \{0, \ldots, k-1\}$, $\{s_i, s_{i+1}\} \in \mathcal{A}$: tout couple de sommets consécutifs dans le chemin est relié par une arête du graphe. k est appelé la **longueur** du chemin. Un chemin de longueur k est un **cycle** si $s_0 = s_k$. Un graphe est dit **acyclique** s'il ne contient aucun cycle. Un chemin est **élémentaire** s'il ne comporte pas deux fois le même sommet, ou s'il est un cycle de longueur k et que s_0 et s_k sont ses seuls sommets identiques. Sauf indication contraire, nous ne parlerons par la suite que de chemins élémentaires.

Soient $G(\mathcal{S}, \mathcal{A})$ et $G'(\mathcal{S}', \mathcal{A}')$ deux graphes. $G(\mathcal{S}', \mathcal{A}')$ est un **sous-graphe** de $G(\mathcal{S}, \mathcal{A})$ si et seulement si $\mathcal{S}' \subseteq \mathcal{S}$ et $\mathcal{A}' \subseteq \mathcal{A}$. Soit $S \subseteq \mathcal{S}$ un sous-ensemble des sommets du graphe $G(\mathcal{S}, \mathcal{A})$. Le sous-graphe de G **induit** par S, noté $G|_S(S, \mathcal{A}|_S)$, est le sous-graphe de G qui contient uniquement les sommets de S ainsi que les arêtes reliant ces sommets : $\mathcal{A}|_S = \{a \in \mathcal{A} \,|\, a \subseteq S\}$. Au contraire, nous noterons $G_{\setminus S} = G|_{\mathcal{S} \setminus S}$ le sous-graphe de G induit par $\mathcal{S} \setminus S$, c'est-à-dire obtenu en éliminant les sommets de S et les arêtes dont l'une des extrémités appartient à S. Un sous-graphe $G'(\mathcal{S}', \mathcal{A}')$ de $G(\mathcal{S}, \mathcal{A})$ est dit **couvrant** s'il contient tous les sommets de G (donc si $\mathcal{S}' = \mathcal{S}$). De façon analogue aux sous-graphes induits, nous noterons $G_{|A}(\mathcal{S}, \mathcal{A})$ le sous-graphe couvrant de $G(\mathcal{S}, \mathcal{A})$

obtenu en conservant uniquement un sous-ensemble des arêtes $A \subseteq \mathcal{A}$ et $G_{\setminus A}(\mathcal{S}, \mathcal{A} \setminus A)$ le sous-graphe couvrant de $G(\mathcal{S}, \mathcal{A})$ obtenu en retirant le sous-ensemble d'arêtes $A \subseteq \mathcal{A}$.

Un graphe $G(\mathcal{S}, \mathcal{A})$ est **connexe** si, pour toute paire de sommets $\{s_1, s_2\} \subseteq \mathcal{S}$, il existe un chemin entre s_1 et s_2 ; dans le cas contraire, G est **non-connexe**. Une **composante connexe** d'un graphe $G(\mathcal{S}, \mathcal{A})$ est un ensemble de sommets $C \subseteq \mathcal{S}$ tel que $G|_C$ est connexe et C est maximal pour cette propriété : pour tout $C' \subseteq \mathcal{S}$, si $C \subsetneq C'$ alors $G|_{C'}$ est non-connexe. L'ensemble des composantes connexes de G, noté $\mathrm{Conn}(G)$, forme une partition de $\mathcal{S}$. G est connexe si et seulement s'il contient une seule composante connexe. Nous appellerons **composante déconnectée** de G (sous-entendu « déconnectée du reste de G ») toute composante connexe ou tout union de composantes connexes de G.

Un sous-ensemble de sommets $S \subseteq \mathcal{S}$ est un **séparateur** du graphe $G(\mathcal{S}, \mathcal{A})$ si $G_{\setminus S}$ est non-connexe, c'est-à-dire si l'on peut rendre G non-connexe en retirant les sommets de S. De même, un sous-ensemble d'arêtes $A \subseteq \mathcal{A}$ est un **ensemble coupant** du graphe $G(\mathcal{S}, \mathcal{A})$ si $G_{\setminus A}$ est non-connexe, c'est-à-dire si l'on peut rendre G non-connexe en retirant les arêtes de A.

Un graphe $G(\mathcal{S}, \mathcal{A})$ est dit **complet** si toute paire de sommets est reliée par une arête, c'est-à-dire si $\mathcal{A} = \mathcal{P}_2(\mathcal{S})$. Un ensemble de sommets $S \subseteq \mathcal{S}$ est une **clique** s'ils sont tous reliés entre eux dans G, donc si $G|_S$ est complet.

1.3.2 Graphes orientés

Un **graphe orienté** $G(\mathcal{S}, \mathcal{A})$ est défini par un ensemble de sommets $\mathcal{S}$ et un ensemble d'**arcs** $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ reliant un sommet de départ s_d à un sommet d'arrivée s_a : $\forall a \in \mathcal{A}, a = (s_\mathrm{d}, s_\mathrm{a}) \in \mathcal{S} \times \mathcal{S}$. Un arc $(s_\mathrm{d}, s_\mathrm{a}) \in \mathcal{A}$ est une **boucle** si $s_\mathrm{d} = s_\mathrm{a}$.

Le graphe non-orienté **sous-jacent** au graphe orienté $G(\mathcal{S}, \mathcal{A})$ est le graphe $G'(\mathcal{S}, \mathcal{A}')$ où $\mathcal{A}' = \{\{s_1, s_2\} \subseteq \mathcal{S} \mid (s_1, s_2) \in \mathcal{A} \text{ ou } (s_2, s_1) \in \mathcal{A}\}$. Le graphe sous-jacent G' relie donc par une arête toute paire de sommets reliés dans G par un arc, quelque soit

son orientation.

Les définitions de chemin, de cycle et d'élémentarité dans un graphe orienté sont similaires aux définitions correspondantes dans un graphe orienté, à la différence près du respect de l'orientation des arcs : un chemin dans le graphe orienté $G(\mathcal{S}, \mathcal{A})$ est une suite de sommets $s_0, s_1, \ldots, s_k \in \mathcal{S}$, $k \in \mathbb{N}^*$ telle que $\forall i \in \{0, k-1\}$, $(s_i, s_{i+1}) \in \mathcal{A}$. Les notions de sous-graphe, de sous-graphe induit et de sous-graphe couvrant restent identiques dans le cas d'un graphe orienté.

Un graphe orienté $G(\mathcal{S}, \mathcal{A})$ est **fortement connexe** si pour toute paire de sommets $\{s_1, s_2\} \subseteq \mathcal{A}$ il existe un chemin de s_1 à s_2 et un chemin de s_2 à s_1 ; il est **faiblement connexe** si son graphe non-orienté sous-jacent est connexe. Un graphe orienté fortement connexe est également faiblement connexe. À partir de ces deux définitions de connexité, on peut dériver les notions de **composantes fortement connexes** et de **composantes faiblement connexes** d'un graphe orienté. Une composante faiblement connexe d'un graphe orienté est l'union d'une ou plusieurs composantes fortement connexes.

1.3.3 Arbres

Un **arbre** $T(\mathcal{N}, \mathcal{A})$ est un graphe non-orienté, acyclique et connexe. Il est défini par un ensemble de **nœuds** $\mathcal{N}$ et un ensemble d'arêtes $\mathcal{A}$ reliant des paires de nœuds : $\forall a \in \mathcal{A}$, $a = \{n, n'\} \subseteq \mathcal{N}$. Pour toute paire de nœuds $n, n' \in \mathcal{N}$, il existe dans T un unique chemin (élémentaire) entre n et n'.

Un **arbre enraciné** est un couple $\mathcal{T}(T(\mathcal{N}, \mathcal{A}), r)$ où $T(\mathcal{N}, \mathcal{A})$ est un arbre et $r \in \mathcal{N}$ est un nœud de T appelé la **racine** de T. L'enracinement d'un arbre définit implicitement une orientation de l'arbre : pour toute arête $a = \{n, n'\} \in \mathcal{A}$, n est le nœud de départ si et seulement si n' est sur le chemin unique de n à r et réciproquement. [1] Pour toute arête $a = \{n_\mathrm{d}, n_\mathrm{a}\}$ où n_d est le nœud de départ et n_a le nœud d'arrivée, n_d est le **fils** de n_a et n_a est le **père** de n_d. Un nœud peut avoir plusieurs fils mais il ne peut avoir

1. Nous supposons donc que les arêtes sont orientées implicitement vers la racine. Cette convention est arbitraire et nous aurions pu utiliser une orientation opposée.

qu'un seul père. Dans un arbre enraciné, la fonction $p : \mathcal{N} \setminus \{r\} \to \mathcal{N}$ associe à tout nœud distinct de la racine son père, et $f : \mathcal{N} \to \mathcal{P}(\mathcal{N})$ associe à tout nœud l'ensemble (possiblement vide) de ses fils. On peut également définir les fonctions $a : \mathcal{N} \to \mathcal{P}(\mathcal{N})$, qui associe à tout nœud $n \in \mathcal{N}$ l'ensemble de ses **ancêtres** (l'ensemble des nœuds sur le chemin unique de n à la racine, n exclu mais racine incluse si elle est différente de n) et $d : \mathcal{N} \to \mathcal{P}(\mathcal{N})$, qui associe à tout nœud $n \in \mathcal{N}$ l'ensemble de ses **descendants** (l'ensemble des nœuds dont le chemin unique vers la racine passe par n, n exclu).

La racine est le seul nœud sans père. Les nœuds sans fils sont appelés les **feuilles** de l'arbre. Les nœuds de l'arbre qui ne sont pas des feuilles sont appelés **nœuds internes**. Un arbre enraciné est dit **binaire** (ou plus généralement ***n*-aire**) si tout nœud interne a au plus 2 fils (resp. n fils). Il est dit **strictement *n*-aire** si tout nœud interne a exactement n fils.

Pour un arbre enraciné $\mathcal{T}(T(\mathcal{N}, \mathcal{A}), r)$ et un nœud $n \in \mathcal{N}$, le **sous-arbre** de $\mathcal{T}$ enraciné en n, noté $\mathrm{s}(\mathcal{T}, n)$, est un arbre enraciné $\mathcal{T}'(T'(\mathcal{N}', \mathcal{A}'), n)$ ou $\mathcal{N}'$ contient n et l'ensemble de ses descendants dans $\mathcal{T}$ et $T'(\mathcal{N}', \mathcal{A}')$ est le sous-graphe de T induit par $\mathcal{N}'$.

Soit $\mathcal{T}(T(\mathcal{N}, \mathcal{A}), r)$ un arbre enraciné. Un **parcours préfixe** de $\mathcal{T}$ est une liste ordonnée de tous les sommets de $\mathcal{N}$ telle que le premier nœud de la liste est r, suivi de la concaténation de parcours préfixes des sous-arbres enracinés en chaque fils de r. L'ordre entre ces différents sous-arbres est arbitraire. Par conséquent, le parcours préfixe énumère entièrement les nœuds d'un sous-arbre donné avant de passer à un autre sous-arbre, et tout nœud est parcouru après tous ses ancêtres.

La **profondeur** d'un nœud $n \in \mathcal{N}$ dans un arbre enraciné $\mathcal{T}(T(\mathcal{N}, \mathcal{A}), r)$ est la longueur du chemin unique entre n et r. La profondeur d'un arbre enraciné $\mathcal{T}$ est la profondeur maximale des nœuds de $\mathcal{T}$.

Un graphe non-orienté, acyclique mais non-connexe est appelé **forêt**, puisqu'il constitue une collection d'arbres.

1.3.4 Hypergraphes

Un **hypergraphe** $H(\mathcal{S}, \mathcal{H})$ est constitué d'un ensemble de sommets $\mathcal{S}$ et d'un ensemble d'**hyperarêtes**, c'est-à-dire de sous-ensembles non-vides de $\mathcal{S}$: $\mathcal{H} \subseteq \mathcal{P}(\mathcal{S}) \backslash \{\emptyset\}$. Un graphe est donc un cas particulier d'hypergraphe dont toutes les hyperarêtes sont de cardinalité 2.

Un chemin dans un hypergraphe se définit de façon similaire à un chemin dans un graphe : une suite de sommets $s_0, s_1, \ldots, s_k \in \mathcal{S}$, $k \in \mathbb{N}^*$ est un chemin si toute paire de sommets consécutifs appartient à une hyperarête commune : $\forall i \in \{0, \ldots, k-1\}, \exists h \in \mathcal{H} \,|\, \{s_i, s_{i+1}\} \subseteq h$. La connexité d'un hypergraphe, ses composantes connexes, séparateurs et ensembles coupants se définissent également de la même manière.

L'**hypergraphe dual** de $H(\mathcal{S}, \mathcal{H})$, noté $H_H^{\mathrm{D}}(\mathcal{S}_H^{\mathrm{D}}, \mathcal{H}_H^{\mathrm{D}})$, est obtenu en inversant les rôles des sommets et des hyperarêtes. Chaque sommet de H_H^{D} correspond à une hyperarête de H, donc $\mathcal{S}_H^{\mathrm{D}} = \mathcal{H}$, et chaque hyperarête de H_H^{D} regroupe les hyperarêtes de H qui ont un certain sommet de H en commun : $\mathcal{H}_H^{\mathrm{D}} = \{\{h \in \mathcal{H} \,|\, s \in h\} \,|\, s \in \mathcal{S}\}$. Tout séparateur dans H correspond à un ensemble coupant dans H_H^{D} et vice versa.

Il existe différentes façons de représenter un hypergraphe sous la forme d'un graphe. Par exemple, le **graphe primal** d'un hypergraphe $H(\mathcal{S}, \mathcal{H})$, noté $G_H^{\mathrm{P}}(\mathcal{S}_H^P, \mathcal{A}_H^P)$, a le même ensemble de sommets que H et deux sommets sont reliés par une arête dans G_H^{P} si et seulement s'ils appartiennent à une hyperarête commune dans H : $\mathcal{S}_H^{\mathrm{P}} = \mathcal{S}$ et $\mathcal{A}_H^{\mathrm{P}} = \{\{s_1, s_2\} \subseteq \mathcal{S} \,|\, \exists h \in \mathcal{H}, \{s_1, s_2\} \subseteq h\}$. Par conséquent, pour toute hyperarête $h \in \mathcal{H}$, h est une clique dans G_H^{P}. Inversement, dans le **graphe dual** [2] de $H(\mathcal{S}, \mathcal{H})$, noté $G_H^{\mathrm{D}}(\mathcal{S}_H^{\mathrm{D}}, \mathcal{A}_H^{\mathrm{D}})$, chaque sommet correspond à une hyperarête de H, et deux sommets de G_H^{D} sont reliés par une arête si et seulement si les hyperarêtes de H correspondantes ont au moins un sommet de H en commun : $\mathcal{S}_H^{\mathrm{D}} = \mathcal{H}$ et $\mathcal{A}_H^{\mathrm{D}} = \{\{h_1, h_2\} \subseteq \mathcal{H} \,|\, h_1 \cap h_2 \neq \emptyset\}$.

---

2. Malgré l'utilisation du même nom, cette définition du graphe dual d'un hypergraphe est sans rapport avec celle de graphe dual d'un graphe planaire, que nous n'utiliserons pas dans cette thèse.

Pour tout sommet $s \in \mathcal{S}$ de H, l'ensemble des sommets de G_H^{D} qui correspondent à une hyperarête de H contenant s forme une clique de G_H^{D}.

Notons que l'hypergraphe dual d'un graphe est une représentation sans perte de l'hypergraphe d'origine, c'est-à-dire qu'elle permet de le retrouver ; en effet, en construisant l'hypergraphe dual de l'hypergraphe dual, on retrouve l'hypergraphe d'origine. Ce n'est cependant pas le cas des graphes primal et dual d'un hypergraphe.

1.4 Théorie de la complexité

La théorie de la complexité algorithmique permet, entre autres, de décrire l'accroissement du temps et de l'espace nécessaire à l'exécution d'un algorithme en fonction de la taille des instances considérées. La taille d'une instance est littéralement la quantité d'espace mémoire nécessaire à l'encoder. En pratique, on utilise souvent, afin de simplifier les analyses de complexité, des paramètres de taille propres au problème considéré qui permettent d'approximer ou de borner la taille globale de l'instance. Par exemple, dans le cas de SAT, le paramètre de taille principal est le nombre de variables de l'instance, noté n. Dans le cas de formules en forme normale conjonctive (voir sous-section 2.2.4), nous considérerons également comme paramètres le nombre de clauses m et la longueur maximale des clauses k. La taille d'une instance SAT en formule normale conjonctive est alors au plus proportionnelle à $n + m \times k$.

De même, toujours pour simplifier les analyses de complexité, on suppose généralement que les opérations élémentaires, telles que la lecture et lecture d'un espace mémoire unitaire ou une opération logique sur les bits, nécessitent dans tous les cas un temps identique. Le temps est alors estimé en nombre de ces étapes élémentaires, alors que par exemple, en pratique, la lecture de deux éléments peut nécessiter des temps d'exécution très différents en raison des mécanismes de mémoire cache.

Enfin, une dernière simplification courante des analyses de complexité algorithmique consiste à décrire uniquement le comportement **asymptotique** de cette complexité, c'est-à-dire la vitesse de son accroissement lorsque la taille des instances tend

vers l'infini. Les analyses de complexité asymptotique ne permettent pas de déterminer parmi deux algorithmes lequel sera le plus efficace en temps ou en espace sur une taille d'instance particulière. Cependant, si un algorithme A a une complexité temporelle (resp. spatiale) supérieure à celle d'un algorithme B, alors il existe une taille d'instance limite telle que l'algorithme B est plus efficace en temps (resp. en espace) que l'algorithme A sur les instances de taille supérieure à cette taille limite.

Pour un algorithme donné, notons respectivement $t(n)$ et $e(n)$ le temps et l'espace mémoire maximums nécessaires à l'exécution de l'algorithme sur toute instance de taille n. Soit $f(n)$ une fonction réelle. On dit que la complexité temporelle (resp. spatiale) d'un algorithme est de $O(f(n))$ s'il existe une constante strictement positive $c \in \mathbb{R}_+^*$ et un seuil $n_s \in \mathbb{N}$ tels que $\forall n > n_s$, $t(n) \leq c \times f(n)$ (resp. $e(n) \leq c \times f(n)$). Au contraire, la complexité temporelle (resp. spatiale) d'un algorithme est de $\Omega(f(n))$ s'il existe une constante strictement positive $c \in \mathbb{R}_+^*$ et un seuil $n_s \in \mathbb{N}$ tels que $\forall n > n_s$, $t(n) \geq c \times f(n)$ (resp. $e(n) \geq c \times f(n)$). Enfin, la complexité temporelle (resp. spatiale) d'un algorithme est de $\Theta(f(n))$, si elle est à la fois de $O(f(n))$ et de $\Omega(f(n))$, c'est-à-dire s'il existe deux constantes strictement positives $c_1, c_2 \in \mathbb{R}_+^*$, $c_1 < c_2$, et un seuil $n_s \in \mathbb{N}$ tels que $\forall n > n_s$, $c_1 \times f(n) \leq t(n) \leq c_2 \times f(n)$ (resp. $c_1 \times f(n) \leq e(n) \leq c_2 \times f(n)$). La complexité en espace d'un algorithme est toujours inférieure ou égale à sa complexité en temps, puisque l'utilisation d'un certaine quantité d'espace implique au minimum d'y accéder une fois au cours de l'algorithme.

Un algorithme est dit **constant** en temps (resp. en espace) si sa complexité temporelle (resp. spatiale) est de $O(1)$, ou **linéaire** en temps (resp. en espace) si sa complexité temporelle (resp. spatiale) est de $O(n)$. De même, on peut définir un algorithme **polynomial** dont la complexité est de $O(n^d)$ pour un degré $d \in \mathbb{N}^*$ donné, ainsi qu'un algorithme **exponentiel** dont la complexité est de $O(c^{d \times n})$, où $c, d \in \mathbb{R}_+^*$ (de façon équivalente, on peut noter $2^{O(n)}$ la complexité d'un algorithme exponentiel).

Si un algorithme a une borne de complexité de $O(f(n))$, cette borne est dite **serrée** s'il existe une suite d'instances de diverses tailles qui atteignent cette borne, c'est-à-dire

si l'algorithme restreint à cette suite a une complexité de $\Theta(f(n))$. On dit alors que l'algorithme a une complexité de $\Theta(f(n))$ **au pire des cas**. Ainsi, un algorithme est linéaire, polynomial ou exponentiel au pire des cas si sa complexité est respectivement de $\Theta(n)$, $\Theta(n^d)$ pour un degré $d \in \mathbb{N}^*$ donné ou $2^{\Theta(n)}$ au pire des cas.

CHAPITRE II

LES ALGORITHMES DE RECHERCHE À SAUT ARRIÈRE POUR SAT ET CSP

Ce chapitre rappelle les notions importantes autour des problèmes SAT et CSP et de leurs principaux algorithmes complets. Nous mettrons particulièrement l'accent sur le problème SAT, ainsi que sur l'algorithme CDCL, un algorithme de recherche en profondeur avec saut arrière pour résoudre SAT. Tout d'abord, la section 2.1 présente quelques notions préliminaires de logique propositionnelle nécessaires à la définition du problème SAT. La section 2.2 expose cette définition et les principales caractéristiques du problème. Nous profitons de ces deux premières sections pour introduire la majorité des notations et conventions que nous adopterons le long de cette thèse. Les sections 2.3 à 2.5 introduisent trois algorithmes pour la résolution du problème SAT, respectivement DP, DPLL et CDCL. Enfin, la section 2.6 propose une présentation plus brève du problème CSP et de certains de ses algorithmes.

2.1 Logique propositionnelle

Cette section fournit un bref aperçu de la logique propositionnelle, c'est-à-dire de la logique sur les formules propositionnelles. Les différentes notions abordées nous permettront de définir le problème SAT. Les sous-sections 2.1.1 et 2.1.2 définissent respectivement les formules propositionnelles et les littéraux, puis la sous-section 2.1.3 couvre les instanciations de variables et les interprétations de formules.

2.1.1 Formules propositionnelles

Une **formule propositionnelle** est une formule logique constituée uniquement de **variables booléennes** et de **connecteurs logiques** de différentes arités. On peut définir récursivement une formule propositionnelle par $F = v \mid \sqcap (G_1, G_2, \ldots, G_n)$ où v est une variable booléenne, $\sqcap$ est un connecteur logique d'arité $n \in N^*$ et $G_1, G_2, \ldots, G_n$ sont des formules propositionnelles.

Les connecteurs les plus courants sont le connecteur unaire de la **négation** ($\neg$) ainsi que les connecteurs binaires de la **conjonction** ($\wedge$), de la **disjonction** ($\vee$), de l'**implication** ($\rightarrow$) et de l'**équivalence** ($\leftrightarrow$). Nous nous restreindrons à des formules propositionnelles contenant uniquement ces connecteurs ; les formules que nous considérerons sont donc toutes récursivement d'une des formes suivantes, où v est une variable booléenne et G et H sont des formules propositionnelles :

- v (une variable booléenne seule) ;
- $\neg G$ (la négation de la sous-formule G) ;
- $G \wedge F$ (la conjonction des sous-formules G et F) ;
- $G \vee F$ (la disjonction des sous-formules G et F) ;
- $G \rightarrow F$ (l'implication de la sous-formule F par la sous-formule G) ;
- $G \leftrightarrow F$ (l'équivalence des sous-formules G et F).

Exemple 2.1. $F = \neg(a \leftrightarrow \neg b) \wedge (\neg a \vee b)$ est une formule propositionnelle car :

- $F_1 = a$ et $F_2 = b$ sont des formules propositionnelles, puisqu'elles sont formées uniquement d'une variable booléenne chacune ;
- $F_3 = \neg a$ et $F_4 = \neg b$ sont des formules propositionnelles, car $F_3 = \neg F_1$ et $F_4 = \neg F_2$;
- $F_5 = a \leftrightarrow \neg b = F_1 \leftrightarrow F_4$ et $F_6 = \neg a \vee b = F_3 \vee F_2$ sont des formules propositionnelles ;
- $F_7 = \neg(a \leftrightarrow \neg b) = \neg F_5$ est une formule propositionnelle ;
- F est donc également une formule propositionnelle car $F = F_7 \wedge F_6$.

Soit $\mathcal{V}$ un ensemble (fini) de variables booléennes. Nous noterons $F(\mathcal{V})$ une formule

propositionnelle F formée sur les variables de $\mathcal{V}$, et $\mathcal{F}(\mathcal{V})$ l'ensemble des formules propositionnelles formées sur $\mathcal{V}$. Inversement, nous noterons $\mathcal{V}(F)$ l'ensemble des variables présentes dans une formule propositionnelle F. Pour tout ensemble $\mathcal{V}$ et toute formule $F(\mathcal{V}) \in \mathcal{F}(\mathcal{V})$, $\mathcal{V}(F(\mathcal{V})) \subseteq \mathcal{V}$.

2.1.2 Littéraux

Un **littéral** est une variable v ou sa négation $\neg v$. Les littéraux v et $\neg v$ sont dits **opposés**. Nous noterons $\mathcal{L} = \{v, \neg v \,|\, v \in \mathcal{V}\}$ l'ensemble des littéraux formés sur un ensemble de variables $\mathcal{V}$. Pour tout littéral $l \in \mathcal{L}$, nous utiliserons la notation $\neg l$ pour désigner son littéral opposé :

$$\neg l = \begin{cases} \neg v & \text{si } l = v, v \in \mathcal{V} \\ v & \text{si } l = \neg v, v \in \mathcal{V} \end{cases}.$$

$\rho : \mathcal{L} \to \mathcal{B}$ associe à tout littéral sa **polarité** ou son **signe** :

$$\rho(l) = \begin{cases} \text{vrai} & \text{si } l = v, v \in \mathcal{V} \\ \text{faux} & \text{si } l = \neg v, v \in \mathcal{V} \end{cases}.$$

$\nu : \mathcal{L} \to \mathcal{V}$ associe à tout littéral sa variable : $\nu(l) = v$ si $l = v$ ou $l = \neg v$. Par commodité, nous étendrons ν aux ensembles de littéraux : $\forall L \subseteq \mathcal{L}$, $\nu(L) = \{\nu(l) \,|\, l \in \mathcal{L}\}$.

2.1.3 Instanciations et interprétations

Soit $\mathcal{B} = \{\text{vrai}, \text{faux}\}$ l'ensemble des deux valeurs booléennes. Pour tout ensemble de variables booléennes $\mathcal{V}$, on peut définir une **instanciation** $\sigma : \mathcal{V} \nrightarrow \mathcal{B}$ comme une fonction partielle de $\mathcal{V}$ vers $\mathcal{B}$. Pour $v \in \mathcal{D}(\sigma)$, nous dirons que σ instancie v à la valeur $\sigma(v)$. Si σ est une fonction totale, nous dirons qu'elle est une **instanciation complète**. Pour deux instanciations σ et σ' sur $\mathcal{V}$, nous dirons que σ' **étend** σ si $\mathcal{D}(\sigma) \subseteq \mathcal{D}(\sigma')$ et $\forall v \in \mathcal{D}(\sigma)$, $\sigma'(v) = \sigma(v)$.

Toute instanciation σ définie sur un ensemble de variables $\mathcal{V}$ peut être étendue en une unique **interprétation** $\varsigma : \mathcal{F}(\mathcal{V}) \nrightarrow \mathcal{B}$ des formules propositionnelles sur $\mathcal{V}$. Cette interprétation est définie récursivement selon la construction des formules telle que définie dans la sous-section 2.1.1 et utilise la sémantique usuelle des connecteurs :

- $\varsigma(v) = \sigma(v)$
- $\varsigma(\neg G) = \begin{cases} \text{vrai} & \text{si } \varsigma(G) = \text{faux} \\ \text{faux} & \text{si } \varsigma(G) = \text{vrai} \\ \text{indéfini} & \text{si } \varsigma(G) = \text{indéfini} \end{cases}$
- $\varsigma(G \wedge H) = \begin{cases} \text{vrai} & \text{si } \varsigma(G) = \text{vrai et } \varsigma(H) = \text{vrai} \\ \text{faux} & \text{si } \varsigma(G) = \text{faux ou } \varsigma(H) = \text{faux} \\ \text{indéfini} & \text{sinon} \end{cases}$
- $\varsigma(G \vee H) = \begin{cases} \text{vrai} & \text{si } \varsigma(G) = \text{vrai ou } \varsigma(H) = \text{vrai} \\ \text{faux} & \text{si } \varsigma(G) = \text{faux et } \varsigma(H) = \text{faux} \\ \text{indéfini} & \text{sinon} \end{cases}$
- $\varsigma(G \rightarrow H) = \begin{cases} \text{vrai} & \text{si } \varsigma(G) = \text{faux ou } \varsigma(H) = \text{vrai} \\ \text{faux} & \text{si } \varsigma(G) = \text{vrai et } \varsigma(H) = \text{faux} \\ \text{indéfini} & \text{sinon} \end{cases}$
- $\varsigma(G \leftrightarrow H) = \begin{cases} \text{vrai} & \text{si } \varsigma(G) \neq \text{indéfini et } \varsigma(G) = \varsigma(H) \\ \text{faux} & \text{si } \varsigma(G) \neq \text{indéfini et } \varsigma(H) \neq \text{indéfini et } \varsigma(G) \neq \varsigma(H) \\ \text{indéfini} & \text{si } \varsigma(G) = \text{indéfini ou } \varsigma(H) = \text{indéfini} \end{cases}$

Exemple 2.2. Reprenons la formule propositionnelle $F = \neg(a \leftrightarrow \neg b) \wedge (\neg a \vee b)$ et ses sous-formules F_1 à F_7 définies dans l'exemple 2.1 et considérons l'instanciation $\sigma = \{a \rightarrow \text{faux}, b \rightarrow \text{faux}\}$. On a alors :

- $\varsigma(F_1) = \sigma(a) = \text{faux}$;
- $\varsigma(F_2) = \sigma(b) = \text{faux}$;
- $\varsigma(F_3) = \varsigma(\neg F_1) = \text{vrai car } \varsigma(F_1) = \text{faux}$;

- $\varsigma(F_4) = \varsigma(\neg F_2) =$ vrai car $\varsigma(F_2) =$ faux ;
- $\varsigma(F_5) = \varsigma(F_1 \leftrightarrow F_4) =$ faux car $\varsigma(F_1) \neq \varsigma(F_4)$;
- $\varsigma(F_6) = \varsigma(F_2 \vee F_3) =$ vrai car $\varsigma(F_3) =$ vrai ;
- $\varsigma(F_7) = \varsigma(\neg F_5) =$ vrai car $\varsigma(F_5) =$ faux ;
- au final, $\varsigma(F) = \varsigma(F_7 \vee F_6) =$ vrai car $\varsigma(F_7) =$ vrai et $\varsigma(F_6) =$ vrai.

Par la suite, afin d'alléger le texte, nous utiliserons la même notation (généralement σ) pour désigner indifféremment l'instanciation des variables et l'interprétation des formules associée.

Notons que pour tout ensemble $\mathcal{V}$ de n variables, il existe 2^n instanciations complètes. Par conséquent, il existe également 2^n interprétations complètes distinctes sur $\mathcal{F}(\mathcal{V})$.

Nous utiliserons parfois un point de vue ensembliste en considérant une instanciation σ comme un ensemble de littéraux. Pour un littéral $l \in \mathcal{L}$ et une instanciation $\sigma : \mathcal{V} \nrightarrow \mathcal{B}$, $l \in \sigma \Leftrightarrow \sigma(l) =$ vrai. Nous dirons alors que σ instancie le littéral l si $l \in \sigma$. Il est évident qu'une instanciation ne peut contenir deux littéraux opposés : $\forall l \in \mathcal{L}, l \in \sigma \Rightarrow \neg l \notin \sigma$. Enfin, σ est une instanciation complète si et seulement si $\forall v \in \mathcal{V}, v \in \nu(\sigma)$.

Exemple 2.3. L'instanciation $\sigma = \{a \rightarrow$ faux, $b \rightarrow$ faux$\}$ peut aussi s'écrire $\sigma = \{\neg a, \neg b\}$.

Une instanciation complète σ sur $\mathcal{V}$ est un **modèle** d'une formule $F(\mathcal{V})$ si $\sigma(F(\mathcal{V})) =$ vrai. Une formule $F(\mathcal{V})$ est dite **satisfaisable** si elle possède au moins un modèle. Elle est dite **insatisfaisable** dans le cas contraire.

Exemple 2.4. L'instanciation $\sigma = \{\neg a, \neg b\}$ est un modèle de la formule $F = \neg(a \leftrightarrow \neg b) \wedge (\neg a \vee b)$ car σ est complète sur $\mathcal{V}(F) = \{a, b\}$ et $\sigma(F) =$ vrai. F est donc satisfaisable.

2.2 Le problème SAT

À l'aide des notions préliminaires abordées précédemment, cette section définit le problème SAT et détaille certaines de ses caractéristiques les plus importantes.

La sous-section 2.2.1 définit la notion de satisfaisabilité et le problème SAT. La sous-section 2.2.2 distingue les différentes catégories d'algorithmes pour la résolution du problème SAT et, plus généralement, pour les problèmes de recherche. La sous-section 2.2.3 introduit quelques notions supplémentaires de complexité algorithmique afin d'expliquer la signification et les conséquences de la NP-complétude du problème SAT. La sous-section 2.2.4 définit la notion de forme normale conjonctive et explique son importance en pratique dans les solveurs SAT. La sous-section 2.2.5 présente quelques-unes des variantes du problème SAT. Enfin, la sous-section 2.2.6 fournit une liste non-exhaustive des principales applications qui sont en pratique résolues par encodage en problème SAT.

2.2.1 Définition

Le **problème SAT**, ou problème de satisfaisabilité d'une formule propositionnelle, consiste à déterminer si une formule donnée $F(\mathcal{V})$ est satisfaisable(Garey et Johnson, 1979, section 2.6). Une formule propositionnelle dont on cherche à déterminer la satisfaisabilité est une **instance** du problème SAT.

Le problème SAT prend en entrée une formule propositionnelle et produit en sortie une réponse positive ou négative. Il s'agit donc d'un **problème de décision**, c'est-à-dire dont la réponse est booléenne. En général, le problème SAT est plus précisément formulé en tant que **problème de recherche**, c'est-à-dire d'un problème qui demande de vérifier l'existence ou non de solutions dans un espace de recherche donné, et de fournir une solution s'il en existe. Dans le cas de SAT, l'espace de recherche est l'ensemble des instanciations complètes de $\mathcal{V}$ et les solutions sont les éventuels modèles de $F(\mathcal{V})$. Nous parlerons de **solveurs SAT** pour désigner les logiciels implémentant des algorithmes de

résolution du problème SAT.

Le problème SAT est fortement lié à la notion de **tautologie**. Une formule $F(\mathcal{V})$ est une tautologie si, pour toute instanciation complète σ sur $\mathcal{V}$, $\sigma(F(\mathcal{V})) =$ vrai. Or, toute formule propositionnelle $F(\mathcal{V})$ est une tautologie si et seulement si $\neg F(\mathcal{V})$ est insatisfaisable.

2.2.2 Catégories d'algorithmes pour le problème SAT

Les algorithmes de résolution pour un problème de recherche, donc en particulier pour le problème SAT, peuvent être classifiés en diverses catégories selon les propriétés de leurs réponses. Les trois propriétés les plus essentielles sont la correction, la complétude et la terminaison des algorithmes [1].

Définition 2.1. *Un algorithme pour un problème de recherche est dit* ***correct*** *si, lorsqu'il retourne une réponse positive, la solution retournée est valide. Si un algorithme peut retourner une réponse positive alors que le problème n'admet aucune solution, il est dit* ***incorrect****.*

Définition 2.2. *Un algorithme pour un problème de recherche est dit* ***complet*** *si, lorsqu'il retourne une réponse négative, il n'existe réellement aucune solution au problème, et s'il finit toujours par retourner une réponse négative lorsque le problème n'admet aucune solution. Si un algorithme peut retourner une réponse négative alors que le problème admet une ou plusieurs solutions, ou s'il peut ne pas retourner de réponse négative lorsque le problème n'admet aucune solution, il est dit* ***incomplet****.*

Définition 2.3. *Un algorithme pour un problème de recherche* ***termine*** *si toute exécution de cet algorithme est finie et retourne une réponse, sous hypothèse de ressources de temps et d'espace mémoire suffisants.*

Une autre propriété importante en pratique est la constructivité éventuelle de

1. Les définitions que nous présentons sont couramment utilisées dans divers domaines, par exemple en programmation logique (Drabent et Miłkowska, 2005).

l'algorithme :

Définition 2.4. *Un algorithme pour un problème de recherche est dit* ***constructif*** *si, lorsqu'il retourne une réponse positive, il fournit également une solution du problème.*

La plupart des algorithmes pour SAT, comme DPLL (section 2.4) et CDCL (section 2.5), sont constructifs : ils prouvent qu'une formule est satisfaisable en exhibant un exemple de modèle. Cependant, quelques algorithmes ne sont pas constructifs. C'est le cas de DP (section 2.3) : celui-ci prouve la satisfaisabilité en montrant qu'aucune contradiction n'est inférable à partir de la formule considérée.

La construction d'un algorithme incorrect pour SAT est triviale : il suffit de répondre positivement en retournant une instanciation complète quelconque. Cependant, les algorithmes incorrects ne sont pas couramment utilisés pour résoudre SAT à notre connaissance. Cela s'explique d'une part par la constructivité de la majorité des algorithmes, et d'autre part par la facilité de vérifier une solution potentielle, qui permet d'éviter facilement de retourner un modèle erroné.

En revanche, il existe différents types d'algorithmes incomplets pour SAT. Par exemple, certains algorithmes sont de type Monte Carlo (Drias, 1998) : ils explorent uniquement une partie limitée de l'espace de recherche. S'ils rencontrent une solution au cours de cette recherche partielle, ils peuvent avec certitude déclarer la formule satisfaisable. Dans le cas contraire, l'algorithme déclare la formule insatisfaisable avec une probabilité donnée que cette réponse soit correcte. Il est donc possible qu'une formule soit déclarée insatisfaisable alors qu'elle possède un modèle.

Les algorithmes de recherche locale (Selman, Kautz et Cohen, 1996) sont un autre exemple d'algorithmes incomplets. Ces algorithmes recherchent heuristiquement des modèles de la formule considérée sans garantir un parcours exhaustif de l'espace de recherche. Ils sont donc incapables de déterminer l'insatisfaisabilité d'une formule : le cas échéant, ils ne terminent pas.

DPLL et CDCL sont eux des algorithmes de recherche systématique, qui s'assurent

d'explorer exhaustivement toute partie de l'espace de recherche pouvant contenir un modèle. Par conséquent, si ces algorithmes terminent sans avoir trouvé de modèle, la formule considérée est forcément insatisfaisable ; ils sont donc complets.

Les algorithmes de recherche locale, comme nous l'avons évoqué précédemment, ne terminent jamais dans le cas de formules insatisfaisables, et peuvent également ne pas terminer sur des formules satisfaisables. Les algorithmes de Monte Carlo, bien qu'incomplets, terminent toujours, car leur principe est d'estimer la satisfaisabilité d'une instance en un nombre fini d'échantillonage, au prix d'une marge d'erreur. Enfin, nous montrerons ultérieurement la terminaison de DP, DPLL et CDCL.

Dans la suite de cette thèse, nous nous restreindrons aux algorithmes corrects, complets et qui terminent. De tels algorithmes sont dits **totalement corrects**.

2.2.3 NP-complétude du problème SAT

Les définitions de bornes de complexité énumérées dans la section 1.4 supposent que le modèle de machine utilisé est **déterministe**, c'est-à-dire qu'à partir d'un état donné de l'exécution de l'algorithme, il existe un unique état suivant possible. Au contraire, si l'on considère que chaque état peut admettre plusieurs états suivants et que le résultat d'un algorithme consiste en l'ensemble de toutes les exécutions possibles, le modèle est dit **non-déterministe**. Si chaque exécution d'un algorithme déterministe peut être représentée par une chaîne d'états visités successivement, une exécution d'un algorithme non-déterministe produit un arbre où chaque branche représente un chemin d'exécution possible.

Un algorithme non-déterministe pour un problème de recherche est correct si toutes les branches de son exécution sont correctes, complet si au moins une de ses branches déclare l'existence d'une solution si le problème en admet, et il termine si toutes ses branches terminent.

À partir du modèle de machine non-déterministe, on peut alors définir la com-

plexité non-déterministe d'un algorithme, qui considère comme fonctions $t(n)$ et $e(n)$ le temps et l'espace maximaux nécessités par toute branche de l'exécution de l'algorithme sur toute instance de taille n. En particulier, un algorithme s'exécute **en temps polynomial non-déterministe** si sa complexité non-déterministe est $O(n^d)$ pour un degré $d \in \mathbb{N}^*$ donné. Dans la suite de cette thèse, sauf indication contraire, nous considérerons la complexité déterministe des algorithmes.

Nous pouvons maintenant introduire les classes de problèmes P et NP. Un problème appartient à la classe des problèmes polynomiaux, notée P, s'il existe un algorithme en temps polynomial déterministe et totalement correct qui le résout. De même, la classe NP des problèmes polynomiaux non-déterministes contient l'ensemble des problèmes pour lesquels il existe un algorithme en temps polynomial non-déterministe et totalement correct qui le résout.

Dans le cas d'algorithmes non-déterministes pour des problèmes de recherche, chaque branche d'exécution représente généralement la vérification d'une solution potentielle particulière de l'espace de recherche. Les problèmes de recherche appartiennent donc à NP si, étant donné un élément de l'espace de recherche, on peut vérifier en temps polynomial si cet élément est une solution au problème.

La classe des problèmes **NP-complets** est, intuitivement, l'ensemble des problèmes les plus complexes de la classe NP. Plus formellement, si p et p' sont deux problèmes, une **réduction polynomiale** de p vers p' est un algorithme qui résout une instance du problème p en utilisant un nombre polynomial d'opérations élémentaires et un nombre polynomial d'appels à un algorithme pour résoudre le problème p'. On dit que p se réduit polynomialement à p' s'il existe une telle réduction polynomiale de p vers p'. Un problème p est dit **NP-difficile** si tout problème $p' \in$ NP se réduit polynomialement à p. Un problème est NP-complet s'il est NP-difficile et qu'il appartient lui-même à NP. Le problème SAT est NP-complet (Cook, 1971) [2], ainsi que plusieurs centaines de

2. La preuve originale de Cook montre plus précisément la NP-complétude du problème consistant à déterminer si une formule propositionnelle est une tautologie, ce qui, nous l'avons vu dans la sous-

problèmes courants en informatique, tels que les problèmes de coloration de graphe, de cycle hamiltonien, de voyageur de commerce ou de clique maximale pour n'en citer que quelques uns (Garey et Johnson, 1979).

D'après les définitions précédentes, il est évident qu'un algorithme en temps polynomial est un cas particulier d'algorithme non-déterministe en temps polynomial ; par conséquent, P $\subseteq$ NP. Cependant, on ignore toujours si la réciproque de cette inclusion est vraie, et donc si P = NP. Les problèmes NP-complets pourraient être une des clés de la réponse : il suffirait de trouver un algorithme polynomial pour résoudre un des problèmes NP-complets pour montrer que P = NP. En effet, par définition, tout problème $p \in$ NP peut se réduire polynomialement à tout problème p' NP-complet. Si l'on effectue une réduction polynomiale de p vers p' que l'on sait résoudre en temps polynomial, on obtient un algorithme en temps polynomial pour p, donc P = NP. Malheureusement, aucun algorithme en temps polynomial n'a été trouvé pour un problème NP-complet, mais il n'a pas non plus été prouvé qu'un tel algorithme ne peut exister. En particulier, tous les algorithmes totalement corrects connus pour résoudre SAT sont en temps exponentiel au pire des cas. Cette exponentialité s'explique par la taille de l'espace de recherche, qui contient, pour une formule à n variables, les 2^n instanciations complètes de ces variables. Un algorithme trivial consiste à énumérer toutes ces instanciations complètes, ce qui implique une complexité en temps de $2^{\Theta(n)}$. Nous verrons que les algorithmes DP, DPLL et CDCL conservent au pire des cas cette même complexité temporelle.

2.2.4 Formules en forme normale conjonctive

Si le problème SAT peut s'appliquer à une formule propositionnelle quelconque, la plupart des solveurs se restreignent en pratique à des formules particulières, dites en **forme normale conjonctive** (ou CNF, pour *conjunctive normal form*). Dans cette sous-section, nous donnerons tout d'abord la définition de cette forme normale conjonctive, puis expliquerons l'intérêt de restreindre les solveurs SAT au support des formules

section 2.2.1, revient à résoudre le problème SAT sur la négation de la formule.

propositionnelles en CNF uniquement. Enfin, nous verrons que cette contrainte ne provoque aucune perte de généralité des solveurs car toute formule propositionnelle peut être transformée en une formule CNF équivalente ou équisatisfaisable.

2.2.4.1 Définition

Une **clause** est une disjonction de littéraux, c'est-à-dire une formule propositionnelle de la forme $l_1 \vee l_2 \vee \ldots \vee l_n$ où $l_1, l_2, \ldots, l_n$ sont des littéraux. Une **formule en forme normale conjonctive** est une conjonction de clauses, c'est-à-dire une formule propositionnelle de la forme

$$(l_{1,1} \vee l_{1,2} \vee \ldots \vee l_{1,n_1}) \wedge (l_{2,1} \vee l_{2,2} \vee \ldots \vee l_{2,n_2}) \wedge \ldots \wedge (l_{m,1} \vee l_{m,2} \vee \ldots \vee l_{m,n_m})$$

où $l_{i,j}$ est un littéral $\forall i \in \{1, \ldots, m\}$, $j \in \{1, n_m\}$.

Exemple 2.5. $F = \neg(a \leftrightarrow \neg b) \wedge (\neg a \vee b)$ n'est pas en forme normale conjonctive, car elle contient un opérateur d'équivalence ($\leftrightarrow$), ainsi qu'un opération de négation qui est appliqué à la sous-formule $F_5 = a \leftrightarrow \neg b$; or dans une formule CNF, cet opérateur peut uniquement être appliqué à une variable.

Exemple 2.6. $G = (a \vee \neg b \vee c) \wedge (b \vee c \vee d) \wedge (\neg a \vee b \vee \neg c \vee \neg d)$ est une formule en forme normale conjonctive.

Notons que l'ordre des littéraux dans une clause ou des clauses dans une formule CNF n'a aucune incidence sur la satisfaisabilité ou les différents modèles de cette formule. Par conséquent, dans la suite de cette thèse, nous utiliserons souvent une notation ensembliste pour définir ou manipuler des formules CNF. Par exemple, la notation $F(\mathcal{V}, \mathcal{C})$ représente une formule CNF par un ensemble de clauses $\mathcal{C}$ sur les variables $\mathcal{V}$. Nous utiliserons généralement les notations $n = |\mathcal{V}|$ et $m = |\mathcal{C}|$ pour désigner respectivement le nombre de variables et de clauses d'une formule CNF. Chaque clause peut elle-même être considérée comme un ensemble de littéraux. Nous supposerons que toute clause ne peut pas contenir les deux littéraux opposés d'une même variable ; dans le cas contraire, la clause obtenue est une tautologie. Nous supposerons également, afin de simplifier les

définitions ultérieures, qu'une clause ne contient qu'une seule occurrence de chaque littéral. Une occurrence multiple d'un littéral n'a aucune incidence sur l'interprétation de la clause. Nous noterons $\top$ la formule vide et $\bot$ la clause vide. Pour un ensemble de variables $\mathcal{V}$, nous noterons $\mathcal{C}(\mathcal{V})$ l'ensemble des clauses formées à partir des variables dans $\mathcal{V}$ ($\bot$ incluse), et pour une formule F en CNF, nous noterons $\mathcal{C}(F)$ l'ensemble de ses clauses.

2.2.4.2 Utilité

L'intérêt de restreindre le problème SAT aux formules CNF est une grande simplification de la détection de la satisfaisabilité. En effet, comme une formule CNF est une conjonction de clauses, elle est vraie (sous une instanciation donnée) si et seulement si chacune de ses clauses est vraie. Une clause est une disjonction de littéraux, donc chaque clause est elle-même vraie si et seulement si au moins un de ses littéraux est vrai. Au final, une formule CNF est vraie si et seulement si toutes ses clauses contiennent au moins un littéral vrai. La plupart des algorithmes de résolution de SAT restreints aux formules CNF utilisent cette propriété pour déterminer si une instanciation est un modèle de la formule testée ; c'est le cas de DPLL (section 2.4) et de CDCL (section 2.5).

D'après cette définition, pour toute instanciation σ, on a $\sigma(\top) =$ vrai (puisqu'aucune clause ne peut être fausse) et $\sigma(\bot) =$ faux (puisqu'aucun littéral ne peut être vrai).

2.2.4.3 Conservation de la généralité

Si limiter le problème SAT aux formules CNF permet de vérifier facilement si une instanciation est un modèle, un inconvénient évident est l'impossibilité de traiter les formules propositionnelles ne vérifiant pas cette forme normale. Le théorème suivant montre toutefois que cette restriction ne provoque aucune perte de généralité de l'algorithme.

Théorème 2.1. *Pour toute formule propositionnelle φ, il existe une formule CNF équivalente φ', c'est-à-dire telle que :*

- $\mathcal{V}(\varphi) = \mathcal{V}(\varphi')$;
- $\forall \sigma : \mathcal{V}(\varphi) \nrightarrow \mathcal{B}, \sigma(\varphi) = \sigma(\varphi')$.

Démonstration. La transformation d'une formule propositionnelle quelconque en formule CNF équivalente se fait en plusieurs étapes successives :

1. L'implication et l'équivalence sont tout d'abord éliminées en les remplaçant par ces formes équivalentes :
   - $F \rightarrow G \equiv \neg F \vee G$
   - $F \leftrightarrow G \equiv (F \wedge G) \vee (\neg F \wedge \neg G)$

   Les seuls connecteurs binaires restants sont la conjonction et la disjonction.
2. Ensuite, les négations à l'extérieur des connecteurs binaires sont récursivement éliminées en utilisant les lois de De Morgan :
   - $\neg(F \vee G) \equiv (\neg F) \wedge (\neg G)$
   - $\neg(F \wedge G) \equiv (\neg F) \vee (\neg G)$
3. Après ces deux premières étapes, toutes les négations s'appliquent soit à une variable, soit à une autre négation. Dans ce second cas, la règle de double négation s'applique : $\neg\neg F \equiv F$.
4. Enfin, les disjonctions extérieures sont éliminées par la règle de distributivité suivante :

$$\bigvee_{i=1}^{m}\left(\bigwedge_{j=1}^{n_i} F_{i,j}\right) \equiv \bigwedge_{j_1=1}^{n_1}\left(\bigwedge_{j_2=1}^{n_2}\left(\ldots\left(\bigwedge_{j_m=1}^{n_m}\left(\bigvee_{i=1}^{m} F_{i,j_i}\right)\right)\ldots\right)\right)$$

□

Exemple 2.7. Utilisons cette stratégie pour construire une formule en CNF F' équivalente à la formule $F = (a \leftrightarrow \neg b) \rightarrow (c \wedge d)$:

$$\begin{aligned}
F &\equiv ((a \wedge \neg b) \vee (\neg a \wedge \neg\neg b)) \rightarrow (c \wedge d) && \text{(élimination de l'équivalence)}\\
&\equiv \neg((a \wedge \neg b) \vee (\neg a \wedge \neg\neg b)) \vee (c \wedge d) && \text{(élimination de l'implication)}\\
&\equiv (\neg(a \wedge \neg b) \wedge \neg(\neg a \wedge \neg\neg b)) \vee (c \wedge d) && \text{(loi de De Morgan)}\\
&\equiv ((\neg a \vee \neg\neg b) \wedge (\neg\neg a \vee \neg\neg\neg b)) \vee (c \wedge d && \text{(loi de De Morgan)}\\
&\equiv ((\neg a \vee b) \wedge (a \vee \neg b)) \vee (c \wedge d) && \text{(élimination des doubles négations)}
\end{aligned}$$

$$\equiv (\neg a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (\neg a \vee b \vee d) \wedge (a \vee \neg b \vee d) \qquad \text{(distributivité)}$$
$$= F'$$

Ce théorème nous assure que la satisfaisabilité de toute formule propositionnelle peut être vérifiée par un algorithme restreint aux formules CNF, en la convertissant en une formule CNF équivalente au préalable. Toutefois, la conversion peut provoquer une augmentation exponentielle de la taille de la formule. On peut en effet facilement constater que, lors de l'application de la règle de distributivité, si $n_1 = n_2 = \ldots = n_m = n$, chaque sous-formule $F_{i,j}$ est dupliquée n^{m-1} fois. Cette explosion potentielle de la taille des formules converties est évidemment en pratique un obstacle important à la résolution de SAT. Il est toutefois possible de contourner ce problème en passant par une formule CNF non pas équivalente à la formule originelle, mais **équisatisfaisable**, c'est-à-dire qui est satisfaisable si et seulement si la formule d'origine est satisfaisable. L'équisatisfaisabilité est donc une propriété moins forte que l'équivalence, notamment parce qu'elle n'impose pas de correspondance entre les variables des deux formules.

Théorème 2.2. *Pour toute formule propositionnelle φ, il existe une formule CNF φ' telle que :*

- *φ et φ' sont équisatisfaisables ;*
- *$\mathcal{V}(\varphi) \subseteq \mathcal{V}(\varphi')$;*
- *Si $\sigma : \mathcal{V}(\varphi') \to \mathcal{B}$ est un modèle de φ', alors $\sigma|_{\mathcal{V}(\varphi)}$, la restriction de σ à $\mathcal{V}(\varphi)$, est un modèle de φ ;*
- *Si $\sigma : \mathcal{V}(\varphi) \to \mathcal{B}$ est un modèle de φ, alors il existe un modèle $\sigma' : \mathcal{V}(\varphi') \to \mathcal{B}$ de φ' qui étend σ ;*
- *La taille de φ' est linéairement proportionnelle à celle de φ.*

Démonstration. La transformation de Tseitin (Tseitin, 1983) vérifie les critères de cette définition. Pour chaque connecteur binaire de φ, cette transformation introduit une nouvelle variable ainsi que plusieurs clauses pour encoder la signification de ce connecteur. Par exemple, dans le cas d'une conjonction $F \wedge G$, supposons que f et g sont les variables

récursivement associées aux sous-formules F et G par la transformation de Tseitin. Cette transformation introduit alors une nouvelle variable v pour représenter la conjonction, ainsi que les clauses suivantes pour en encoder la sémantique : $\neg v \vee f$, $\neg v \vee g$ et $v \vee \neg f \vee \neg g$. Les deux premières clauses signifient que si la conjonction est vraie, alors F et G respectivement sont vraies. La troisième signifie que si la conjonction est fausse, alors au moins une des deux sous-formules est fausse.

Tous les connecteurs sont encodés de cette façon, puis une dernière clause est ajoutée. Celle-ci est unitaire et contient uniquement une occurrence positive de la variable associée au connecteur le plus externe de la formule d'origine. Cette clause signifie que la formule d'origine est vraie, et garantit donc l'équisatisfaisabilité des formules. La croissance linéaire de la taille de la formule est facilement vérifiable car chaque type de connecteur est encodé par un nombre fixe de clause : par exemple, la conjonction est toujours encodée par trois clauses. Les propriétés sur le passage d'un modèle de φ à un modèle de φ' et vice-versa sont facilement démontrables par la structure de la transformation de Tseitin. □

Exemple 2.8. Appliquons la transformation Tseitin à la formule $F = (a \leftrightarrow \neg b) \rightarrow (c \wedge d)$. Soient $F_1 = a \leftrightarrow \neg b$ et $F_2 = c \wedge d$ les deux sous-formules de F non-réduites à un littéral, et v, v_1 et v_2 les variables introduites par l'encodage qui seront vraies si et seulement si F, F_1 et F_2 sont vraies respectivement.

- La valeur de v_1 en fonction des valeurs des variables a et b est encodée par les clauses $\neg v_1 \vee a \vee b$, $\neg v_1 \vee \neg a \vee \neg b$, $v_1 \vee a \vee \neg b$ et $v_1 \vee \neg a \vee b$. La conjonction des deux premières clauses implique que si v_1 est vraie, alors a et b ont obligatoirement des valeurs opposées, donc F_1 est vraie. Au contraire, si v_1 est fausse, les deux dernières clauses assurent que a et b ont la même valeur, et donc que F_1 est fausse.
- La valeur de v_2 en fonction des valeurs des variables c et d est encodée par les clauses $\neg v_2 \vee c$, $\neg v_2 \vee d$ et $v_2 \vee \neg c \vee \neg d$. Les deux premières clauses obligent respectivement c et d à être vraies si v_2 est vraie, tandis que la troisième spécifie qu'au moins une des variables doit être fausse si v_2 est fausse.

- Enfin, la valeur de v en fonction de v_1 et v_2 est encodée par les clauses $\neg v \vee \neg v_1 \vee v_2$, $v \vee v_1$ et $v \vee \neg v_2$. La première clause signifie que si v est vraie, alors soit v_1 est fausse, soit v_2 est vraie. Au contraire, les deux clauses suivantes imposent à v_1 d'être vraie et à v_2 d'être fausse si v est fausse.

Au final, la formule CNF obtenue par transformation de Tseitin est $F'' = v \wedge (\neg v \vee \neg v_1 \vee v_2) \wedge (v \vee v_1) \wedge (v \vee \neg v_2) \wedge (\neg v_1 \vee a \vee b) \wedge (\neg v_1 \vee \neg a \vee \neg b) \wedge (v_1 \vee a \vee \neg b) \wedge (v_1 \vee \neg a \vee b) \wedge (\neg v_2 \vee c) \wedge (\neg v_2 \vee d) \wedge (v_2 \vee \neg c \vee \neg d)$.

Comme le problème SAT est NP-complet, une croissance linéaire de la taille de la formule reste significative car elle peut suffire à induire une croissance exponentielle du temps d'exécution au pire des cas d'un algorithme complet. La conversion d'une formule en CNF peut avoir d'autres inconvénients, comme la perte d'informations structurelles ; c'est pourquoi certains solveurs sont conçus pour traiter directement des formules propositionnelles de forme quelconque (Thiffault, Bacchus et Walsh, 2004). Les solveurs ne traitant que les formules CNF restent toutefois les plus répandus.

2.2.5 Variantes du problème SAT

Il existe de nombreuses variantes de problèmes liés à la satisfaisabilité des formules propositionnelles. Parmi les plus courantes, nous pouvons citer les problèmes #SAT, All-SAT, Max-SAT et Max-SAT pondéré.

- Le problème #SAT, pour une formule propositionnelle donnée, consiste à donner le nombre de modèles de cette formule. Il s'agit du **problème de comptage** associé au problème de décision SAT.
- Le problème All-SAT est le **problème d'énumération** associé au problème SAT. Pour une formule propositionnelle, résoudre All-SAT revient à énumérer tous ses modèles, donc toutes les solutions du problème SAT correspondant.
- Le problème Max-SAT est un **problème d'optimisation** associé au problème SAT sur les formules en forme normale conjonctive. Il consiste à déterminer le nombre maximal de clauses qui peuvent être satisfaites simultanément par une

instanciation complète des variables de la formule considérée.

- Le problème Max-SAT pondéré est également un problème d'optimisation associé à SAT. Étant donnés une formule propositionnelle $F(\mathcal{V},\mathcal{C})$ et une fonction de pondération $\omega : \mathcal{C} \rightarrow \mathbb{R}_+^*$ qui associe à chaque clause un poids strictement positif, le coût d'une instanciation complète sur $\mathcal{V}$ est la somme des poids des clauses non satisfaites par cette instanciation. Le problème Max-SAT pondéré demande de déterminer le coût minimum d'une instanciation. Max-SAT pondéré est une généralisation de Max-SAT, puisque toute instance de Max-SAT peut être considéré comme une instance de Max-SAT pondéré où un poids identique est assigné à chaque clause.

Notons que toutes ces variantes de SAT sont NP-difficiles[3], puisque SAT peut être facilement résolu à partir de leurs réponses sur la même formule ; en effet, une formule propositionnelle est satisfaisable si et seulement si :

- la réponse de #SAT est non-nulle ;
- All-SAT retourne au moins un modèle ;
- la réponse de Max-SAT est identique au nombre de clauses de la formule ;
- la réponse de Max-SAT pondéré est nulle.

2.2.6 Applications du problème SAT

Comme nous l'avons vu à la sous-section 2.2.3, SAT est un problème NP-complet, et par conséquent tous les algorithmes corrects et complets connus qui le résolvent sont exponentiels au pire des cas. Malgré cet obstacle théorique, des recherches intensives ont permis d'améliorer significativement l'efficacité des solveurs SAT au cours des années, comme le montrent les sections 2.3 à 2.5. Cet intérêt dans la résolution efficace du problème SAT s'explique aisément par la grande quantité de problèmes qu'il permet d'encoder. En effet, la logique propositionnelle est un langage relativement intuitif, et de

3. Plus exactement, les problèmes de décision dérivés de ces problèmes sont NP-difficiles. Par exemple, le problème consistant à déterminer si le nombre de modèles d'une formule est égal à un entier donné est le problème de décision dérivé de #SAT.

nombreux problèmes de recherche se réduisent aisément en problèmes de satisfaisabilité propositionnelle.

De plus, l'existence de solveurs SAT performants facilite le développement d'outils pour la résolution de ces problèmes, puisqu'il suffit d'encoder le problème considéré en problème de satisfaisabilité, d'utiliser un solveur SAT, puis, le cas échéant, de décoder l'éventuel modèle de la formule afin de récupérer une solution au problème d'origine. Ce procédé se révèle souvent moins long à développer qu'un algorithme ad hoc entier. L'intérêt pour la résolution efficace de SAT et pour son utilisation comme sous-routine se sont entraînés mutuellement, les progrès dans l'efficacité des solveurs encourageant son utilisation pour encoder de nouveaux problèmes, ce qui en retour augmentait les retombées et donc l'intérêt de poursuivre les améliorations des solveurs.

Les applications de SAT appartiennent à des domaines variés. Dans celui de la conception et de la vérification électronique, SAT a été entre autres utilisé pour encoder des problèmes d'équivalence de circuits (Goldberg, Prasad et Brayton, 2001), de routage de circuits FPGA (Nam, Sakallah et Rutenbar, 1999) et pour effectuer des générations automatiques de tests (Stephan, Brayton et Sangiovanni-Vincentelli, 1996). En vérification formelle, SAT est utilisé dans la vérification de modèles bornée (Biere et al., 2003) et la vérification de microprocesseurs (Velev et Bryant, 2003). Parmi les autres applications, nous pouvons également citer la planification (Kautz et Selman, 1992), l'ordonnancement (Memik et Fallah, 2002), le diagnostic (Grastien et al., 2007), l'inférence d'haplotypes en bioinformatique (Lynce et Marques-Silva, 2008) et différentes applications en cryptanalyse (Mironov et Zhang, 2006; Eibach, Pilz et Völkel, 2008). SAT est également utilisé pour résoudre le problème voisin de satisfaction de contraintes (Argelich et al., 2012) ou comme composant de systèmes de raisonnements logiques plus complexes, comme les formules booléennes quantifiées (Biere, 2005), les « satisfaisabilité modulo théories » (Barrett et al., 2009), la programmation par ensembles réponses (Lin, Zhang et Hernandez, 2006), le calcul des événements (Mueller, 2004) et les raisonnements probabilistes (Saad, 2009).

2.3 L'algorithme DP

L'algorithme DP, nommé d'après ses concepteurs Davis et Putnam (1960), est un des plus anciens algorithmes relativement efficaces pour la résolution du problème SAT sur les formules CNF. À l'origine, l'algorithme est en fait destiné à vérifier la validité d'une formule logique du premier ordre et se base pour cela sur une procédure pour résoudre le problème SAT. C'est cependant cette seule partie de l'algorithme qui a été retenue comme contribution la plus significative ; par conséquent, on désigne généralement par le nom de DP uniquement cette procédure de satisfaisabilité propositionnelle et non pas l'algorithme au complet. Nous suivrons ici cette convention.

Contrairement à DPLL et CDCL, qui sont des algorithmes de recherche et tentent de trouver un modèle de la formule étudiée en assignant si besoin des valeurs arbitraires à certaines variables, DP repose uniquement sur des mécanismes d'inférence, c'est-à-dire sur des raisonnements logiques à partir de la formule d'origine.

La première sous-section introduit la notion de résolution de clauses. La seconde détaille l'algorithme DP. La dernière sous-section résume les propriétés de l'algorithme, notamment sa correction totale et sa complexité en temps et en espace.

2.3.1 Résolution

Soient une variable v et deux clauses c et c' telles que $c = l_1 \vee l_2 \vee \ldots \vee l_n \vee v$ et $c' = l'_1 \vee l'_2 \vee \ldots \vee l'_{n'} \vee \neg v$. La **résolution** de c et c' sur v est la clause $c \otimes_v c' = l_1 \vee l_2 \vee \ldots \vee l_n \vee l'_1 \vee l'_2 \vee \ldots \vee l'_{n'}$ ou, en notation ensembliste, $c \otimes_v c' = (c \setminus \{v\}) \cup (c' \setminus \{\neg v\})$. Cette clause est la disjonction de tous les littéraux de c et c' différents de v et $\neg v$ (en supposant que ni c ni c' ne contient à la fois v et $\neg v$). $c \otimes_v c'$ est une conséquence logique de c et c', c'est à dire que toute instanciation vérifiant c et c' vérifie également $c \otimes_v c'$. Nous allons définir formellement puis démontrer cette propriété.

Définition 2.5. *Soit $\Gamma = F_1, F_2, \ldots, F_n$ un ensemble de formules propositionnelles et G une formule propositionnelle. G est une* ***conséquence logique*** *de Γ, noté $\Gamma \models G$, si*

pour toute instanciation complète σ sur $\mathcal{V}(\Gamma \cup \{G\})$, $\sigma(\Gamma) = \text{vrai} \Rightarrow \sigma(G) = \text{vrai}$.

Si $n = 0$, alors G est une tautologie, et on note $\models G$.

L'interprétation d'un ensemble de formules $\Gamma = F_1, F_2, \ldots, F_n$ est définie par $\sigma(\Gamma) = \sigma(F_1 \wedge F_2 \wedge \ldots \wedge F_n)$.

Proposition 2.1. *Soient une variable v et deux clauses c et c' telles que $c = l_1 \vee l_2 \vee \ldots \vee l_n \vee v$ et $c' = l'_1 \vee l'_2 \vee \ldots \vee l'_{n'} \vee \neg v$. Alors $c, c' \models c \otimes_v c'$.*

Démonstration. Soit σ une instanciation complète telle que $\sigma(c \wedge c') = \text{vrai}$; par conséquent, $\sigma(c) = \text{vrai}$ et $\sigma(c') = \text{vrai}$. Si $\sigma(v) = \text{vrai}$, alors $\exists i \in \{1, \ldots, n'\} \,|\, \sigma(l'_i) = \text{vrai}$, puisque $\sigma(c') = \text{vrai}$. Comme $l'_i \in c \otimes_v c'$, $\sigma(c \otimes_v c') = \text{vrai}$. De même, si $\sigma(v) = \text{faux}$, alors $\exists i \in \{1, \ldots, n\} \,|\, \sigma(l_i) = \text{vrai}$, puisque $\sigma(c) = \text{vrai}$. Comme $l_i \in c \otimes_v c'$, $\sigma(c \otimes_v c') = \text{vrai}$. □

2.3.2 Description algorithmique de DP

DP est décrit par l'algorithme 2.1. Il consiste à progressivement éliminer toutes les variables de la formule par des transformations conservant la satisfaisabilité de la formule. Le type de transformation utilisé en priorité est celui de la **clause unitaire** (lignes 2 à 6) : si une clause de $\mathcal{C}$ contient un unique littéral l, alors tout modèle de $F(\mathcal{V}, \mathcal{C})$ contient obligatoirement l. Toutes les clauses $c \in \mathcal{C}$ contenant l ou $\neg l$ sont alors simplifiées selon cette contrainte :

- si $l \in c$, c est éliminée de $\mathcal{C}$, car instancier l satisfait obligatoirement la clause ;
- si $\neg l \in c$, $\neg l$ est retiré de c, car ce littéral est obligatoirement rendu faux par l'instanciation de l.

Si l'on note $\mathcal{C}'$ le nouvel ensemble de clauses ainsi obtenu et $F'(\mathcal{V}', \mathcal{C}')$ la formule associée, où $\mathcal{V}' = \mathcal{V} \setminus \{\nu(l)\}$, il est évident que :

- Si σ est un modèle de F, alors $l \in \sigma$ et $\sigma \setminus \{l\}$ est un modèle de F' ;
- Si σ' est un modèle de F', alors $\sigma \cup \{l\}$ est un modèle de F.

Algorithme 2.1 DP($F(\mathcal{V},\mathcal{C})$)

```
1: tant que C ≠ ∅ faire
2:     si ∃l ∈ L | {l} ∈ C alors /*clause unitaire*/
3:         C ← {c \ {¬l} | c ∈ C, l ∉ c}
4:         /*on élimine les clauses qui contiennent l*/
5:         /*et on retire ¬l des clauses restantes*/
6:         V ← V \ {ν(l)}
7:     sinon si ∃l ∈ L | ∀c ∈ C, ¬l ∉ C alors /*littéral pur*/
8:         C ← {c | c ∈ C, l ∉ c} /*on élimine les clauses qui contiennent l*/
9:         V ← V \ {ν(l)}
10:    sinon /*élimination d'une variable par résolution*/
11:        choisir v ∈ V
12:        C_v ← {c ∈ C | v ∈ c} /*C_v ≠ ∅*/
13:        C_¬v ← {c ∈ C | ¬v ∈ c} /*C_¬v ≠ ∅*/
14:        pour tout c_v ∈ C_v faire
15:            pour tout c_¬v ∈ C_¬v faire
16:                C ← C ∪ {c_v ⊗_v c_¬v}
17:                /*ajouter la résolution de c_v et c_¬v sur v*/
18:        C ← C \ (C_v ∪ C_¬v) /*éliminer les clauses contenant v*/
19:        V ← V \ {v}
20:    si ⊥ ∈ C alors /*la clause vide a été inférée*/
21:        retourner faux
22: retourner vrai /*la formule vide est atteinte*/
```

Si $\mathcal{C}$ ne contient aucune clause unitaire, alors l'algorithme recherche un éventuel **littéral pur**. Un littéral $l \in \mathcal{L}$ est pur si son littéral opposé $\neg l$ n'apparaît dans aucune clause de $\mathcal{C}$. Par conséquent, pour tout modèle σ de F, si $\neg l \in \sigma$, alors $(\sigma \setminus \{\neg l\}) \cup \{l\}$ est aussi un modèle de F. On peut alors transformer $\mathcal{C}$ en retirant toutes les clauses contenant l tout en conservant l'équisatisfaisabilité. En effet, grâce à la propriété énoncée précédemment, la transformation entre modèles de F et de F' dans le cas de la clause unitaire est également valide dans celui du littéral pur, à une nuance près : si σ est un modèle de F, alors on peut avoir $l \in \sigma$ ou $\neg l \in \sigma$. Cependant, dans tous les cas, $\sigma \setminus \{l, \neg l\}$ est un modèle de F'. Cette transformation est effectuée dans les lignes 7 à 9 de l'algorithme.

Si aucune de ces deux transformations n'est applicable, l'algorithme sélectionne une variable quelconque de la formule et l'élimine par résolution (lignes 11 à 19) : les clauses contenant la variable v sont remplacées par l'ensemble des clauses obtenues

par résolution sur v de toutes les paires de clauses contenant respectivement les littéraux v et $\neg v$. Plus formellement, soient $\mathcal{C}_v$ et $\mathcal{C}_{\neg v}$ les ensembles des clauses de $\mathcal{C}$ qui contiennent respectivement les littéraux v et $\neg v$. Soit $R = \{c_v \otimes_v c_{\neg v} \mid c_v \in \mathcal{C}_v,\ c_{\neg v} \in \mathcal{C}_{\neg v}\}$ l'ensemble des clauses obtenues par résolution d'une clause de $\mathcal{C}_v$ et d'une clause de $\mathcal{C}_{\neg v}$ sur la variable v. Alors $F(\mathcal{V},\mathcal{C})$ est transformée en $F'(\mathcal{V}',\mathcal{C}')$ où $\mathcal{V}' = \mathcal{V} \setminus \{v\}$ et $\mathcal{C}' = (\mathcal{C} \cup R) \setminus (\mathcal{C}_v \cup \mathcal{C}_{\neg v})$. Cette élimination par résolution préserve également la satisfaisabilité de la formule, comme le montrent les résultats suivants.

Proposition 2.2. *Soit $F'(\mathcal{V}',\mathcal{C}')$ la formule obtenue après l'élimination de v par résolution dans $F(\mathcal{V},\mathcal{C})$. Si σ est un modèle de F, alors $\sigma' = \sigma \setminus \{v, \neg v\}$ est un modèle de F'.*

Démonstration. Toute clause $c \in \mathcal{C}'$ est soit une clause originale de $\mathcal{C}$ ne contenant pas v, soit le produit de la résolution d'une clause $c_v \in \mathcal{C}_v$ et d'une clause $c_{\neg v} \in \mathcal{C}_{\neg v}$. Dans le premier cas, il est évident que c est satisfaite par σ', puisqu'elle est par définition satisfaite par σ.

Dans le second cas, comme σ est un modèle de F, $\sigma(c_v) = \sigma(c_{\neg v}) =$ vrai. Par la proposition 2.1, on a donc $\sigma(c) =$ vrai (car $c = c_v \otimes_v c_{\neg v}$), donc $\sigma'(c) =$ vrai puisque $v \notin \nu(c)$.

Par conséquent, σ' satisfait toutes les clauses de F'. □

Proposition 2.3. *Soit $F'(\mathcal{V}',\mathcal{C}')$ la formule obtenue après l'élimination de v par résolution dans $F(\mathcal{V},\mathcal{C})$. Si σ' est un modèle de F', alors σ' peut être étendu en un modèle de F.*

Démonstration. Soient $\sigma_v = \sigma' \cup \{v\}$ et $\sigma_{\neg v} = \sigma' \cup \{\neg v\}$ les deux extensions possibles de σ' sur $\mathcal{V}$. Trivialement, toutes les clauses de $\mathcal{C}_v$ sont satisfaites par σ_v et toutes les clauses de $\mathcal{C}_{\neg v}$ sont satisfaites par $\sigma_{\neg v}$. Soit $c \in \mathcal{C} \setminus (\mathcal{C}_v \cup \mathcal{C}_{\neg v})$. Par définition de $\mathcal{C}_v$ et $\mathcal{C}_{\neg v}$, c ne contient pas la variable v ; par conséquent, $c \in \mathcal{C}'$. Comme σ' est un modèle de F', $\sigma'(c) =$ vrai. σ_v et $\sigma_{\neg v}$ sont des extensions de σ', donc $\sigma_v(c) = \sigma_{\neg v}(c) = \sigma'(c) =$ vrai.

Pour montrer qu'une extension de σ' satisfait F, il suffit donc de montrer qu'elle satisfait $\mathcal{C}_v$ et $\mathcal{C}_{\neg v}$. Nous allons montrer que soit toutes les clauses de $\mathcal{C}_v$, soit toutes les clauses de $\mathcal{C}_{\neg v}$ sont satisfaites quelle que soit l'instanciation de v. Il suffit alors d'instancier v de façon à satisfaire $\mathcal{C}_{\neg v}$ ou $\mathcal{C}_v$ respectivement.

Pour toute clause $c_v \in \mathcal{C}_v$, soit $c'_v = c_v \setminus \{v\}$ la clause obtenue en retirant v ; similairement, pour toute clause $c_{\neg v} \in \mathcal{C}_{\neg v}$, soit $c'_{\neg v} = c_{\neg v} \setminus \{\neg v\}$. Si $\forall c_v \in \mathcal{C}_v$, c'_v est satisfaite par σ', alors $\sigma_{\neg v}$ satisfait toutes les clauses de $\mathcal{C}_v$; par conséquent, $\sigma_{\neg v}$ est un modèle de F.

Sinon, soit $c_v \in \mathcal{C}_v$ telle que $\sigma'(c'_v) =$ faux. Pour toute clause $c'_{\neg v} \in \mathcal{C}_{\neg v}$, $c_v \otimes_v c_{\neg v} = c'_v \cup c'_{\neg v}$, donc, par définition, $c'_v \cup c'_{\neg v} \in \mathcal{C}'$. Puisque σ' est un modèle de F', $\sigma'(c'_v \cup c'_{\neg v}) =$ vrai. Comme $\sigma'(c'_v) =$ faux, on a forcément $\sigma'(c'_{\neg v}) =$ vrai. Par conséquent, $\forall c_{\neg v} \in \mathcal{C}_{\neg v}, \sigma_v(c_{\neg v}) =$ vrai. L'instanciation σ_v est donc un modèle de F. □

DP élimine donc successivement toutes les variables du problème à l'aide de ces trois types de transformations conservant la satisfaisabilité de la formule. Si la clause vide est produite au cours d'une transformation (ligne 20), cela prouve que la formule courante, ainsi que la formule d'origine, sont insatisfaisables. Si toutes les variables sont éliminées sans générer de clause vide, la formule vide est atteinte. Celle-ci étant satisfaisable, la formule d'origine l'est également.

L'ordre de priorité des trois types de transformation n'est pas essentiel aux propriétés de DP. Cet ordre correspond en fait à l'impact des transformations sur la taille de la formule transformée : DP applique de préférences les règles les plus susceptibles de réduire cette taille. En effet, la règle de clause unitaire élimine certaines clauses de la formule ainsi qu'un littéral dans certaines de clauses conservées. La règle de littéral pur élimine également certaines clauses mais sans modifier les clauses conservées. La règle de résolution est quant à elle la seule règle pouvant augmenter le nombre de clauses de la formule ; c'est donc pour cette raison qu'elle n'est utilisée que lorsque les deux premières règles ne peuvent être appliquées.

2.3.3 Propriétés et complexité de DP

DP est correct, complet et il termine. La correction et la complétude sont une conséquence de la conservation de l'équisatisfaisabilité par les trois type de transformation que nous avons prouvée dans la sous-section précédente[4]. L'algorithme termine au pire des cas après avoir éliminé toutes les variables, et chaque type d'élimination manipule un nombre fini de clauses : les règles de clause unitaire et de littéral pur éliminent et modifient un certain nombre de clauses mais n'en créent aucune. La règle de résolution élimine une variable v en retirant les m_v clauses contenant v et les $m_{\neg v}$ clauses contenant $\neg v$ et en rajoutant les $m_v \times m_{\neg v}$ clauses obtenues par résolution sur v.

DP n'est pas un algorithme constructif, puisque dans le cas d'une formule satisfaisable, il détermine cette satisfaisabilité sans avoir à trouver un modèle de la formule. On peut toutefois le modifier afin de le rendre constructif, mais cela nécessite de conserver toutes les clauses produites par résolution au cours de l'algorithme et aggrave donc le problème d'espace mémoire évoqué ci-après.

La complexité temporelle de DP est exponentielle au pire des cas. Cette complexité découle du rapport entre l'algorithme DP et les preuves de réfutation par résolution.

Une **dérivation par résolution** de la clause c à partir de l'ensemble $\mathcal{C}$ est un graphe orienté acyclique $G(\mathcal{S}, \mathcal{A})$ tel que :

- une fonction $\gamma : \mathcal{S} \rightarrow \mathcal{C}(\mathcal{L})$ associe à tout sommet $s \in \mathcal{S}$ une clause $\gamma(s)$;
- il existe un sommet qui correspond au but de la dérivation, c'est-à-dire à la clause c : $\exists b \in \mathcal{S} \mid \gamma(b) = c$;
- tout sommet a soit deux arcs entrant, soit aucun arc entrant ;
- si un sommet $s \in \mathcal{S}$ n'a aucun arc entrant, alors $\gamma(s) \in \mathcal{C}$;
- b n'a aucun arc sortant ;
- $\forall s_1, s_2, s_3 \in \mathcal{S}$, si $(s_1, s_3) \in \mathcal{A}$ et $(s_2, s_3) \in \mathcal{A}$, alors $\exists v \in \mathcal{V}$ telle que $\gamma(s_1) \otimes_v \gamma(s_2) = \gamma(s_3)$.

4. Des preuves différentes ont été proposées dans (Davis et Putnam, 1960)

Plus informellement, c est dérivable par résolution à partir de $\mathcal{C}$ si elle peut être obtenue par une suite de résolutions entre les clauses de $\mathcal{C}$ et les clauses précédemment obtenues par de telles résolutions.

Une **réfutation par résolution** d'un ensemble de clauses $\mathcal{C}$ (ou d'une formule $F(\mathcal{V},\mathcal{C})$) est une dérivation de $\perp$ à partir de $\mathcal{C}$. Une dérivation (ou réfutation) par résolution est dite **régulière** si, sur tout chemin d'une clause de $\mathcal{C}$ vers c (ou $\perp$), toutes les résolutions successives sont effectuées sur des variables différentes.

Au cours d'une preuve d'insatisfaisabilité par DP, toutes les clauses crées ou modifiées par l'algorithme le sont par résolution de clauses initiales ou précédemment dérivées par résolution.[5] Ces suites de résolutions constituent donc une réfutation par résolution. Or, il existe des familles de formules propositionnelles insatisfaisables telles que, pour une formule à n variables, le nombre de clauses de la formule est polynomial selon n, mais toute réfutation par résolution nécessite la dérivation de c^n clauses, où c est une constante strictement positive, lorsque n est suffisamment grand Haken (1985).[6] Par conséquent, toute exécution de DP sur ces formules demande un temps exponentiel selon la taille de la formule, quelque soit l'ordre d'élimination des variables.

La complexité spatiale de DP, quant à elle, est plus délicate à analyser, puisque l'algorithme élimine également des clauses au fil des étapes de résolution. Il n'existe pas, à notre connaissance, de résultats théoriques indiquant si la complexité spatiale de DP est ou non exponentielle au pire des cas. Toutefois, il est clair que le grand nombre de clauses générées et stockées par l'algorithme est en pratique un inconvénient important et provoque facilement une explosion de la mémoire nécessaire. Cette consommation excessive de l'espace mémoire explique le plus grands succès des algorithmes de recherche,

---

5. Notons que les transformations de clauses lors de l'application de la règle de clause unitaire (pour une clause unitaire $\{l\}$, toutes les occurrences du littéral $\neg l$ sont retirées) peuvent être considérées comme des résolutions de ces clauses avec la clause unitaire $\{l\}$.

6. L'article cité traite de la preuve de tautologie de formules propositionnelles en forme normale disjonctive, ce qui est un problème équivalent à l'insatisfaisabilité de formules en forme normale conjonctive.

comme DPLL, dont la complexité spatiale est seulement linéaire.

2.4 L'algorithme DPLL

DPLL, comme DP, est un algorithme nommé d'après ses concepteurs, Davis, Logemann et Loveland (1962), auxquels on ajoute généralement le nom de Putnam. En effet, l'algorithme DPLL a été à l'origine conçu comme une simple optimisation pratique de DP : l'étape de résolution de DP, qui peut provoquer une explosion de la taille de la formule et donc de l'espace mémoire requis, est remplacée par une analyse par cas, c'est-à-dire par des prises de décisions arbitraires. Cette modification a un impact significatif sur la nature de l'algorithme, puisqu'elle fait de DPLL un algorithme de recherche, qui parcourt explicitement l'espace des diverses instanciations possibles.

Le remplacement des résolutions par des décisions a l'effet escompté sur l'efficacité de l'algorithme : la complexité spatiale de DPLL est linéaire selon la taille de la formule. En pratique, cet avantage a fait de DPLL l'algorithme complet le plus couramment utilisé pour la résolution de SAT jusqu'au milieu des années 90. L'algorithme CDCL (voir section 2.5), qui l'a détrôné, est lui-même une évolution de DPLL.

La sous-section 2.4.1 décrit en détail l'algorithme DPLL. La sous-section 2.5.8 énumère ses principales propriétés. La sous-section 2.4.3 aborde les heuristiques de choix des décisions de l'algorithme. Enfin, la sous-section 2.4.4 illustre les occurrences multiples de conflits dans DPLL, un des principaux inconvénients de cet algorithme, qui a motivé la conception de CDCL.

2.4.1 Description algorithmique de DPLL

Cette sous-section décrit l'algorithme DPLL. Elle énonce tout d'abord ses principes de base informellement, puis le présente selon deux paradigmes différents : tout d'abord comme un algorithme récursif, qui est sa description la plus naturelle, puis comme un algorithme itératif, qui correspond à la façon dont DPLL est généralement implémenté.

2.4.1.1 Principe de base

Contrairement à DP, DPLL est un algorithme constructif : au lieu de transformer successivement la formule de départ, DPLL conserve cette formule tout au long de son exécution, mais cherche à construire progressivement un modèle de la formule en partant de l'instanciation vide. Les règles de clause unitaire et de littéraux purs sont toujours utilisées dans DPLL mais sont définies par rapport à l'instanciation courante σ : une clause est considérée comme unitaire non pas si elle contient un seul littéral, mais si un de ses littéraux l est indéfini et tous les autres sont faux selon σ. De même, un littéral l est pur non pas si $\neg l$ n'apparaît dans aucune clause, mais si tout clause où $\neg l$ apparaît est déjà satisfaite (elle contient un littéral vrai). Dans les deux cas, la formule n'est pas modifiée, mais l est ajoutée à l'instanciation courante. Dans le premier cas, on parle alors d'une **propagation unitaire** : l est dite propagée par c.

Comme DP, DPLL utilise prioritairement les règles de clause unitaire et de littéral pur lorsque c'est possible. Dans le cas contraire, au lieu d'éliminer une variable v par résolution, DPLL choisit arbitrairement $l \in \{v, \neg v\}$ et continue la recherche en ajoutant l à l'instanciation courante σ ; il s'agit d'une **décision**. Si un modèle est trouvé, la formule est prouvée satisfaisable ; dans le cas contraire, la recherche retourne à l'état de l'instanciation σ avant cette décision et teste le choix inverse $\neg l$. Si celui-ci échoue également, la formule ne peut pas être satisfaite par un modèle qui étend σ.

DPLL explore explicitement l'espace de recherche des diverses instanciations des variables ; il s'agit donc d'un algorithme de recherche. Plus précisément, DPLL est un algorithme de recherche en profondeur : pour une instanciation courante σ, il explore tout d'abord entièrement l'espace de recherche des instanciations qui étendent σ avant de défaire partiellement σ si nécessaire.

Comme dans le cas de DP, l'ordre des règles utilisées n'a pas d'incidence sur la correction totale de DPLL mais est motivé par leur impact sur la recherche, plus précisément sur la taille de l'espace de recherche. En effet, pour une instanciation courante

σ, appliquer la règle de clause unitaire ou de littéral pur rajoute un littéral l à l'instanciation, ce qui réduit donc de moitié l'espace de recherche à explorer. Dans le cas d'une décision d'un littéral l, l'algorithme explore dans le pire des cas les deux sous-espaces $\sigma \cup \{l\}$ et $\sigma \cup \{\neg l\}$ successivement, ce qui ne contribue donc aucunement à élaguer l'espace de recherche.

Notons qu'en pratique, de nombreuses implémentations de DPLL omettent la règle de littéral pur. En effet, celle-ci est considérée comme peu efficace, la détection des littéraux purs étant trop coûteuse par rapport aux bénéfices de l'élagage de l'espace de recherche. Les propagations unitaires sont en revanche toujours implémentées, car elles sont très fréquentes et permettent d'élaguer considérablement l'espace de recherche ; la détection des clauses unitaires est donc très souvent payante, même si elle est implémentée de façon naïve. De plus, des techniques de détection des clauses unitaires de plus en plus efficaces ont été conçues : la technique des littéraux surveillés (Moskewicz et al., 2001) est présentement la plus répandue (voir la sous-section 2.5.2).

2.4.1.2 Description récursive

Compte tenu de la structure de DPLL, il est naturel de le décrire par un algorithme récursif (algorithme 2.3). Celui-ci prend comme paramètre une formule F et une instanciation σ, et cherche à déterminer s'il existe un modèle de F qui étend σ. Il est clair que ce n'est pas le cas si σ falsifie tous les littéraux d'une clause de F. Sinon, si σ est complète, elle est un modèle de F. Si aucun de ces cas ne s'applique et s'il existe une clause unitaire ou un littéral pur, l'algorithme est rappelé en ajoutant à σ le littéral correspondant. Sinon, pour un littéral l arbitraire pas encore assigné, l'algorithme récursif est rappelé successivement avec les instanciations $\sigma \cup \{l\}$ puis $\sigma \cup \{\neg l\}$. La réponse est alors positive si au moins un de ces appels récursifs a une réponse positive. F est satisfaisable si et seulement s'il existe un modèle qui étend $\emptyset$, c'est-à-dire un modèle quelconque ; il s'agit donc du cas de base de la récursion (algorithme 2.2).

Algorithme 2.2 DPLL-RÉCURSIF-BASE($F(\mathcal{V},\mathcal{C})$)

1: **retourner** DPLL-RÉCURSION($F(\mathcal{V},\mathcal{C}), \emptyset$)

Algorithme 2.3 DPLL-RÉCURSION($F(\mathcal{V},\mathcal{C}), \sigma$)

1: **si** $\exists c \in \mathcal{C} \mid \sigma(c) =$ faux **alors** /*un conflit est atteint*/
2: **retourner** faux
3: **sinon si** $\mathcal{D}(\sigma) = \mathcal{V}$ **alors** /*σ est une instanciation complète*/
4: **retourner** σ /*σ est un modèle de F*/
5: **sinon si** $\exists l \in \mathcal{L}, c \in \mathcal{C} \mid l \in c, \sigma(l) =$ indéfini, $\forall l' \in c \setminus \{l\}, \sigma(l') =$ faux **alors**
6: /*c est unitaire*/
7: **retourner** DPLL-RÉCURSION($F(\mathcal{V},\mathcal{C}), \sigma \cup \{l\}$)
8: /*tout modèle de F qui étend σ contient l*/
9: **sinon si** $\exists l \in \mathcal{L} \mid \forall c \in \mathcal{C}, \sigma(c) \neq$ vrai $\Rightarrow \neg l \notin \mathcal{C}$ **alors**
10: /*l est un littéral pur*/
11: **retourner** DPLL-RÉCURSION($F(\mathcal{V},\mathcal{C}), \sigma \cup \{l\}$)
12: /*s'il existe un modèle de F qui étend σ,*/
13: /*il existe aussi un modèle de F qui étend $\sigma \cup \{l\}$*/
14: **sinon** /*décision*/
15: **choisir** $l \in \mathcal{L} \mid \sigma(l) =$ indéfini
16: **si** DPLL-RÉCURSION($F(\mathcal{V},\mathcal{C}), \sigma \cup \{l\}$) = vrai **alors**
17: **retourner** vrai
18: **sinon**
19: **retourner** DPLL-RÉCURSION($F(\mathcal{V},\mathcal{C}), \sigma \cup \{\neg l\}$)

2.4.1.3 Description itérative

Si DPLL se décrit plus facilement comme un algorithme récursif, il est toutefois en général implémenté itérativement dans un soucis d'efficacité. L'algorithme itératif 2.4 est équivalent à l'algorithme récursif ci-dessus. Il utilise quelques structures de données additionnelles afin de conserver des repères sur l'espace de recherche déjà parcouru.

Les décisions successives sont numérotées de 1 à n (on ne considère à tout moment que les décisions qui n'ont pas encore été défaites ; lorsqu'une décision est défaite, on suppose que l'on peut réutiliser son numéro ultérieurement). À chaque décision correspond un niveau de décision : le niveau de décision $i \in \{1, \ldots, n\}$ regroupe la $i^{\text{ième}}$ décision et toutes les variables instanciées après cette décision et avant la décision suivante, si elle existe. Par convention, le pseudo-niveau de décision 0 regroupe les variables instanciées avant la première décision. À tout moment de la recherche, s'il y a n décisions actives,

Algorithme 2.4 DPLL-ITÉRATIF($F(\mathcal{V},\mathcal{C})$)

1: $\Lambda \leftarrow \{0\}$
2: $\lambda_c \leftarrow 0$ /*on commence au pseudo-niveau de décision 0*/
3: $\sigma \leftarrow \emptyset$ /*on commence avec l'affectation vide*/
4: **boucle**
5: **si** $\exists c \in \mathcal{C} \mid \sigma(c) =$ faux **alors** /*c est un conflit*/
6: **si** $\forall i \in \Lambda \setminus \{0\}$, *DécisionInversée*$(i) =$ vrai **alors**
7: /*toutes les décisions ont déjà été inversées*/
8: **retourner** faux /*c ne peut pas être satisfaite*/
9: **sinon**
10: **tant que** *DécisionInversée*$(\lambda_c) =$ vrai **faire**
11: RETOURARRIÈRE()
12: $l \leftarrow \delta(\lambda_c)$
13: DÉFAIRENIVEAU(λ_c)
14: INSTANCIER$(\neg l)$
15: $\delta(\lambda_c) \leftarrow \neg l$
16: *DécisionInversée*$(\lambda_c) \leftarrow$ vrai
17: **sinon si** $\mathcal{D}(\sigma) = \mathcal{V}$ **alors** /*toutes les variables sont instanciées*/
18: **retourner** vrai /*σ est un modèle de F*/
19: **si** $\exists l \in \mathcal{L}$, $c \in \mathcal{C} \mid l \in c$, $\sigma(l) =$ indéfini, $\forall l' \in c \setminus \{l\}$, $\sigma(l') =$ faux **alors**
20: INSTANCIER(l) /*c est unitaire*/
21: **sinon si** $\exists l \in \mathcal{L} \mid \sigma(l) =$ indéfini, $\forall c \in \mathcal{C}$, $\sigma(c) \neq$ vrai $\Rightarrow \neg l \notin c$ **alors**
22: INSTANCIER(l) /*l est un littéral pur*/
23: **sinon** /*aucune nouvelle inférence possible, prendre une décision*/
24: $\lambda_c \leftarrow$ NOUVEAUNIVEAU()
25: $\Lambda \leftarrow \Lambda \cup \{\lambda_c\}$
26: **choisir** $l \in \mathcal{L} \mid \sigma(l) =$ indéfini
27: INSTANCIER(l)
28: $\delta(\lambda_c) \leftarrow l$
29: *DécisionInversée*$(\lambda_c) \leftarrow$ faux

Algorithme 2.5 INSTANCIER(l) *[DPLL]*

Requis : $\nu(l) \notin \mathcal{D}(\sigma)$

1: $v \leftarrow \nu(l)$
2: $\sigma(v) \leftarrow \rho(l)$
3: $\lambda(v) \leftarrow \lambda_c$

Algorithme 2.6 DÉFAIRENIVEAU(i) *[DPLL]*

1: **pour** $v \in \mathcal{V} \mid \lambda(v) = i$ **faire**
2: $\sigma(v) \leftarrow$ indéfini
3: $\lambda(v) \leftarrow$ indéfini
4: $\alpha(v) \leftarrow$ indéfini

$\Lambda = \{0, 1, \ldots, n\}$ est l'ensemble des niveaux de décision.

$\forall i \in \Lambda \setminus \{0\}$, nous noterons $\delta(i)$ la décision du niveau i, soit la $i^{\text{ième}}$ décision. *DécisionInversée* : $\Lambda \setminus \{0\} \rightarrow \mathcal{B}$ est une fonction qui est vraie pour un niveau de décision i si l'algorithme a déjà au préalable testé la valeur opposée de cette décision avant de la renverser. La fonction partielle $\lambda : \mathcal{V} \nrightarrow \Lambda$ associe à toute variable $v \in \mathcal{V}$ son niveau de décision $\lambda(v)$. λ est définie pour toute variable instanciée, donc $\mathcal{D}(\lambda) = \mathcal{D}(\sigma)$. Enfin, à tout moment de la recherche, $\lambda_c = \max(i \in \Lambda)$ est le niveau de décision courant, c'est-à-dire qui correspond à la décision la plus récente.

Tant qu'aucune clause ne devient fausse, l'algorithme, comme dans la version récursive, utilise les règles de clause unitaire ou de littéral pur si possible ; sinon, un NOUVEAUNIVEAU de décision est créé et une décision y est prise. Lorsqu'un conflit est atteint, c'est-à-dire qu'une clause fausse est détectée, l'instanciation courante ne peut être étendue à un modèle de la formule ; l'algorithme cherche à la modifier en inversant la décision la plus récente, si possible (lignes 12 à 16) ; toutes les autres instanciations de ce niveau de décision sont défaites, puisqu'elles étaient des conséquences de l'ancienne décision. Si la dernière décision avait déjà été inversée auparavant, cela correspond dans l'algorithme récursif au cas où les deux appels récursifs correspondant à une décision échouent. Par conséquent, le niveau courant est entièrement défait et l'algorithme retourne au niveau précédent : il s'agit d'un **retour arrière**. L'algorithme défait successivement tous les niveaux par retour arrière jusqu'à atteindre une décision qu'il peut inverser ; s'il n'existe

Algorithme 2.7 RETOURARRIÈRE() *[DPLL]*

1: DÉFAIRENIVEAU(λ_c)
2: $\delta(\lambda_c) \leftarrow$ indéfini
3: $\Lambda \leftarrow \Lambda \setminus \{\lambda_c\}$
4: $\lambda_c \leftarrow \max(i \in \Lambda)$

pas de telle décision, la formule considérée est insatisfaisable (lignes 6 à 8). Elle est satisfaisable si l'algorithme instancie toutes les variables sans falsifier aucune clause.

2.4.2 Propriétés et complexité de DPLL

Il est facile de démontrer que DPLL est correct, complet et qu'il termine. Sa correction est évidente puisqu'il déclare une formule satisfaisable seulement s'il détecte explicitement un modèle.

DPLL est complet parce qu'il effectue un parcours systématique de l'espace de recherche. Toute partie de cet espace élaguée par une inversion de décision ou un retour arrière est insatisfaisable, car toute instanciation complète dans cet espace étend une instanciation qui rend fausse une clause du problème. De même, toute instanciation de l'espace de recherche élagué par une propagation unitaire falsifie cette clause unitaire. Par conséquent, pour une implémentation ne contenant pas la règle de littéral pur, toute partie de l'espace de recherche élaguée par l'algorithme ne contient aucun modèle, ce qui prouve la complétude de l'algorithme.

La règle de littéral pur peut quant à elle élaguer une partie de l'espace de recherche contenant un modèle mais, si c'est le cas, il existe également un modèle dans la partie de l'espace conservée. Par conséquent, les implémentations de DPLL contenant la règle de littéral pur sont également complètes.

La terminaison de DPLL est plus facile à prouver au travers de la version récursive de l'algorithme. Chaque appel récursif à DPLL-RÉCURSION ajoute un littéral à l'instanciation courante σ. Comme le nombre de variables de la formule est fini, la profondeur de récursion est également finie. Le pseudo-code de DPLL-RÉCURSION est lui-même fini

(il ne comporte aucune boucle), donc DPLL termine.

Le temps d'exécution de DPLL est exponentiel au pire des cas. Cette borne découle de la propriété suivante :

Proposition 2.4. *Toute exécution de DPLL sur une formule insatisfaisable construit implicitement une réfutation arborescente par résolution dont le nombre de sommets est au plus le double du nombre d'appels à* DPLL-RÉCURSION.

Une **réfutation arborescente** est une réfutation où tout sommet a au plus un arc sortant. Par conséquent, si une même clause c est utilisée dans n résolutions différentes, alors la réfutation contient n sommets distincts associés à c.

Cette propriété a été précédemment prouvée sous une forme légèrement différente par Goldberg (1979).

Démonstration. Supposons que la version récursive de DPLL a été exécutée sur une formule insatisfaisable $F(\mathcal{V}, \mathcal{C})$. Nous représenterons les appels successifs à DPLL-RÉCURSION par un arbre $T(\mathcal{N}, \mathcal{A})$ enraciné en $r \in \mathcal{N}$ tel que tout nœud $n \in \mathcal{N}$ correspond à un appel, et pour tout arc (n_1, n_2) [7], n_2 a effectué l'appel récursif n_1. $\forall n \in \mathcal{N}$, soit σ_n l'instanciation avec laquelle l'appel n a été effectuée (on a $\sigma_r = \emptyset$). $\forall n \in \mathcal{N} \setminus \{r\}$, soient $p(n)$ le père du nœud n et $l(n) = \sigma(n) \setminus \sigma(p(n))$ le nouveau littéral ajouté à l'instanciation courante par l'appel n.

Nous allons construire une réfutation arborescente $T'(\mathcal{N}', \mathcal{A}')$ enracinée en r'. $\gamma(n')$ est la clause associée à tout nœud $n' \in \mathcal{N}'$. Une fonction $\epsilon : \mathcal{N} \to \mathcal{N}'$ associe a tout nœud de T un nœud de T' (ϵ n'est ni injective ni surjective : certains nœuds de T' ne correspondent à aucun nœud de T, d'autres peuvent correspondre à plusieurs nœuds de T). De plus, ϵ est telle que $\epsilon(r) = r'$ et, si $(n_1, n_2) \in \mathcal{A}$ et $\epsilon(n_1) \neq \epsilon(n_2)$, alors $(\epsilon(n_1), \epsilon(n_2)) \in \mathcal{A}'$. Enfin, pour tout nœud $n \in \mathcal{N}$, $\sigma_n(\gamma(\epsilon(n))) =$ faux. Intuitivement, cela signifie que pour chaque appel récursif, on peut dériver une clause qui est fausse

7. Rappelons que selon nos conventions n_2 est le nœud le plus proche de la racine.

selon l'instanciation courante par résolutions successives à partir des clauses originales du problème. De plus, cette propriété garantit que $\gamma(r') = \bot$ (puisque $\sigma_r = \emptyset$) et donc que T' est bien une réfutation. La borne sur la taille de la réfutation découle du fait que pour chaque appel récursif, on y rajoute au plus deux nœuds.

Construisons T' et γ par récursion sur T, en s'assurant que toute clause associée à une feuille de T' est bien une clause d'origine de la formule :

Case de base : soit $n \in \mathcal{N}$ une feuille de T. Comme $F(\mathcal{V}, \mathcal{C})$ est insatisfaisable et que l'appel n n'a pas effectué de nouvel appel récursif, cela signifie qu'il existe une clause $c \in \mathcal{C}$ telle que $\sigma_n(c) =$ faux. Il suffit alors d'ajouter un nouveau nœud $\epsilon(n)$ à $\mathcal{N}'$ et de définir $\gamma(\epsilon(n)) = c$.

Récursion : soit $n \in \mathcal{N}$ un nœud interne de T dont le(s) fils vérifie(nt) les propriétés demandées. Si n a au moins un fils $n' \in f(n)$ tel que $\neg l(n') \notin \gamma(\epsilon(n'))$, alors $\sigma_n(\gamma(\epsilon(n'))) =$ faux, puisque $\sigma_{n'}(\gamma(\epsilon(n'))) =$ faux par hypothèse de récurrence. Il suffit alors de définir $\epsilon(n) = \epsilon(n')$ pour respecter les spécifications. Supposons le contraire, c'est-à-dire que pour tout fils $n' \in f(n)$ de n on a $\neg l(n') \in \gamma(\epsilon(n'))$; nous appellerons cette condition l'hypothèse de filiation. Elle signifie que la clause qui correspond au nœud de la réfutation $\epsilon(n')$ contient l'opposé du littéral ajouté à l'instanciation courante par l'appel n'. L'analyse dépend alors du type de règle appliqué par le nœud n.

Si n a appliqué une règle de littéral unitaire, alors ce nœud a un seul fils : $f(n) = \{n'\}$. Soit $l = l(n')$. Puisqu'une propagation unitaire a été effectuée, il existe une clause $c \in \mathcal{C}$ qui contient l et dont tous les autres littéraux sont faux selon σ_n. Nous savons que $\neg l \in \gamma(\epsilon(n'))$ (par hypothèse de filiation) et $\sigma_{n'}(\gamma(\epsilon(n'))) =$ faux (par récurrence). Par conséquent, on peut résoudre c et $\gamma(\epsilon(n'))$ sur $\nu(l)$ et $\sigma_n(c \otimes_{\nu(l)} \gamma(\epsilon(n'))) =$ faux. Nous vérifierons donc les propriétés demandées en ajoutant deux nœuds n" et $\epsilon(n)$ à $\mathcal{N}'$, les arcs $(n", \epsilon(n))$ et $(\epsilon(n'), \epsilon(n))$ à $\mathcal{A}'$, et en définissant $\gamma(n") = c$ et $\gamma(\epsilon(n)) = c \otimes_{\nu(l)} \gamma(\epsilon(n'))$. Notons que n" est une feuille de la réfutation et que la clause qui lui est associée est bien une clause originale de F.

Si n a appliqué une règle de littéral pur, alors ce nœud a un seul fils $n' \in \mathcal{N}$ et $l = l(n')$ est le littéral pur utilisé par n pour réaliser l'appel n' ; par conséquent, toute clause contenant $\neg l$ est vraie selon σ_n ou toute instanciation qui l'étend. Par récurrence, on sait que $\sigma_{n'}(\gamma(\epsilon(n'))) =$ faux, donc $\neg l \notin \gamma(\epsilon(n'))$, ce qui est contradictoire avec l'hypothèse de filiation. Celle-ci ne peut donc être vérifiée lors de l'application d'une règle de littéral pur.

Sinon, n a appliqué une règle de décision et il a deux fils : $f(n) = \{n_1, n_2\} \subseteq \mathcal{N}$. Soit $l = l(n_1)$. On a $\sigma_{n_1}(\gamma(\epsilon(n_1))) = \sigma_{n_2}(\gamma(\epsilon(n_2))) =$ faux (par récurrence), $l \in \gamma(\epsilon(n_1))$ et $\neg l \in \gamma(\epsilon(n_2))$ (par hypothèse de filiation). On peut donc résoudre $\gamma(\epsilon(n_1))$ et $\gamma(\epsilon(n_2))$ sur $\nu(l)$ et $\sigma_n(\gamma(\epsilon(n_1)) \otimes_{\nu(l)} \gamma(\epsilon(n_2))) =$ faux. On respecte les propriétés demandées en ajoutant un nouveau nœud $\epsilon(n)$ à $\mathcal{N}'$, deux arcs $(\epsilon(n_1), \epsilon(n))$ et $(\epsilon(n_2), \epsilon(n))$ à $\mathcal{A}'$ et en définissant $\gamma(\epsilon(n)) = \gamma(\epsilon(n_1)) \otimes_{\nu(l)} \gamma(\epsilon(n_2))$. □

D'après cette proposition, pour toute exécution de DPLL sur une formule insatisfaisable, il existe une réfutation arborescente par résolution dont la taille est linéairement proportionnelle au nombre d'étapes élémentaires de l'algorithme. D'autre part, nous avons vu dans la sous-section 2.3.3 qu'il existe des familles de formules insatisfaisables pour lesquelles la taille de la plus petite réfutation par résolution croît exponentiellement selon la taille des formules. Par conséquent, le temps d'exécution au meilleur des cas de DPLL sur ces familles croît également exponentiellement. DPLL a donc une complexité temporelle exponentielle au pire des cas.

En revanche, pour une formule à n variables, DPLL ne nécessite qu'un espace mémoire linéaire selon n. En effet, si l'on considère sa formulation itérative, toutes les informations mémorisées sont de taille proportionnelle à n : les domaines des fonctions σ et λ sont les variables présentement instanciées, et donc de taille au plus n ; l'ensemble Λ contient au plus $n+1$ niveaux, et est le domaine des fonctions δ et *DécisionInversée*. Cette faible complexité spatiale est son principal avantage sur l'algorithme DP.

2.4.3 Choix des décisions

L'algorithme DPLL comporte un degré de liberté conséquent : si une décision doit être prise, elle peut concerner n'importe quelle variable pas encore instanciée. La polarité affectée à cette variable est également arbitraire. Il a été rapidement remarqué que le choix des décisions peut avoir un impact très important sur l'exécution de l'algorithme et sur sa durée. Par conséquent, de nombreux efforts ont été consacrés à la conception d'heuristiques de décision dans le but d'obtenir des exécutions de DPLL les plus courtes possibles Franco et Martin (section 17 de 2009); Marques-Silva (section 17 de 1999, section 4). Certaines de ces stratégies sont très locales et tentent de diriger la recherche le plus vite possible vers un modèle de la formule en évitant de rencontrer des conflits ; par exemple, l'heuristique Jeroslow-Wang (Jeroslow et Wang, 1990) sélectionne le littéral à décider selon le nombre de nouvelles clauses qu'il peut satisfaire (ce nombre est pondéré de façon à favoriser la satisfaction des clauses comportant le moins de littéraux indéfinis). D'autres heuristiques prennent également en considération la « qualité » du littéral opposé afin d'améliorer également la recherche en cas d'inversion de la décision ; c'est le cas de l'heuristique Böhm (Buro et Kleine Büning, 1993).

Cependant, ces heuristiques de décision présentent l'inconvénient d'introduire des calculs et comptages supplémentaires qui risquent d'alourdir l'algorithme. La plupart d'entre elles effectuent des mesures et des calculs avant chaque décision, qui nécessitent le plus souvent de parcourir intégralement toutes les clauses et leurs littéraux.

2.4.4 Occurrences multiples de conflits

Un des inconvénients majeurs de DPLL est qu'en cas de conflit, il effectue un élagage minimal de l'espace de recherche en inversant simplement la décision la plus récente possible. Cet élagage ne tient aucun compte du conflit rencontré et, par conséquent, n'empêche pas de rencontrer par la suite à nouveau exactement le même conflit.

La figure 2.1 illustre ce comportement sur un exemple d'exécution de DPLL.

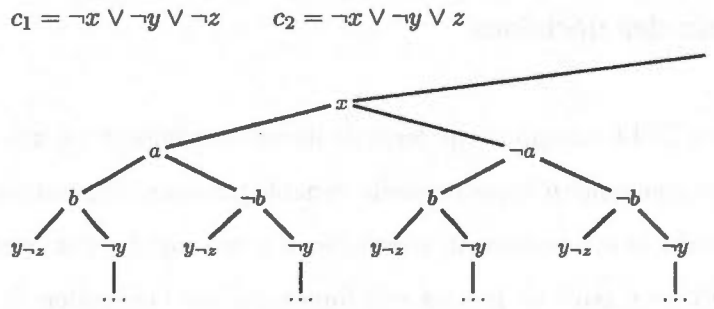


FIGURE 2.1: Exemple d'occurrences multiples d'un conflit lors d'une recherche DPLL. La figure décrit une partie de l'arbre de recherche de DPLL sur la formule $\mathcal{F}(\{a, b, x, y, z\}, \{c_1, c_2\})$ avec une heuristique de décision qui décide en priorité les variables x, a, b et y dans cet ordre en leur affectant la valeur vrai.

Supposons que la formule testée contient, entre autres, les clauses $c_1 = \neg x \vee \neg y \vee \neg z$ et $c_2 = \neg x \vee \neg y \vee z$. Supposons encore que DPLL effectue successivement les décisions x, a, b puis y. Cette dernière décision rend c_1 et c_2 unitaires. Supposons que c_1 est propagée la première : $\neg z$ est instanciée comme propagation au niveau de décision de y, ce qui rend la clause c_2 fausse : ce conflit provoque l'inversion de la dernière décision y. Supposons maintenant que la branche de recherche de la décision inversée $\neg y$ échoue : c'est maintenant l'avant-dernière décision b qui est inversée. Il est alors possible que DPLL choisisse à nouveau y, ce qui provoquera le même conflit que précédemment. Au cours de l'exécution, ce conflit peut survenir pour chaque combinaison d'instanciations des variables a et b, soit 4 fois. Plus généralement, pour n variables intercalées entre x et y, le conflit peut se répéter 2^n fois.

La stratégie d'élagage de DPLL revient à supposer que lorsqu'un conflit survient, il est dû à la fois au niveau de décision courant et à la combinaison de tous les niveaux de décision précédents. Or, notre exemple démontre que seul un sous-ensemble de ces niveaux de décision précédents peut être impliqué dans le conflit. Modifier un des niveaux de décision non-impliqués n'empêche donc en rien le conflit de survenir à nouveau.

Le mauvais élagage de l'espace de recherche et l'occurrence multiple de conflits qu'il entraîne est donc un inconvénient majeur de DPLL, qui a été en grande partie corrigé par CDCL.

2.5 L'algorithme CDCL

L'algorithme *Conflict-Driven Clause Learning* (CDCL) (Marques-Silva et Sakallah, 1999; Bayardo Jr. et Schrag, 1997) est une évolution du DPLL qui améliore son efficacité en évitant les occurrences multiples d'un même conflit. Pour cela, chaque conflit est analysé afin de dériver une **clause de conflit** qui est une conséquence logique de la formule testée et explique le conflit. Le retour arrière simple du DPLL est remplacé par un **saut arrière** (*conflict-directed backtracking*) qui peut défaire plusieurs niveaux de décision à la fois afin de retourner à la cause du conflit et de la corriger. Par conséquent, ce conflit ne pourra plus être rencontré tant qu'un saut arrière plus profond n'a pas lieu. De plus, la clause de conflit est apprise, ce qui permet de ne plus rencontrer ce conflit tant que cette clause est conservée.

La section 2.5.1 décrit globalement l'algorithme CDCL. Les sections 2.5.2 et 2.5.3 présentent respectivement le mécanisme des littéraux surveillés, qui permet de détecter efficacement les clauses unitaires et fausses, et une optimisation supplémentaire de ce mécanisme, les littéraux bloqués. Les sections 2.5.4, 2.5.5 et 2.5.6 abordent plus en détail divers aspects du CDCL : les heuristiques de décision, les clauses de conflits et l'apprentissage, et les redémarrages. La section 2.5.7 présente la variante du CDCL utilisée dans le solveur GRASP (Marques-Silva et Sakallah, 1999). La section 2.5.8 démontre les propriétés de CDCL et de la variante de GRASP. Enfin, la section 2.5.9 présente un inconvénient du CDCL : la destructivité des sauts arrière.

2.5.1 Description algorithmique de CDCL

Comme le DPLL, le CDCL combine la recherche en profondeur (les décisions) avec de l'inférence logique (les propagations unitaires). Il est décrit par l'algorithme 2.8, qui est très proche de la description itérative du CDCL (algorithme 2.4). Les deux algorithmes, tant qu'ils ne rencontrent pas de conflits, propagent les clauses unitaires

Algorithme 2.8 CDCL

1: $\Lambda \leftarrow \{0\}$
2: $\lambda_c \leftarrow 0$ /*on commence au pseudo-niveau de décision 0*/
3: $\sigma \leftarrow \emptyset$ /*on commence avec l'affectation vide*/
4: **boucle**
5: $c \leftarrow$ PROPAGER() /*propager les clauses unitaires*/
6: **si** $c \neq$ NUL **alors** /*un conflit a été découvert*/
7: **si** $\lambda_c = 0$ **alors** /*conflit au niveau de décision 0*/
8: **retourner** faux /*$\mathcal{F}$ est insatisfaisable*/
9: **sinon**
10: $\gamma \leftarrow$ ANALYSER(c) /*déduire la clause de conflit γ*/
11: $a \leftarrow l \in \gamma \mid \lambda(a) = \lambda_c$
12: /*a est unique ; c'est la future assertion*/
13: $\lambda_a \leftarrow$ NIVEAUASSERTION(γ)
14: /*calcul du niveau d'assertion*/
15: SAUTARRIÈRE(λ_a) /*retour au niveau d'assertion*/
16: $\lambda_c \leftarrow \lambda_a$ /*λ_a devient le niveau courant*/
17: $\mathcal{C} \leftarrow \mathcal{C} \cup \{\gamma\}$ /*γ est apprise*/
18: PROPAGERASSERTION(a, γ)
19: **sinon** /*aucun conflit pendant les propagations*/
20: **si** $\mathcal{D}(\sigma) = \mathcal{V}$ **alors** /*toutes les variables sont instanciées*/
21: **retourner** σ /* σ est un modèle de $\mathcal{F}$ */
22: **sinon**
23: $\lambda_c \leftarrow$ NOUVEAUNIVEAU()
24: $\Lambda \leftarrow \Lambda \cup \{\lambda_c\}$
25: **choisir** $v \in \mathcal{V} \backslash \mathcal{D}(\sigma)$ /*v est une variable non-instanciée*/
26: **choisir** $l \in \{v, \neg v\}$
27: INSTANCIER(l, NUL)
28: /*l est une décision, il n'a pas d'antécédent*/

Algorithme 2.9 ANALYSER(c) *[CDCL]*

1: /*c est la clause fausse détectée pendant les propagations unitaires*/
2: /*γ est la clause de conflit générée par l'analyse*/
3: $\gamma \leftarrow c$
4: **tant que** $|\{l \in \gamma \mid \lambda(l) = \lambda_c\}| > 1$ **faire**
5: /*il reste plus d'un littéral de niveau λ_c dans γ*/
6: $l \leftarrow$ DERNIER(γ, λ_c)
7: /*l est le littéral de niveau λ_c dans γ instancié le plus récemment*/
8: $\gamma \leftarrow \gamma \otimes_{\nu(l)} \alpha(\neg l)$ /*résolution de γ et $\alpha(\neg l)$ sur la variable de l*/
9: **retourner** γ

en priorité, ou prennent une décision s'il ne reste aucune clause unitaire.[8] Dans notre description du CDCL, les étapes de propagations et de recherche de clauses fausses sont encapsulées dans la procédure PROPAGER (détaillée dans la sous-section 2.5.2). Celle-ci effectue des propagations de clauses unitaires jusqu'à ce qu'un conflit soit atteint ou qu'il ne reste aucune clause unitaire.

La différence principale de CDCL avec DPLL se situe dans la gestion des conflits ; celle-ci nécessite de maintenir une fonction partielle supplémentaire $\alpha : \mathcal{V} \nrightarrow \mathcal{C}$, qui à toute variable instanciée par propagation unitaire associe son **antécédent**, c'est-à-dire la clause unitaire qui a causé cette propagation. Pour un littéral $l \in \mathcal{L}$, nous utiliserons l'abus de notation $\alpha(l) = \alpha(\nu(l))$. Pour tout littéral propagé $l \in \sigma$, $l \in \alpha(l)$. Par abus de langage, nous appellerons antécédents (au pluriel) de l tous les littéraux de $\alpha(l)$ différents de l.

Lorsqu'une clause fausse c est découverte, elle est analysée afin d'inférer une nouvelle clause qui explique le conflit et détermine à partir de quel niveau de décision il aurait pu être évité. L'analyse, détaillée par l'algorithme 2.9, produit une clause de conflit γ, conséquence logique du problème d'origine, qui est comme c fausse sous l'instanciation courante, mais avec uniquement un seul littéral instancié au niveau courant λ_c (c a forcément au moins deux littéraux instanciés au niveau λ_c ; dans le cas contraire, cela signifierait qu'une propagation unitaire a été omise avant la dernière décision).

8. La règle de littéral pur n'est quasiment jamais implémentée dans les solveurs CDCL, nous l'avons donc omise dans sa description.

Algorithme 2.10 NIVEAUASSERTION(γ) *[CDCL]*

1: $\Lambda_\gamma \leftarrow \{\lambda(l) \mid l \in \gamma\} \setminus \{\lambda_c\}$
2: **si** $\Lambda_\gamma = \emptyset$ **alors** /*$|\gamma| = 1$*/
3: **retourner** 0
4: **sinon**
5: **retourner** $\max(\Lambda_\gamma)$

La clause γ est tout d'abord initialisée à c. Tant que γ contient au moins deux littéraux du niveau courant, ANALYSER sélectionne le dernier de ces littéraux à avoir été instancié. Soit l ce littéral. $\neg l$ ne peut pas être la décision du niveau λ_c, c'est donc forcément une propagation ; par conséquent, $\alpha(\neg l)$ est défini. $l \in \gamma$ et $\neg l \in \alpha(\neg l)$, donc ces deux clauses peuvent être résolues sur $\nu(l)$ et γ est remplacée par le résultat de cette résolution. La nouvelle clause γ ne contient que des variables instanciées strictement avant $\neg l$. De plus, γ est une conséquence logique de la formule testée, puisqu'elle le résultat de la résolution de $c \in \mathcal{C}$ avec les antécédents de divers littéraux, qui sont eux-mêmes des clauses de $\mathcal{C}$ ou d'autres clauses de conflits précédemment apprises. L'opération est répétée jusqu'à ce qu'il ne reste plus qu'un seul littéral du niveau courant dans γ. Cette suite de résolutions termine toujours, puisqu'à chaque résolution on élimine un littéral du niveau courant qui ne sera jamais réintroduit, en raison de l'ordre dans lesquels les littéraux sont sélectionnés.

L'unique littéral de niveau λ_c dans γ s'appelle l'**assertion,** notée a. γ devient unitaire lorsque ce niveau est défait. La clause de conflit indique en fait une propagation unitaire qui aurait permis d'éviter le conflit si elle avait été connue précédemment. De plus, l'assertion aurait pu être propagée dès le **niveau d'assertion** λ_a, défini comme le deuxième plus grand niveau de décision présent dans γ, ou comme le pseudo-niveau de décision 0 si a est le seul littéral de γ (algorithme 2.10). Par conséquent, le CDCL effectue alors un SAUTARRIÈRE (algorithme 2.11) pour retourner directement à ce ni-

Algorithme 2.11 SAUTARRIÈRE(λ_a) *[CDCL]*

1: **pour** $v \in \mathcal{D}(\sigma) \mid \lambda(v) > \lambda_a$ **faire**
2: DÉFAIRE(v)
3: $\Lambda \leftarrow \{i \in \Lambda \mid i \leq \lambda_a\}$

Algorithme 2.12 PROPAGERASSERTION(a, γ) *[CDCL]*

1: INSTANCIER(a, γ)

veau d'assertion et y propager la clause de conflit (algorithme 2.12). Tous les niveaux de décision supérieurs au niveau d'assertion sont entièrement désinstanciés par le saut arrière. Comme a est instancié au niveau d'assertion, il sera dans la suite de la recherche impossible de rencontrer à nouveau le même conflit tant que le niveau d'assertion n'est pas défait. La clause de conflit est également apprise par l'algorithme (elle est ajoutée à C), ce qui permet de continuer à éviter le conflit même après la désinstanciation du niveau d'assertion.

Dans l'algorithme CDCL, une décision ne peut pas être inversée comme dans le DPLL, elle ne peut être que défaite. À tout moment, les décisions dans un algorithme CDCL représentent réellement des choix arbitraires, alors qu'au cours d'une exécution de DPLL les décisions inversées sont en fait des instanciations inférées par l'échec de branches antérieures de la recherche. Il est possible qu'au cours d'une résolution de conflit dans CDCL l'assertion a corresponde à l'opposé de l'ancienne décision du niveau de conflit [9], mais celui-ci devient alors une propagation à un niveau de décision différent.

DPLL et CDCL détectent tous deux l'insatisfaisabilité d'une formule lorsqu'un conflit survient et qu'aucune décision ne peut être modifiée. Dans DPLL, c'est le cas lorsque toutes les décisions ont déjà été inversées ; dans CDCL, ce cas survient lorsqu'il n'y a aucune décision et que le conflit est donc déclenché au niveau de décision 0.

9. Nous appellerons **niveau de conflit** le plus grand niveau de décision présent dans la clause qui a déclenché le conflit. Dans le cas du CDCL ordinaire, il s'agit toujours du niveau courant λ_c, mais ça ne sera pas le cas pour les algorithmes de CDCL sans saut arrière décrits dans le chapitre 6 .

Algorithme 2.13 INSTANCIER(l, c) *[CDCL]*

Requis : $\nu(l) \notin \mathcal{D}(\sigma)$

1: $v \leftarrow \nu(l)$
2: $\sigma(v) \leftarrow \rho(l)$
3: $\lambda(v) \leftarrow \lambda_c$
4: $\alpha(v) \leftarrow c$

Algorithme 2.14 DÉFAIRE(v) *[CDCL]*

1: $\sigma(v) \leftarrow$ indéfini
2: $\lambda(v) \leftarrow$ indéfini
3: $\alpha(v) \leftarrow$ indéfini
4: *propagé*$(v) \leftarrow$ faux

Au contraire, la satisfaisabilité est prouvée lorsque la procédure PROPAGER termine sans conflit et que toutes les variables de la formule sont instanciées. Cela signifie que l'instanciation courante est complète et satisfait toutes les clauses de la formule ; elle est donc un modèle.

2.5.2 Propagations unitaires par littéraux surveillés

Dans les premiers solveurs CDCL, comme GRASP (Marques-Silva et Sakallah, 1999), les propagations unitaires sont détectées de la même façon que dans la plupart des solveurs DPLL, c'est-à-dire en utilisant des listes d'occurrences des littéraux. Pour chaque littéral du problème, une liste de toutes les clauses dans lesquelles ce littéral apparaît est maintenue. Lorsqu'un littéral l est instancié, l'algorithme parcourt alors la listes des clauses contenant $\neg l$ pour vérifier si certaines deviennent unitaires, en usant de diverses heuristiques. Par exemple, GRASP maintient pour chaque clause deux compteurs de phases, indiquant respectivement le nombre de littéraux positifs et négatifs dans cette clause. Une clause comportant n littéraux est ainsi détectée comme unitaire lorsque $n - 1$ de ses littéraux sont négatifs et aucun n'est positif, ce qui permet de vérifier son statut sans avoir à parcourir ses littéraux. Cette stratégie implique cependant de mettre à jour les compteurs de toutes les clauses contenant l ou $\neg l$ lors de l'instanciation ou la désinstanciation d'un littéral $l \in \mathcal{L}$.

Le mécanisme de littéraux surveillés (Moskewicz et al., 2001) se base sur le constat qu'il n'est pas nécessaire de vérifier le statut d'une clause à chaque instanciation de l'opposé d'un de ses littéraux. En effet, si deux littéraux indéfinis ou vrais sont choisis arbitrairement, la clause ne peut devenir unitaire tant qu'aucun de ces deux littéraux ne devient faux. La stratégie des littéraux surveillés consiste donc à maintenir une telle liste

Algorithme 2.15 PROPAGER() *[CDCL]*

```
1: Π ← {l ∈ σ | propagé(l) = faux}
2: tant que Π ≠ ∅ faire
3:     choisir l ∈ Π
4:     pour c ∈ C | ¬l ∈ ω(c) faire /*¬l est surveillé dans c*/
5:         w ← ω(c) \ {¬l}  /*w est le second littéral surveillé dans c*/
6:         si σ(w) ≠ vrai alors
7:             Ω ← {l' ∈ c \ {w} | σ(l') ≠ faux}
8:             /*Ω est l'ensemble des littéraux pouvant remplacer ¬l*/
9:             si Ω ≠ ∅ alors
10:                choisir w'∈Ω
11:                ω(c) ← {w, w'} /*w' est surveillé à la place de ¬l*/
12:            sinon /* tous les autres littéraux de c sont faux*/
13:                si σ(w) = indéfini alors /*c est unitaire*/
14:                    INSTANCIER(w, c)  /*w est propagé par c*/
15:                    Π ← Π ∪ {w} /*w doit être lui-même propagé*/
16:                sinon
17:                    retourner c /*c est un conflit*/
18:    Π ← Π \ {l}
19:    propagé(l) ← vrai
20: retourner NUL  /*aucun conflit rencontré*/
```

de deux littéraux pour chaque clause. Lorsqu'un littéral l est instancié, on ne vérifie pas toutes les clauses où $\neg l$ apparaît, mais uniquement celles où il est surveillé, ce qui réduit considérablement le temps d'exécution en pratique. De plus, les auteurs de ce mécanisme avaient identifié les propagations unitaires comme la tâche de loin la plus coûteuse en temps de calcul dans un solveur CDCL. L'économie permise par les littéraux surveillés est donc d'autant plus cruciale ; ce mécanisme est par conséquent utilisé par tous les solveurs CDCL contemporains.

La détection des propagations unitaires à l'aide des littéraux surveillés est décrite par l'algorithme PROPAGER (2.15). Pour chaque clause $c \in \mathcal{C}$, on note sa paire de littéraux surveillés par $\omega(c) = \{w_1, w_2\} \subseteq c$[10]. Au début de l'exécution, $\omega(c)$ est une paire quelconque de littéraux de c.

10. On peut considérer que chaque clause contient au moins deux littéraux car toute clause unaire $\{l\}$ n'est pas représentée en tant que telle mais comme une propagation unitaire de l au pseudo-niveau 0.

Dans le contexte des propagations unitaires par littéraux surveillés, nous dirons qu'une instanciation $l \in \sigma$ a déjà été propagée si toutes les clauses où $\neg l$ est surveillé ont déjà été parcourues et traitées si nécessaire. La fonction partielle *propagé* : $\mathcal{V} \nrightarrow \mathcal{B}$ est utilisée pour garder la trace des instanciations qui ont déjà été propagées.

L'algorithme PROPAGER traite l'ensemble des instanciations pas encore propagées, jusqu'à ce qu'il n'en reste plus. Pour chaque littéral l, il examine toutes les clauses c dans lesquelles $\neg l$ est surveillé. Le remplacement du littéral surveillé $\neg l$ dans c dépend du statut du second littéral surveillé $w = \omega(c) \setminus \{\neg l\}$. Si w est vrai, il n'est pas nécessaire de remplacer $\neg l$. En effet, comme w est instancié au moment où $\neg l$ est propagé, $\lambda(w) \leq \lambda(l)$. Comme les niveaux de décision sont défaits dans l'ordre inverse de leur création, $\neg l$ sera forcément désinstancié avant w. Lorsque w sera désinstancié, $\neg l$ le sera donc également, et la clause sera toujours correctement surveillée. Il n'est donc pas nécessaire de remplacer un littéral surveillé faux si le second littéral surveillé est vrai.

Si w n'est pas vrai, il est alors nécessaire de remplacer $\neg l$ par un autre littéral non faux de c distinct de w (lignes 7 à 11). Si un tel littéral n'existe pas et si w est indéfini, alors c est unitaire et w est instancié par propagation de c (lignes 13 à 15). Si w est faux, alors la clause c entière est fausse. L'étape de propagation est interrompue et c déclenche un conflit (ligne 17).

Notons que la méthode des littéraux surveillés a été inspirée par la technique utilisée dans SATO (Zhang et Stickel, 2000), qui surveille également deux littéraux dans chaque clause. Dans le cas de SATO, les littéraux de chaque clause sont ordonnés, et les deux littéraux surveillés sont obligatoirement la tête et la queue de la clause, c'est-à-dire respectivement le premier et le dernier littéral indéfinis. Comme pour les littéraux surveillés, lorsqu'un littéral l est instancié, il est uniquement nécessaire de vérifier les clauses dont $\neg l$ est la tête ou la queue. Cependant, si l est désinstancié ultérieurement, il doit possiblement redevenir la tête ou la queue de certaines clauses, ce qui demande un travail supplémentaire. En supprimant les contraintes d'ordre entre littéraux, les littéraux surveillés ont l'avantage de ne nécessiter aucune modification lors

de désinstanciations de variables.

2.5.3 Littéraux bloqués

L'utilisation de littéraux bloqués (Sörensson et Eén, 2009) est une optimisation supplémentaire pour améliorer l'efficacité des propagations unitaires. Il s'agit plus précisément d'une minimisation de l'accès à la mémoire vive. Les implémentations ordinaires gèrent les littéraux surveillés en listant pour chaque littéral les clauses dans lesquelles il est surveillé. Lors de la vérification d'une clause, l'algorithme commence par consulter la valeur du second littéral surveillé afin de savoir si le remplacement du littéral surveillé faux peut être évité. L'accès à ce second littéral se fait indirectement, en passant par un premier accès mémoire à la clause.

La stratégie des littéraux bloqués, décrite par l'algorithme 2.16, consiste à mémoriser pour chaque littéral non seulement la liste des clauses où il est surveillé mais aussi, pour chacune de ces clauses, un de ses littéraux, qui pourra donc être accédé directement. On note $\beta(c, l)$ le littéral bloqué associé à la clause c pour le littéral surveillé l. $\beta(c, l)$ n'est pas forcément identique au second littéral surveillé ; il peut être n'importe quel littéral de c, bien qu'il soit inutile s'il est identique à l. Heuristiquement, on choisit en général comme littéral bloqué le dernier littéral vrai rencontré lors de la vérification de la clause c par propagation du littéral $\neg l$ (ligne 8). Lorsqu'une clause doit être vérifiée, on regarde d'abord la valeur de son littéral bloqué (ligne 5). S'il est vrai, on a prouvé que la clause n'est pas unitaire avec un seul accès mémoire et on ne modifie aucun littéral surveillé. Sinon, on effectue l'algorithme de littéraux surveillés classique sur la clause. Lorsqu'un littéral surveillé est remplacé, le nouveau littéral surveillant conserve le même littéral bloqué que l'ancien (ligne 16).

Les littéraux bloqués assouplissent grandement la façon dont les clauses sont surveillées. On peut considérer qu'un littéral bloqué vrai $\beta(c, l)$ surveille c à la place de l. Par exemple, si l devient faux et que $\beta(c, l)$ est vrai, l ne sera pas remplacé, même si le second littéral surveillé dans c n'est pas vrai. En pratique, il est assez fréquent que les lit-

Algorithme 2.16 PROPAGER() *[$CDCL_{LB}$]*

```
1: Π ← {l ∈ σ | propagé(l) = faux}
2: tant que Π ≠ ∅ faire
3:     choisir l ∈ Π
4:     pour c ∈ C | ¬l ∈ ω(c) faire /*¬l est surveillé dans c*/
5:         si σ(β(c,¬l)) = faux alors /*rien à modifier si le littéral bloqué est vrai*/
6:             w ← ω(c)\{¬l} /*w est le second littéral surveillé dans c*/
7:             si σ(w) = vrai alors
8:                 β(c,¬l) ← w /*on bloque w pour ¬l*/
9:             sinon
10:                Ω ← {l' ∈ c \ {w} | σ(l') ≠ faux}
11:                /*Ω est l'ensemble des littéraux pouvant remplacer ¬l*/
12:                si Ω ≠ ∅ alors
13:                    choisir w'∈Ω
14:                    ω(c) ← {w, w'}
15:                    /*w' est surveillé à la place de ¬l*/
16:                    β(c, w') ← β(c,¬l)
17:                    /*transmission du littéral bloqué*/
18:                sinon /*tous les autres littéraux de c sont faux*/
19:                    si σ(w) = indéfini alors /*c est unitaire*/
20:                        INSTANCIER(w, c) /*w est propagé par c*/
21:                        Π ← Π∪{w} /*w doit être lui-même propagé*/
22:                    sinon
23:                        retourner c /*c est un conflit*/
24:    Π ← Π \ {l}
25:    propagé(l) ← vrai
26: retourner NUL /*aucun conflit rencontré*/
```

téraux bloqués évitent d'accéder au second littéral surveillé, ce qui peut significativement diminuer le temps d'exécution des solveurs.

Nous nommerons $CDCL_{LB}$ la variante de l'algorithme CDCL utilisant les littéraux bloqués. La procédure PROPAGER est la seule différence entre ces deux variantes.

2.5.4 Heuristiques de décision

Lorsqu'une décision doit être prise dans CDCL, comme dans DPLL, le choix du littéral à instancier est totalement libre et peut être pris à l'aide de différentes heuristiques. Toutefois, si la plupart des heuristiques utilisées dans les solveurs DPLL reposent

sur des statistiques à propos des différentes variables et clauses au moment de la décision, la quasi-totalité des heuristiques utilisées dans les solveurs CDCL depuis CHAFF (Moskewicz et al., 2001) base ses choix sur les **activités** des variables : l'algorithme choisit en priorité des variables qui ont été impliquées souvent et récemment dans les analyses de conflits.

La stratégie originale de CHAFF, VSIDS (*Variable State Independent Decaying Sum*), maintient pour chaque littéral un compteur, qui est incrémenté lorsque le littéral apparaît dans une clause de conflit. Lors d'une décision, le littéral non-assigné dont le compteur est le plus élevé est choisi. De plus, l'algorithme décrémente périodiquement l'ensemble des compteurs afin de favoriser les variables dont l'activité est récente. Les solveurs CDCL ultérieurs utilisent des variantes plus ou moins proches de VSIDS. MINISAT (Eén et Sörensson, 2004), par exemple, utilise une stratégie quasiment identique, à la différence qu'il maintient un compteur d'activité pour chaque variable et non pas pour chaque littéral. BERKMIN maintient également un compteur par variable, mais il incrémente le compteur d'une variable pour chaque occurrence dans une clause parcourue ou dérivée lors de l'analyse de conflit, alors que CHAFF et MINISAT considèrent uniquement les occurrences de variables dans la clause de conflit finale. De plus, pour encore renforcer l'impact des derniers conflits sur les décisions, BERKMIN restreint le choix aux variables de la clause de conflit la plus récente qui n'est pas satisfaite.

L'intuition derrière les heuristiques par activité est que les variables les plus impliquées récemment dans des conflits sont plus pertinentes dans la partie actuelle de l'espace de recherche, car elles permettraient a priori de rencontrer de nouveaux conflits plus rapidement et donc d'accélérer l'élagage de l'espace de recherche. On peut constater qu'il s'agit d'une stratégie totalement opposée à celle des heuristiques que nous avons précédemment mentionnées dans la sous-section 2.4.3 dans le cadre du DPLL, qui, au contraire, cherchaient à compléter l'instanciation courante en évitant les conflits autant que possible.

Toutefois, l'objectif premier de ces heuristiques de décision à partir de l'activité

des variables est d'éviter le surcoût de calculs des heuristiques classiques au moment de la décision. En effet, la gestion des activités nécessite uniquement de mettre à jour les compteurs de quelques littéraux lors de chaque conflit, et si nécessaire de réordonner les littéraux indéfinis selon leur activité. Les heuristiques de choix à base d'activité de variables combinent donc les avantages d'une bonne efficacité et d'un très faible surcoût de calcul. Les résultats expérimentaux semblent effectivement confirmer l'efficacité de ces heuristiques (Moskewicz et al., 2001).

Notons que dans la plupart des solveurs CDCL, les variables de décisions sont de temps à autre choisies aléatoirement, dans le but de varier un peu plus les différentes exécutions possibles sur un problème donné, et ainsi d'accroître l'utilité des redémarrages (voir sous-section 2.5.6).

2.5.5 Clauses de conflits et apprentissage

Il existe de nombreuses tactiques de dérivation de clauses de conflits. Celle que nous avons détaillée dans l'algorithme 2.9 est dite du **premier point d'implication unique**. Il s'agit de la tactique la plus couramment employée ; elle est notamment utilisée dans CHAFF (Moskewicz et al., 2001), MINISAT (Eén et Sörensson, 2004) et GLUCOSE (Audemard et Simon, 2009).

Un point d'implication unique est un littéral du niveau courant qui, rajouté aux instanciations des niveaux précédents, suffit à provoquer le conflit analysé. L'algorithme 2.9 interrompt des résolutions successives dès qu'il obtient une clause contenant un seul littéral de niveau de décision courant, il s'agit donc du premier point d'implication unique (dans l'ordre des résolutions à partir de la clause fausse, c'est-à-dire dans l'ordre inverse de l'instanciation des littéraux). L'intérêt d'une clause de conflit contenant un point d'implication unique (ou PIU) est qu'elle provoque une propagation unitaire immédiatement après le saut arrière, ce qui contribue à l'élagage de l'espace de recherche.

Il est cependant possible de choisir différemment le point d'implication unique ;

par exemple, RELSAT (Bayardo Jr. et Schrag, 1997) utilise le dernier PIU, c'est-à-dire la décision du niveau courant, et continue donc les résolutions jusqu'à ce que cette décision soit le seul littéral de niveau courant. Le choix du premier PIU a cependant de nombreux avantages sur tous les autres PIU : non seulement son calcul est plus rapide, puisqu'il requiert le moins de résolutions successives, mais de plus les résolutions supplémentaires nécessitées par les autres PIU peuvent alourdir la clause en rajoutant de nouveaux littéraux de niveaux de décision inférieurs. Ces nouveaux littéraux peuvent même augmenter le niveau d'assertion et par conséquent affaiblir l'élagage de l'espace de recherche lors du saut arrière Marques-Silva, Lynce et Malik (2009).

D'autres tactiques existent, dont certaines effectuent également des résolutions sur des littéraux de niveaux inférieurs, dans le but par exemple d'obtenir une clause de conflit avec un seul littéral par niveau de conflit. Les solveurs utilisant la stratégie de premier PIU semblent cependant en pratique obtenir de meilleurs résultats (Zhang et al., 2001), certainement en partie grâce à leur rapidité de dérivation des clauses de conflit.

Enfin, GRASP (Marques-Silva et Sakallah, 1999) a la particularité d'apprendre plusieurs clauses lors de certains conflits, en plus de la clause de conflit. Cela lui permet d'apprendre plus d'informations de chaque conflit, et donc d'améliorer par la suite l'élagage de l'espace de recherche ; cependant, si cette tactique est utilisée en conjonction avec les littéraux surveillées, ce plus grand nombre de clauses apprises a l'inconvénient de ralentir significativement les propagations unitaires (Zhang et al., 2001).

L'inconvénient principal de l'apprentissage des clauses est qu'il peut causer une explosion de l'espace mémoire nécessaire, puisque chaque conflit produit une clause inédite. De plus, comme nous l'avons évoqué plus haut, l'augmentation du nombre de clauses provoque un ralentissement de la détection des propagations unitaires par les littéraux surveillés.

L'apprentissage n'est cependant pas nécessaire à la correction de l'algorithme, et toutes les clauses apprises peuvent être effacées à tout moment (sauf lorsqu'elles sont

utilisées comme antécédent d'une instanciation). En pratique, les solveurs CDCL utilisent donc différentes heuristiques d'effacement de clauses apprises, en recherchant un compromis entre la conservation de connaissances et l'encombrement de la mémoire et des propagations. Par exemple, CHAFF évalue la pertinence des clauses apprises par leur nombre de littéraux non-assignés et les efface lorsque ce nombre dépasse un seuil prédéfini. MINISAT gère une activité des clauses apprises similaire à l'activité des variables : l'activité d'une clause augmente lorsqu'elle intervient dans une analyse de conflit. Les clauses trop inactives sont périodiquement effacées. GLUCOSE se base sur le nombre de niveaux de décision distincts dans les clauses au moment de leur apprentissage pour estimer leur utilité future. La précision de cette estimation permet à GLUCOSE d'effacer souvent de nombreuses clauses, et donc d'alléger le mécanisme de propagation unitaire, sans impact négatif sur l'élagage de l'espace de recherche.

2.5.6 Redémarrages

Un redémarrage au cours d'une exécution de CDCL consiste à interrompre la recherche courante, c'est-à-dire à défaire entièrement l'instanciation courante, et à en recommencer une nouvelle. Le but original des redémarrages est de contrer le phénomène de « queue lourde » (*heavy-tail*) constaté dans les solveurs SAT (Gomes et al., 2000) : pour un algorithme CDCL à l'heuristique de décision partiellement aléatoire, il est très fréquent que parmi les différentes exécutions possibles sur une instance donnée, quelques-unes soient bien plus longues que la moyenne, alors qu'un nombre relativement élevé d'exécutions sont au contraire relativement courtes. Introduire la possibilité de redémarrages au cours d'une exécution de CDCL permet donc d'échapper à ces exécutions de la queue lourde si elles sont rencontrées.

Des redémarrages trop systématiques (par exemple au bout d'un temps fixé ou d'un nombre fixé de décisions ou de conflits) peuvent évidemment compromettre la complétude de l'algorithme. Toutefois, il suffit d'élargir progressivement cette borne afin de garantir la complétude (Baptista et Marques-Silva, 2000). De plus, il a été prouvé que l'efficacité des redémarrages peut être améliorée en tenant compte de critères tels que la

longueur des clauses apprises (Pipatsrisawat et Darwiche, 2009) : si la recherche conduit à apprendre une grande quantité de clauses au-dessus d'une longueur limite, l'exécution courante est considérée de mauvaise qualité et un redémarrage intervient.

Lors d'un redémarrage dans CDCL, il est possible de conserver les instanciations du niveau 0 (car aucun modèle ne peut contenir leur opposé), ainsi que les clauses précédemment apprises. Ces éléments conservés permettent un meilleur élagage après redémarrage que pour l'exécution initiale. De plus, les activités des variables peuvent également être conservées. Le redémarrage permet donc aussi de revenir sur des décisions qui se sont révélées peu pertinentes ; la nouvelle exécution peut être plus efficace en effectuant des décisions sur des variables à plus haute activité (Huang, 2007). Cette dernière observation plaide pour une utilisation fréquente des redémarrages, dont l'efficacité s'observe en pratique (Haim et Heule, 2010).

Avec des redémarrages très fréquents, les activités des variables ont tendance à moins varier ; par conséquent, il est fort probable que la nouvelle décision prenne initialement les mêmes décisions dans le même ordre. L'utilisation des redémarrages a donc évolué avec le temps : à l'origine conçus pour interrompre une exécution peu prometteuse et entamer une nouvelle exécution potentiellement très différente, l'augmentation de leur fréquence les a fait évoluer vers un moyen de passer à une nouvelle exécution très proche de l'ancienne tout en évitant certains choix qui se sont avérés après coup peu pertinents (Biere, 2008).

2.5.7 Le CDCL de type GRASP

L'algorithme de GRASP présente des différences fondamentales avec le CDCL tel que présenté dans la sous-section 2.5.1 et décrit par l'algorithme 2.8. Par la suite, nous nommerons ce dernier « CDCL classique » ou « CDCL ordinaire » pour éviter toute ambiguïté avec le CDCL de type GRASP si nécessaire. La variante de GRASP n'est en pratique plus utilisée par les solveurs SAT contemporains, qui adoptent tous la démarche du CDCL ordinaire ; nous présentons toutefois son principe car nous montrerons par la

suite dans la section 5.6 qu'elle ne vérifie pas certaines des propriétés du CDCL classique.

La particularité de GRASP est qu'il utilise parfois des méthodes de résolution de conflits de type DPLL au lieu du saut arrière du CDCL. Plus exactement, GRASP conserve la notion d'inversion de décisions présente dans DPLL. Lors d'un conflit, si la décision du niveau courant n'a pas déjà été inversée auparavant, GRASP dérive une clause de conflit correspondant au premier PIU. Cependant, l'analyse n'est pas utilisée pour effectuer un saut arrière. GRASP défait simplement le niveau courant et propage la clause de conflit à ce même niveau courant. L'assertion devient la **pseudo-décision** de ce niveau, qui est donc une propagation positionnée à un niveau de décision différent de celui de tous ses antécédents. Notons que, dans le cas de conflits ne comportant qu'un seul PIU, cette stratégie correspond à inverser la décision. Pour alléger le texte, nous désignerons donc ce procédé comme une inversion de décision, bien que la pseudo-décision ne corresponde pas toujours à l'opposé de la décision défaite.

Si, lors du déclenchement du conflit, le niveau courant comporte une pseudo-décision, la clause de conflit γ est cette fois obtenue par résolution sur la clause fausse des antécédents de tous ses littéraux du niveau courant, pseudo-décision comprise. La clause de conflit obtenue ne comporte donc que des niveaux de décision strictement inférieurs au niveau courant. GRASP effectue alors un saut arrière jusqu'au niveau λ_γ, le plus grand niveau de décision dans γ. Le conflit n'est pas résolu après ce saut arrière, car il n'a défait aucun littéral de γ . L'algorithme considère alors γ comme une nouvelle clause fausse et résout ce nouveau conflit de la même façon, selon que le niveau λ_γ contient ou non une pseudo-décision.

Le mécanisme de résolution de conflits de GRASP est bien un hybride de DPLL et CDCL. DPLL résout un conflit en inversant la décision la plus récente possible. CDCL le résout en effectuant un saut arrière. GRASP inverse la décision du niveau courant si possible, sinon il effectue un saut arrière. On peut noter que GRASP effectue un élagage de l'espace de recherche moins exhaustif que CDCL. En effet, lors d'une inversion de décision, la pseudo-décision pourrait être instanciée à un niveau de décision

strictement inférieur. De plus, l'alternance d'inversions de décisions et de sauts arrière a pour conséquence d'empêcher la détection de certaines propagations unitaires dans GRASP (voir la proposition 5.10 dans le chapitre 5).

Notons enfin que, comme nous l'avons évoqué dans la sous-section 2.5.5, GRASP apprend parfois plus d'une clause par conflit. Dans le cas d'une inversion de décision, si le conflit comporte plusieurs PIU distincts, GRASP apprend en plus de la clause du premier PIU des clauses de reconvergence entre les PIU successifs. Pour chaque paire de PIU successifs p_i et p_{i+1} [11], la clause de reconvergence correspondante exprime le fait que l'instanciation de p_{i+1} et de certains littéraux des niveaux de décision antérieurs impliquent p_i. Cette clause est obtenue par résolutions successives de $\alpha(p_i)$ avec les antécédents des autres littéraux de niveau courant dans $\alpha(p_i)$ jusqu'à obtenir une clause ne contenant que deux littéraux du niveau courant, qui sont p_i et p_{i+1} (il est possible qu'aucune résolution ne soit nécessaire ; dans ce cas la clause de reconvergence est déjà connue est n'est bien sûr pas apprise). Dans le cas d'un saut arrière, en plus de la clause de conflit qui ne contient aucun littéral du niveau courant, GRASP apprend également la clause du premier PIU et toutes les clauses de reconvergence.

2.5.8 Propriétés et complexité de CDCL

Cette sous-section présente les principales propriétés du CDCL, c'est-à-dire sa correction, sa complétude, sa terminaison et sa complexité temporelle, qui sont prouvées dans la première partie. La seconde partie discute l'adaptation de ces preuves à la variante de GRASP. Nous verrons dans les chapitres 6 et 7 que ces propriétés peuvent également être prouvées pour les différentes variantes de CDCL sans saut arrière et de CDCL à ordre partiel soit à l'aide des même preuves exposées ici pour le CDCL classique, soit en les adaptant à certaines spécificités des algorithmes.

---

11. On considère ici encore les PIU dans l'ordre inverse de leur instanciation.

2.5.8.1 Propriétés du CDCL classique

Nous développons ici les preuves de correction, complétude et terminaison du CDCL classique telles que présentées par Zhang (2003). Ces preuves sont valables avec ou sans gestion des littéraux bloqués.

Proposition 2.5. *CDCL est correct.*

Démonstration. La preuve de correction de Zhang se base sur le postulat qu'au cours de l'exécution du CDCL, toutes les clauses fausses sont détectées par le mécanisme de propagation unitaire par littéraux surveillés (ce que nous prouverons formellement par le corollaire 5.6 du chapitre 5). Lorsque la propagation unitaire termine sans conflit et que toutes les variables sont instanciées, cela signifie donc que toutes les clauses sont satisfaites par l'instanciation courante, qui est donc un modèle de la formule propositionnelle. □

Proposition 2.6. *CDCL est complet.*

Démonstration. Comme pour DP et DPLL, la complétude de CDCL peut se montrer en prouvant qu'à partir de son exécution sur une formule insatisfaisable, on peut construire une réfutation par résolution de cette formule.

CDCL déclare une formule insatisfaisable lorsqu'un conflit est déclenché par une clause fausse finale c dont tous les littéraux ont été instanciés au niveau 0, et ont donc été propagés par des clauses unitaires ou par d'autres propagations au niveau 0. Par conséquent, l'antécédent de chaque littéral $l \in c$ contient uniquement des littéraux de niveau 0 instanciés avant l. On peut donc éliminer toutes les variables de c par résolutions successives avec leur antécédent, dans l'ordre inverse de leur instanciation. On aboutira finalement à une clause vide. La clause c, ainsi que tous les antécédents de littéraux utilisés pour les résolutions successives, sont soit des clauses de la formule d'origine, soit des clauses apprises au cours de l'exécution, qui ont elles-mêmes été obtenues par résolutions successives à partir des clauses d'origine et précédemment apprises. On obtient

donc bien une réfutation par résolution de la formule. □

Proposition 2.7. *CDCL termine.*

Démonstration. Zhang prouve la terminaison de CDCL en définissant une fonction qui pondère les variables instanciées selon leur niveau de décision. Cette fonction est conçue afin que le poids d'une seule variable à un niveau de décision quelconque soit strictement supérieur au poids de toutes les variables de la formule placées à des niveaux de décision strictement supérieurs. Plus précisément, cette fonction est $f = \sum_{l=0}^{n} \frac{k(l)}{(n+1)^l}$, où n est le nombre de variables de la formule et $k(l)$ est le nombre de variables instanciées au niveau de décision l. Chaque variable instanciée a donc un poids de $\frac{1}{(n+1)^l}$.

La valeur de f est modifiée lorsqu'un littéral est instancié ou lorsqu'un saut arrière est effectué. Dans le cas d'une instanciation, il est évident que f croît strictement. Dans le cas d'un saut arrière du niveau courant λ_c au niveau d'assertion λ_a, toutes les variables des niveaux $\lambda_a + 1$ à λ_c sont désinstanciées, sauf la variable d'assertion $\nu(a)$, qui passe du niveau λ_c au niveau $\lambda_a < \lambda_c$. Compte tenu des propriétés de f, le gain apporté par le changement de niveau de décision de l'assertion est strictement supérieur à la perte des variables désinstanciées. f est donc une fonction monotone strictement croissante. Or, elle ne peut prendre qu'un nombre fini de valeurs, qui est borné par le nombre de distributions des variables dans les différents niveaux de décision. Par conséquent, l'algorithme termine. □

Proposition 2.8. *La complexité temporelle de CDCL est exponentielle au pire des cas.*

Démonstration. Cette complexité temporelle se prouve à l'aide de la réfutation par résolution partiellement générée par CDCL sur une formule insatisfaisable, mise en évidence par la preuve de complétude ci-dessus. La plus grande partie de ces résolutions est réalisée explicitement par la phase d'analyse de conflit de l'algorithme. Pour obtenir la clause vide, il est toutefois nécessaire d'effectuer des résolutions supplémentaires sur la clause fausse finale. Comme les littéraux sont éliminés dans l'ordre inverse de leur instanciation, le nombre de résolutions supplémentaires nécessaires est borné par le nombre de

variables de la formule. Pour une famille de formules insatisfaisables dont la taille minimale des réfutations augmente exponentiellement selon le nombre de variables, toute exécution de CDCL contient donc un nombre de résolutions exponentiel selon le nombre de variables. Comme de telles familles de formules existent Haken (1985), cela prouve la complexité temporelle exponentielle de CDCL. □

La complexité spatiale de CDCL dépend fortement de la stratégie d'effacement des clauses apprises. Si le solveur n'efface jamais aucune clause, la complexité peut devenir exponentielle au pire des cas. En effet, si le nombre de résolutions au cours d'une exécution de CDCL est exponentiel, le nombre de conflits, et donc de clauses apprises, l'est également, puisque le nombre de résolutions par conflit est borné par le nombre de variables de la formule. À l'opposé, il est également possible d'effacer chaque clause apprise dès qu'elle n'est plus utilisée comme antécédent d'un littéral. Pour une formule à n variables, l'algorithme retient au plus n clauses apprises en tout temps. La complexité spatiale reste alors polynomiale.

2.5.8.2 Propriétés de GRASP

Tout comme le CDCL classique, GRASP est complet et correct, il termine et sa complexité temporelle est exponentielle au pire des cas. Cependant, les différences entre les deux algorithmes font que la plupart des preuves de ces propriétés sont différentes ou doivent être adaptées.

La preuve de correction de GRASP, comme celle du CDCL classique, repose sur la capacité du mécanisme de propagation unitaire à détecter tous les conflits. C'est effectivement le cas avec la détection par compteur de littéraux de GRASP : ceux-ci sont toujours mis à jour lors de l'instanciation ou la désinstanciation de toute variable de la clause. Une clause rendue fausse est donc immédiatement détectée lors de l'instanciation de son dernier littéral.

La complétude de l'algorithme peut être prouvée de la même façon que pour

le CDCL classique. L'algorithme utilisé pour dériver les clauses de conflit n'est pas exactement similaire, mais il s'agit toutefois toujours de résolutions successives de clauses originales ou précédemment apprises. Par conséquent, la preuve de complexité temporelle est elle aussi identique.

La preuve de terminaison nécessite une adaptation plus conséquente, car la fonction telle que définie dans la preuve de Zhang reste constante ou décroît lors d'une résolution de conflit. En effet, dans GRASP, tout conflit est résolu par l'inversion d'une décision, possiblement précédée d'un ou plusieurs sauts arrière dont chacun détruit entièrement un ou plusieurs niveaux de décision. L'inversion de décision cause une décroissance stricte de la fonction si le niveau de décision comprenait au moins deux littéraux avant l'inversion, et ne modifie pas sa valeur dans le cas contraire. Chaque saut arrière diminue strictement la valeur de la fonction.

On peut toutefois retrouver une fonction strictement croissante en considérant uniquement les décisions inversées dans son calcul et en observant cette fonction après chaque résolution de conflit. En effet, chaque résolution de conflit se termine par une inversion de décision ; il est donc évident que le poids de la fonction croît strictement si aucun saut arrière ne précède l'inversion de décision. Dans le cas contraire, il est alors possible que le ou les sauts arrière détruisent une ou plusieurs décisions inversées ; cependant, la nouvelle inversion a lieu à un niveau strictement inférieur et suffit donc à garantir la croissance stricte de la fonction. Notons que cette démonstration peut également être utilisée pour prouver la terminaison de DPLL dans le paradigme itératif.

La correction et la complétude de GRASP ont aussi été prouvées par ses auteurs (Marques-Silva et Sakallah, 1999). Leur preuve de complétude est toutefois différente de celle de Zhang ; au lieu de tracer une démonstration de l'insatisfaisabilité par résolution, elle se base sur la complétude de DPLL et montre que toute partie de l'espace de recherche supplémentaire élaguée par GRASP par rapport à DPLL ne contient aucune solution. À notre connaissance, aucune preuve de terminaison spécifique à GRASP n'a en revanche été publiée.

2.5.9 La destructivité des sauts arrière

À l'aide des sauts arrière et de l'apprentissage des clauses de conflits, le CDCL résout en majeure partie le problème des occurrences multiples de conflits qui handicape lourdement les performances de DPLL. Cependant, les sauts arrière introduisent un nouvel inconvénient dans CDCL qui n'était pas présent dans DPLL.

Lorsqu'un conflit survient dans un CDCL au niveau de décision λ_c et qu'une clause de conflit γ est dérivée, le saut arrière permet de propager γ le plus tôt possible, c'est-à-dire au niveau d'assertion λ_a. Pour retourner à ce niveau, le saut arrière détruit tous les niveaux de décision supérieurs à λ_a, qui est le plus grand niveau de décision présent dans γ hormis λ_c. Cela signifie que tous les niveaux de décision situés entre λ_a et λ_c, qui sont détruits par le saut arrière, ne sont pas impliqués directement dans le conflit.

Illustrons cet aspect par la figure 2.2, qui montre le résultat de l'exécution de CDCL sur la formule utilisée préalablement dans la figure 2.1, en supposant que les mêmes décisions sont prises dans le même ordre. Les décisions successives de x, a et b ne provoquent aucune propagation ni aucun conflit. La décision y rend les deux clauses $c_1 = \neg x \vee \neg y \vee \neg z$ et $c_2 = \neg x \vee \neg y \vee z$ unitaires. Si c_1 est traitée la première, $\neg z$ est propagé au niveau de décision de y, ce qui rend c_2 fausse. Jusqu'ici, l'exécution de CDCL est similaire à celle de DPLL : la branche de recherche obtenue est représenté par la figure 2.2a.

c_2 contient deux littéraux de niveau courant : y et $\neg z$. Ce dernier est le plus récemment instancié ; l'analyse commence donc par résoudre la clause fausse c_2 avec l'antécédent de $\neg z$, c_1, sur la variable z. Le résultat de cette résolution est la clause $\gamma = \neg x \vee \neg y$, qui ne contient qu'un seul littéral de niveau maximal, $\neg y$. L'analyse est terminée et γ est la clause de conflit. Le niveau d'assertion est $\lambda(x)$, puisque x est la seule variable de γ qui n'appartient pas au niveau courant. CDCL retourne donc au niveau $\lambda(x)$ en défaisant les niveaux $\lambda(y)$, $\lambda(b)$ et $\lambda(a)$, puis propage l'assertion $\neg y$ au niveau $\lambda(x)$. La figure 2.2b montre l'état final de la recherche après résolution du conflit.

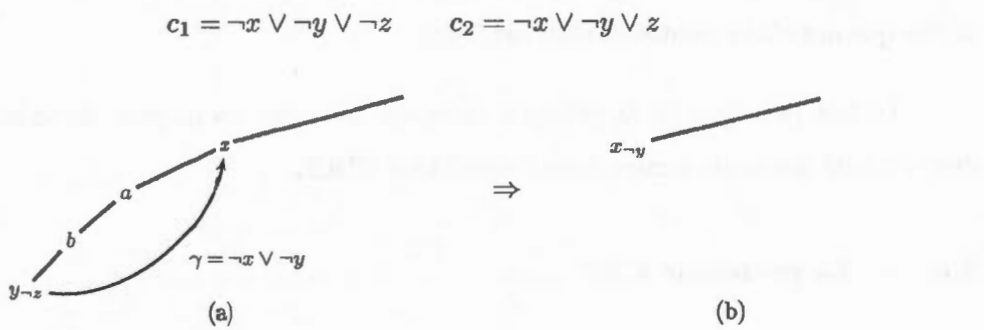


FIGURE 2.2: Illustration de la destructivité du saut arrière de CDCL. La figure 2.2a décrit l'arbre de recherche produit par CDCL sur la formule $\mathcal{F}(\{a, b, x, y, z\}, \{c_1, c_2\})$ après les décisions consécutives x, a, b et y et la propagation de $\neg z$ par la clause unitaire c_1, ainsi que la clause γ résultant de l'analyse du conflit déclenché par la clause fausse c_2 et le niveau de retour du saut arrière. La figure 2.2b représente l'arbre de recherche après la résolution du conflit.

Cet exemple illustre bien l'évitement des occurrences multiples de conflits dans CDCL, puisque la propagation de y empêche de redéclencher le même conflit dans toute la branche de la recherche correspondant à l'instanciation de x. Il illustre également la destructivité du saut arrière : les décisions a et b sont défaites par le saut arrière alors qu'elles n'ont aucun rapport avec le conflit rencontré.

L'efficacité de l'élagage de CDCL a donc comme désavantage de détruire une partie du travail de recherche effectué. En pratique, il est fréquent qu'un niveau de décision contienne plusieurs centaines, voire plusieurs milliers de propagations. Le saut arrière détruit donc également toutes les inférences déduites des décisions défaites.

L'impact négatif de cette destruction est évident. Les instanciations défaites auraient pu contribuer à déclencher un conflit ultérieur, voire faire partie d'un modèle. Dans les deux cas, l'algorithme va possiblement répéter la chaîne de décisions et de propagations une seconde fois, ce qui alourdit son exécution. L'effet est particulièrement négatif dans le cas où le conflit et les niveaux défaits appartiennent à des composantes connexes distinctes du problème, c'est-à-dire où ils n'ont aucune interaction possible, même indirectement. Lorsque la recherche reviendra aux instanciations défaites, elle refera forcément les mêmes décisions et les mêmes propagations dans le même ordre (à

moins que des choix aléatoires interviennent).

Le but principal de la présente thèse est d'étudier les moyens de réduire cette destructivité des sauts arrière dans l'algorithme CDCL.

2.6 Le problème CSP

Cette section présente plus brièvement le problème de satisfaction de contraintes, ou problème CSP, en mettant l'accent sur les principaux algorithmes complets pour sa résolution. La sous-section 2.6.1 fournit une définition formelle de ce problème. La sous-section 2.6.2 explique les points communs entre les problèmes SAT et CSP, en présentant notamment des réductions polynomiales entre ces deux problèmes. La sous-section 2.6.3 présente différents algorithmes d'inférence sur le problème CSP, tandis que les sous-sections 2.6.4 et 2.6.5 sont consacrées aux algorithmes de recherche en profondeur pour CSP, respectivement à retour arrière et à saut arrière.

2.6.1 Définition du CSP

Un **réseau de contraintes** est caractérisé par un sextuplet $R(\mathcal{X}, \mathcal{V}, \mathcal{C}, \delta, \pi, \tau)$ où $\mathcal{X}$, $\mathcal{V}$ et $\mathcal{C}$ sont des ensembles finis et δ, π et τ des fonctions.

- $\mathcal{X}$, $\mathcal{V}$ et $\mathcal{C}$ sont respectivement les **variables**, les **valeurs** et les **contraintes** du réseau.
- La fonction $\delta : \mathcal{X} \to \mathcal{P}(\mathcal{V})$ associe à chaque variable $x \in \mathcal{X}$ son **domaine**. Le domaine d'une variable est l'ensemble des valeurs auxquelles elle peut être instanciée, c'est-à-dire un sous-ensemble de $\mathcal{V}$: $\forall x \in \mathcal{X}, \delta(x) \subseteq \mathcal{V}$.
- La fonction $\pi : \mathcal{C} \to \mathcal{P}(\mathcal{X})$ associe à chaque contrainte $c \in \mathcal{C}$ sa **portée**, qui est l'ensemble des variables sur lesquelles la contrainte c s'applique. On suppose en général que toutes les contraintes d'un réseau de contraintes ont une portée distincte : $\forall \{c_1, c_2\} \subseteq \mathcal{C}, \pi(c_1) \neq \pi(c_2)$.
- Soit $X \subseteq \mathcal{X}$ un ensemble de variables. Un **tuple** sur X est une fonction $t : X \to \mathcal{V}$ qui associe à chaque variable $x \in X$ une valeur de son domaine :

$\forall x \in X$, $t(x) \in \delta(x)$. Nous noterons $\mathcal{T}(X)$ l'ensemble des tuples sur X. La fonction $\tau : \mathcal{C} \to \bigcup_{X \subseteq \mathcal{X}} \mathcal{P}(\mathcal{T}(X))$ associe à chaque contrainte $c \in \mathcal{C}$ un ensemble de **tuples autorisés** sur sa portée : $\forall c \in \mathcal{C}$, $\tau(c) \subseteq \mathcal{T}(\pi(c))$.

L'**arité** d'une contrainte est le nombre de variables que comporte sa portée. Un réseau de contraintes est dit **binaire** s'il ne comporte que des contraintes d'arité au plus 2 : $\forall c \in \mathcal{C}$, $|\pi(c)| \leq 2$.

Les paramètres de taille principaux d'un réseau de contraintes sont le nombre de variables $n = |\mathcal{X}|$, le nombre de contraintes $m = |\mathcal{C}|$, la taille maximale des domaines $d = \max_{x \in \mathcal{X}}(|\delta(x)|)$, bornée par $|\mathcal{V}|$, l'arité maximale des contraintes $k = \max_{c \in \mathcal{C}}(|\pi(c)|)$, bornée par n, et le nombre maximal de tuples autorisés par contrainte $a = \max_{c \in \mathcal{C}}(|\tau(c)|)$, borné par d^k. En effet, la taille de l'encodage du réseau est au plus proportionnelle à $n \times d + m \times (k + a)$.

Une **instanciation** $\sigma : \mathcal{X} \nrightarrow \mathcal{V}$ est une fonction partielle qui associe à une variable $x \in \mathcal{X}$ une valeur de son domaine : $\forall x \in \mathcal{D}(\sigma)$, $\sigma(x) \in \delta(x)$. Une instanciation est **complète** si $\mathcal{D}(\sigma) = \mathcal{X}$. Une **solution** d'un réseau de contraintes R est une instanciation complète σ qui respecte toutes les contraintes du réseau : pour toute contrainte $c \in \mathcal{C}$, il existe un tuple $t \in \tau(c)$ tel que $\forall x \in \pi(c)$, $\sigma(x) = t(x)$. On dit alors que σ **satisfait** le réseau de contraintes R. R est dit **satisfaisable** s'il admet au moins une solution, ou **insatisfaisable** sinon.

Le **problème de satisfaction de contraintes** (ou **CSP** pour *constraint satisfaction problem*) est un problème de recherche. Il consiste, pour un réseau de contraintes donné, à déterminer s'il admet au moins une solution. Le problème de satisfaction de contraintes binaires est identique au CSP, mais restreint aux réseaux de contraintes binaires.

Tout comme dans le cas de SAT, il existe de nombreuses variantes de CSP, comme **#CSP**, **All-CSP**, **Max-CSP** et **CSP pondéré** (aussi appelé **CSP partiel**), définies de la même façon que les variantes de SAT correspondantes.

2.6.2 Rapports entre CSP et SAT

Les ressemblances entre CSP et SAT sont évidentes : les deux problèmes consistent à instancier un ensemble de variables tout en respectant un ensemble de contraintes. Les différences essentielles entre ces deux formulations sont d'une part le type de variables utilisées et d'autre part la façon dont les contraintes sont exprimées :

- Les variables du problème SAT sont booléennes, elles peuvent donc prendre uniquement deux valeurs différentes. Les variables d'un CSP peuvent prendre un nombre arbitraire (quoique fini) de valeurs.
- Dans une instance de SAT, chaque clause $c = l_1 \vee l_2 \vee \ldots \vee l_i \in \mathcal{C}$ revient à interdire un tuple particulier sur la portée $\{\nu(l_1), \nu(l_2), \ldots, \nu(l_i)\}$; plusieurs clauses peuvent donc correspondre à une même portée. Au contraire, chaque contrainte $c \in \mathcal{C}$ dans un CSP énumère les tuples autorisés sur sa portée $\pi(c)$.

Tout comme SAT, CSP est un problème NP-complet (Mackworth, 1977), ce qui implique qu'il existe une réduction polynomiale de CSP vers SAT et vice-versa. De telles réductions sont relativement simples à formuler, compte tenu des similitudes entre les deux problèmes.

Proposition 2.9. *Il existe une réduction polynomiale du CSP vers le CSP binaire.*

Démonstration. Soit $R(\mathcal{X}, \mathcal{V}, \mathcal{C}, \delta, \pi, \tau)$ une instance du problème CSP. Nous allons construire un CSP binaire $R'(\mathcal{X}', \mathcal{V}', \mathcal{C}', \delta', \pi', \tau')$ tel que R admet une solution si et seulement si R' admet une solution. Soient n, m, d, k et a respectivement le nombre de variables, le nombre de contraintes, la taille maximale des domaines de variables, l'arité maximale des contraintes et le nombre maximal de tuples autorisés par contrainte dans R. Soient n', m', d', k' et a' les propriétés correspondantes dans R' (par définition, $k' = 2$). Cette construction suit la technique de l'encodage dual d'un réseau de contraintes (Dechter et Pearl, 1989) :

- Chaque variable de R' représente une contrainte de R : $\mathcal{X}' = \mathcal{C}$. On a donc $n' = m$.
- Les valeurs de R' sont l'ensemble des tuples autorisés par les contraintes de R :

$\mathcal{V}' = \bigcup_{c \in \mathcal{C}} \tau(c)$.

- Le domaine d'une variable de R' est l'ensemble des tuples autorisés par la contrainte de R correspondante : $\forall x' \in \mathcal{X}'$, $\delta'(x') = \tau(x')$. On a donc $d' = a$, ce qui implique $a' \leq a^2$ puisque R' est un CSP binaire.
- Chaque contrainte de R' correspond à une paire de clauses de R (donc de variables de R') ayant une ou plusieurs variables de R en commun : $\mathcal{C}' = \{\{x'_1, x'_2\} \in \mathcal{X}' \mid \pi(x'_1) \cap \pi(x'_2) \neq \emptyset\}$. Par conséquent, $m' \leq m(m-1)$.
- La portée d'une contrainte de R' est le couple de variables de R' qui lui correspond : $\forall c' \in \mathcal{C}'$, $\pi'(c') = c'$.
- Soit $c' \in \mathcal{C}'$ une contrainte de R', $\pi'(c') = \{x'_1, x'_2\}$ sa portée et $t' \in \mathcal{T}(\{x'_1, x'_2\})$ un tuple sur cette portée. $t_1 = t'(x'_1)$ et $t_2 = t'(x'_2)$ sont dans R des tuples autorisés sur les contraintes de R correspondantes : $t_1 \in \tau(x'_1)$ et $t_2 \in \tau(x'_2)$. t' est un tuple autorisé par c' si et seulement si les tuples de R t_1 et t_2 associent les mêmes valeurs à leurs variables de R en commun : $\tau(c') = \{t' \in \mathcal{T}(\{x'_1, x'_2\}) \mid \forall x \in \pi(x'_1) \cap \pi(x'_2),\ t_1(x) = t_2(x)\}$.

Une instanciation de R' est une solution si et seulement si toutes les valeurs assignées aux variables de R' correspondent à des tuples autorisés dans R qui assignent les mêmes valeurs aux variables de R. Par conséquent, R et R' sont équisatisfaisables. De plus, tous les paramètres de taille de R' sont des fonctions polynomiales des paramètres de taille de R. L'encodage dual est donc bien une réduction polynomiale du CSP vers le CSP binaire. □

Proposition 2.10. *Il existe une réduction polynomiale du CSP binaire vers SAT.*

Démonstration. Soit $R(\mathcal{X}, \mathcal{V}, \mathcal{C}, \delta, \pi, \tau)$ un réseau de contraintes binaire. Nous allons construire une formule CNF $F(\mathcal{V}', \mathcal{C}')$ équisatisfaisable en utilisant l'encodage dit direct (de Kleer, 1989). Soient n, m, d et a respectivement le nombre de variables, le nombre de contraintes, la taille maximale des domaines de variables et le nombre maximal de tuples autorisés par contrainte dans R. Soient n', m' et k' le nombre de variables, le nombre de clauses et la taille maximale des clauses dans F.

- F contient une variable pour chaque valeur du domaine de chaque variable de $R : \mathcal{V}' = \{v'_{x,v} \mid x \in \mathcal{X}, v \in \delta(x)\}$. L'instanciation de $v'_{x,v}$ (resp. de $\neg v'_{x,v}$) signifie que l'on instancie (resp. que l'on n'instancie pas) x à la valeur v. On a donc $n' \leq n \times d$.
- Pour chaque variable $x \in \mathcal{X}$ de R, F contient une clause qui signifie que x doit prendre au moins une valeur : $\forall x \in \mathcal{X}, \bigvee_{v \in \delta(x)} v'_{x,v} \in \mathcal{C}'$. F contient n clauses de ce type, de taille au plus d.
- Pour chaque paire de valeurs du domaine d'une variable $x \in \mathcal{X}$ de R, F contient une clause qui signifie que x ne peut pas prendre ces deux valeurs à la fois : $\forall x \in \mathcal{X}, \forall \{v_1, v_2\} \subseteq \delta(x), (\neg v'_{x,v_1} \vee \neg v'_{x,v_2}) \in \mathcal{C}'$. F contient au plus $n \times d(d-1)$ clauses binaires de ce type.
- Pour chaque tuple non-autorisé par une contrainte de R, une clause de F interdit la conjonction de variables correspondante :

$$\forall c \in \mathcal{C}, \forall t \in \mathcal{T}(\pi(c)) \setminus \tau(c), \bigvee_{x \in \pi(c)} \neg v'_{x,t(x)} \in \mathcal{C}'.$$

  Comme R est un réseau binaire, ce type de clause est également binaire et F en contient au plus $m \times d^2$.

Au final, il est clair qu'une instanciation σ de $\mathcal{V}$ est un modèle de F si et seulement s'il encode une instanciation complète valide de $\mathcal{X}$ (qui associe une et une seule valeur à chaque variable) et qui vérifie toutes les contraintes de $\mathcal{C}$. De plus, tous les paramètres de taille de F sont bornées par des fonctions polynomiales des paramètres de taille de $R : n' \leq n \times d$, $m' \leq d^2(n+m) - n \times (d-1)$ et $k' \leq d$. L'encodage direct est donc bien une réduction polynomiale du CSP binaire vers SAT. □

Proposition 2.11. *Il existe une réduction polynomiale de CSP vers SAT.*

Démonstration. On peut l'obtenir par réductions polynomiales successives de CSP vers le CSP binaire, puis du CSP binaire vers SAT. □

Notons que l'encodage direct peut être généralisé afin d'encoder vers SAT un CSP quelconque. Toutefois, un tel encodage n'est pas forcément polynomial. En effet, supposons qu'un CSP à n variables booléennes contient une contrainte dont la portée est l'ensemble de toutes les variables et qui ne contient qu'un seul tuple autorisé. Si l'on utilise exactement l'encodage décrit ci-dessus, cette contrainte génère $2^n - 1$ clauses dans la formule propositionnelle encodée, chacune de ces clauses correspondant à l'interdiction d'un tuple.

Proposition 2.12. *Il existe une réduction polynomiale de SAT vers CSP.*

Démonstration. Soit $F(\mathcal{V}, \mathcal{C})$ une formule CNF. Nous allons construire un réseau de contraintes binaire $R(\mathcal{X}', \mathcal{V}', \mathcal{C}', \delta', \pi', \tau')$ qui admet une solution si et seulement si F est satisfaisable. Soient n, m et k respectivement le nombre de variables, le nombre de clauses et la taille maximale des clauses de F. Soient n', m', d' et a' respectivement le nombre de variables, le nombre de contraintes, la taille maximale des domaines et le nombre maximal de tuples autorisés par contrainte de R. Nous allons construire R en utilisant l'encodage dit par littéraux (Bennaceur, 1996) :

- R contient une variable pour chaque clause de F : $\mathcal{X} = \mathcal{C}$. Par conséquent, $n' = m$.
- L'ensemble des valeurs de R correspond à l'ensemble des littéraux de F : $\mathcal{V}' = \mathcal{L}$.
- Le domaine de chaque variable de R est l'ensemble des littéraux contenus dans la clause correspondante de F : $\forall x \in \mathcal{X}$, $\delta(x) = x$. On a donc $d' = k$.
- Les contraintes de R sont les paires de clauses de F ayant au moins une variable de F en commun : $\mathcal{C}' = \{\{c_1, c_2\} \subseteq \mathcal{C} \mid \nu(c_1) \cap \nu(c_2) \neq \emptyset\}$. On peut donc borner le nombre de contraintes de R par $m' \leq m(m-1)$.
- La portée d'une contrainte de R est la paire de variables de R qui correspond à la paire de clauses de F : $\forall c' \in \mathcal{C}'$, $\pi(c') = c'$.
- Toute contrainte de R autorise les tuples correspondant à des paires de littéraux non-opposés : $\forall c' = \{c_1, c_2\} \in \mathcal{C}'$, $\tau(c') = \{t \in \mathcal{T}(\pi(c')) \mid t(c_1) \neq \neg t(c_2)\}$. On a donc $a' \leq k(k-1)$.

Soit σ une solution de R. Alors l'ensemble $\bigcup_{x \in \mathcal{X}} \{\sigma(x)\}$ est une instanciation (possiblement partielle) de $\mathcal{V}$ qui rend vrai au moins un littéral dans chaque clause de $\mathcal{C}$. F est donc satisfaisable. Inversement, à partir d'un modèle de F, on peut facilement construire une solution de R : il suffit d'assigner à chaque variable de R un des littéraux qui satisfait la clause correspondante avec ce modèle de F. R et F sont donc équisatisfaisables. Comme les paramètres de taille de R sont polynomiaux par rapport aux paramètres de taille de F, l'encodage par littéraux est une réduction polynomiale de SAT vers CSP. □

Ces réductions montrent que, malgré la plus grande liberté de CSP sur la taille des domaines de ses variables, SAT et CSP ont une expressivité équivalente et permettent d'encoder les même problèmes. De plus, leur NP-complétude implique une complexité temporelle similaire pour leurs algorithmes totalement corrects respectifs.

2.6.3 Algorithmes d'inférence sur les CSP

Nous avons vu que le problème SAT peut être résolu ou simplifié en utilisant différentes techniques d'inférences qui rajoutent des contraintes aux problèmes, soit en forçant l'instanciation de variables (propagations unitaires, littéraux purs), soit en rajoutant de nouvelles clauses (résolution). De même, l'inférence dans les CSP permet d'éliminer des valeurs des domaines de variables et de définir de nouvelles contraintes ou de retirer des tuples de contraintes existantes.

Les principales techniques d'inférence dans CSP reposent sur la notion de **consistance**. Une instanciation σ est dite consistante si elle respecte toutes les contraintes dont elle instancie toutes les variables ; autrement dit, si pour toute contrainte $c \in \mathcal{C}$ telle que $\pi(c) \subseteq \mathcal{D}(\sigma)$, le tuple $\sigma|_{\pi(c)}$appartient à $\tau(c)$. Une solution du réseau de contraintes est donc une instanciation complète et consistante. Les techniques d'inférences les plus courantes modifient un réseau de contraintes afin d'assurer que les instanciations consistantes d'une certaine taille peuvent être étendues tout en préservant leur consistance.

L'**arc-consistance** est une des propriétés de consistance les plus simples et les

plus utilisées. Un réseau de contraintes R est dit arc-consistant si, pour toute variable $x \in \mathcal{X}$, toute valeur $v \in \delta(x)$ et toute contrainte binaire $\{x, x'\} \in \mathcal{C}$, il existe une valeur $v' \in d(x')$ telle que le tuple $\{x \mapsto v, x' \mapsto v'\}$ appartient à $\tau(\{x, x'\})$. v' est alors un **support** de v dans la contrainte $\{x, x'\}$. Dans un réseau arc-consistant, pour toute instanciation consistante d'une variable $\sigma = \{x \mapsto v\}$ et toute autre variable x', il existe une valeur v' telle que $\sigma' = \{x \mapsto v, x' \mapsto v'\}$ est aussi consistante. Par conséquent, toute instanciation consistante de taille 1 peut être étendue en une instanciation arbitraire de taille 2 en utilisant une variable quelconque. Un réseau peut être rendu arc-consistant en éliminant successivement toutes les valeurs qui n'ont aucun support dans une contrainte donnée. Cette opération peut être réalisée en temps $O(m_2 \times d^2)$ où $m_2 \leq m$ est le nombre de contraintes binaires du réseau (Mohr et Henderson, 1986).

L'arc consistance peut être généralisée à une taille quelconque d'instanciation. Ainsi, un réseau est i-consistant si toute instanciation consistante de $i-1$ variables peut être étendue en une instanciation consistante de i variables en rajoutant une variable quelconque. L'arc-consistance correspond au cas $i = 2$. Un réseau est fortement i-consistant s'il est j-consistant $\forall j \in \{1, 2, \ldots, i\}$. Un réseau de contraintes peut être rendu fortement i-consistant (en rajoutant des contraintes d'arité 1 à $i-1$ ou en retirant des tuples de telles contraintes) en temps $O(n^i d^i)$ (Cooper, 1989; Dechter, 1992) ; l'algorithme correspondant est donc polynomial si l'on considère i comme un paramètre fixé.

Si un réseau à n variables est fortement n-consistant, alors ce réseau est satisfaisable si et seulement si toutes les variables ont un domaine non-vide. Il est donc possible de résoudre un CSP en rendant le réseau fortement n-consistant. L'algorithme n'est cependant plus polynomial, puisque la complexité est dans ce cas de $O(n^n d^n)$. Généralement, des formes limitées d'inférence, comme l'arc-consistance, sont utilisées dans les algorithmes de recherche pour élaguer plus efficacement l'espace de recherche.

Notons qu'il existe également la notion de consistance directionnelle, qui considère uniquement la consistance selon un certain ordre des variables. L'intérêt principal est que

l'on peut vérifier la satisfaisabilité d'un CSP en le rendant directionnellement $(w^* + 1)$-consistant, où w^* est la largeur induite du réseau par rapport à l'ordre des variables considéré (Dechter et Pearl, 1987). Or, il existe au moins un ordre de variables pour lequel $w^* = w$, où w est la largeur d'arborescence du graphe sous-jacent (voir sous-section 3.1.1). Par conséquent, il est possible pour certaines classes de CSP de déterminer efficacement leur satisfaisabilité par inférence. Par exemple, comme tout arbre a une largeur d'arborescence égale à 1, la satisfaisabilité d'un CSP arborescent est vérifiable en le rendant 2-consistant.

2.6.4 Recherche en profondeur avec retour arrière sur les CSP

Comme dans le cas de SAT, les algorithmes totalement corrects les plus utilisés en pratique pour résoudre les CSP reposent sur un principe de recherche en profondeur : ils cherchent à étendre progressivement une instanciation consistante des variables en partant de l'instanciation vide.

L'algorithme de recherche en profondeur pur (sans inférence) sur CSP est généralement noté BT, pour *backtracking* (retour arrière). Le déroulement de BT est très similaire à celui de DPLL pour SAT. Soit σ l'instanciation courante. À chaque étape, BT choisit arbitrairement une variable libre $x \in \mathcal{X} \setminus \mathcal{D}(\sigma)$ à assigner, ainsi qu'une valeur $v \in \delta(x)$ telle que $\sigma' = \sigma \cup \{x \mapsto v\}$ reste consistante. S'il existe une telle valeur, BT continue avec σ' et cherche une nouvelle variable à instancier. Si aucune valeur de x ne convient, l'algorithme effectue un retour arrière en défaisant la décision précédente, puis cherche à l'instancier avec une autre valeur si possible. Le réseau de contraintes est satisfaisable si BT parvient à construire une instanciation complète consistante ; il est insatisfaisable s'il épuise toutes les valeurs possibles de la première variable choisie. Il est évident que la gestion des conflits de DPLL est similaire à celle de BT ; une valeur inconsistante avec l'instanciation courante dans BT correspond à une clause rendue fausse dans DPLL, et l'inversion des décisions dans DPLL correspond à la recherche de valeurs alternatives dans le cas de variables booléennes.

La recherche en profondeur sur les CSP peut être combinée à des techniques d'inférence afin d'élaguer plus efficacement l'espace de recherche. Les deux tactiques les plus couramment utilisées sont la vérification en avant (*forward checking*, notée FC) et la maintenance de l'arc-consistance (*maintaining arc-consistency*, notée MAC).

FC (Haralick et Elliott, 1980) utilise une inférence minimale au moment du choix de la valeur à associer à une variable $x \in \mathcal{X} \setminus \mathcal{D}(\sigma)$. Si BT s'assure uniquement que la valeur v choisie ne rend pas l'instanciation courante σ inconsistante, FC vérifie de plus que l'ajout de l'assignation $x \mapsto v$ n'empêche pas l'instanciation future des autres variables. Plus formellement, FC peut assigner v à x seulement si, pour toute variable $x' \in \mathcal{X} \setminus (\mathcal{D}(\sigma) \cup \{x\})$, il existe une valeur $v' \in \delta(x')$ telle que l'instanciation $\sigma \cup \{x \mapsto v, x' \mapsto v'\}$ est consistante. De plus, l'algorithme élimine (dans la branche de recherche courante) les valeurs des variables x' incompatibles avec $\sigma \cup \{x \mapsto v\}$, ce qui contribue également à l'élagage.

L'algorithme MAC (Sabin et Freuder, 1994), quant à lui, rend le réseau de contraintes arc-consistant avant la première décision et après tout ajout d'une assignation (lors d'une assignation d'une valeur v à une variable x, on considère que le domaine $\delta(x)$ est restreint au singleton $\{v\}$). L'instanciation courante peut alors être prouvée inconsistante si un domaine de variable devient vide pendant la maintenance de l'arc-consistance ; au contraire, dans un algorithme BT simple, seul le domaine de la décision courante est modifié.

Il existe également des stratégies d'inférences intermédiaires entre FC et MAC, par exemple le *look-ahead* complet ou partiel (Haralick et Elliott, 1980) qui après chaque décision effectue une forme limitée de maintenance de l'arc consistance.

Tout comme pour DPLL, on peut prouver par analogie avec la réfutation par résolution que BT, FC, MAC et toutes les variantes ajoutant des techniques d'inférences à BT ont une complexité temporelle exponentielle selon le nombre de variables au pire des cas(Mitchell, 1998).

2.6.5 Recherche en profondeur avec saut arrière sur les CSP

Le phénomène d'occurrence multiples de conflits, que nous avons remarqué dans le cadre du DPLL (voir la sous-section 2.4.4), a également été identifié dans les algorithmes à retour arrière sur les CSP ; il est par exemple mentionné par Mackworth (1977) sous le nom de *trashing*. Différents types d'algorithmes à saut arrière ont donc été développés pour la résolution du CSP, dont le plus répandu est CBJ, pour *conflict-directed backjumping* (Prosser, 1993). CBJ a été précédé par BJ (*backjumping*) (Gaschnig, 1979), qui n'effectue des sauts arrière que lors de certains types de conflits et utilise sinon un retour arrière simple, et par GBJ (*graph-based backjumping*) (Dechter, 1990), qui effectue des sauts arrière pour tous les conflits mais qui revient parfois moins loin en arrière que CBJ. Notons aussi que CBJ est la principale source d'inspiration du CDCL (Bayardo Jr. et Schrag, 1997), qui n'est autre qu'une transposition de son concept dans le contexte du problème SAT.

Au cours de la recherche, pour toute variable $x \in \mathcal{X}$ et toute valeur $v \in \delta(x)$, CBJ maintient une raison d'élimination $\rho(v) \subseteq \sigma$ qui est un ensemble de valeurs des variables précédemment instanciées. Si une valeur v est éliminée du domaine d'une variable x sans avoir été instanciée, que ce soit par la recherche elle-même ou par une technique d'inférence, alors elle est inconsistante avec l'instanciation courante. Cela signifie qu'il existe une contrainte $c \in C$ dont la portée $\pi(c)$ contient x et qui ne contient pas de support pour x compatible avec σ. Soit $I = \pi(c) \cap \mathcal{D}(\sigma)$ l'ensemble des variables de la contrainte c déjà instanciées ; alors $\forall t \in \tau(c)$, $t(x) \neq v$ ou $\exists x' \in I \,|\, t(x') \neq \sigma(x')$. L'inconsistance de $x \mapsto v$ avec σ est donc justifiée par l'instanciation des variables de I. Par conséquent, la raison d'élimination de v consiste en cet ensemble d'instanciations : $\rho(v_{i,\iota}) = \sigma|_I$.

Si toutes les valeurs d'une variable $x \in \mathcal{X}$ sont éliminées, alors l'instanciation courante σ est inconsistante. La raison du conflit γ est un tuple obtenu par l'union de toutes les raisons d'éliminations des valeurs de x : $\gamma = \bigcup_{v \in \delta(x)} \rho(v) \subseteq \sigma$. La variable responsable du conflit, notée x_γ, est la variable dans $\mathcal{D}(\gamma)$ la plus récemment instan-

ciée. Soit $v_\gamma = \sigma(x_\gamma)$ la valeur actuellement assignée à x_γ. CBJ effectue un saut arrière en désinstanciant les variables dans l'ordre inverse de leur instanciation jusqu'à x_γ incluse. Toutes les valeurs de variables éliminées dont la raison contenait une des valeurs désinstanciées sont rétablies (et leur raison d'élimination est invalidée).[12] Enfin, v_γ est éliminée du domaine de x_γ et la raison de son élimination est $\rho(v_\gamma) = \gamma \setminus \{v_\gamma\}$.

Si l'algorithme réussit à construire une instanciation complète consistante, le réseau de contraintes est satisfaisable; si le domaine de $x_1 \in \mathcal{X}$, la première variable instanciée, devient vide, le réseau est insatisfaisable.

Les similarités entre CBJ et le CDCL sont évidentes; les raisons d'élimination des valeurs dans CBJ tiennent un rôle similaire aux antécédents des propagations dans CDCL, et les conflits sont diagnostiqués respectivement par union des raisons d'élimination ou par résolution sur les antécédents. Dans CBJ, il existe également différentes stratégies pour réduire la taille des raisons de conflits, et celles-ci peuvent être apprises sous la forme de *nogoods*, c'est-à-dire de tuples interdits (Dechter, 1990). Comme CDCL, CBJ a une complexité temporelle de $2^{\Omega(n)}$ au pire des cas (Mitchell, 1998). Enfin, les sauts arrière de CBJ peuvent pareillement provoquer la destruction d'une partie significative de la recherche. En effet, ils peuvent désinstancier des variables sans rapport direct avec le conflit, et ces désinstanciations provoquent elles-mêmes le rétablissement de valeurs éliminées.

---

12. Pour être précis, le CBJ de base rétablit toutes les valeurs éliminées des variables non-instanciées après le saut arrière sans tenir compte de leur raison d'élimination. Le rétablissement sélectif que nous décrivons correspond plus exactement à une variante nommée CBJ-NG (Zivan et al., 2009).

CHAPITRE III

MINIMISER LA PERTE DE PROGRESSION LORS DES SAUTS ARRIÈRE

Les contributions de cette thèse ont comme objectif commun de chercher à réduire l'impact destructif des sauts arrière sur l'instanciation partielle courante dans l'algorithme CDCL. Ce chapitre présente différentes variantes de CDCL (ou de CBJ, son équivalent pour la résolution de CSP) qui ont également pour effet de réduire la destructivité des sauts arrière.

La section 3.1 détaille différentes méthodes de décomposition arborescente pour la résolution de SAT et CSP. La section 3.2 couvre les techniques de détection de sous-problèmes résiduels connexes. La section 3.3 introduit l'heuristique de sauvegarde de phase dans l'algorithme CDCL. La section 3.4 décrit différents algorithmes à sauts arrière partiels pour la résolution de CSP. Enfin, la section 3.5 aborde des algorithmes qui, au contraire des précédents, tendent à accroître la destructivité des sauts arrière.

3.1 Les décompositions arborescentes

Les décompositions arborescentes sont un concept de théorie des graphes qui permet d'encapsuler un graphe quelconque dans une description arborescente. Les décompositions arborescentes permettent de définir des classes de graphes, bornées par des largeurs d'arborescence arbitraire, sur lesquelles des problèmes NP-complets dans le cas général peuvent être résolus en temps polynomial selon la taille des graphes. Si les problèmes SAT et CSP ne sont pas des problèmes de graphe au sens strict, les relations

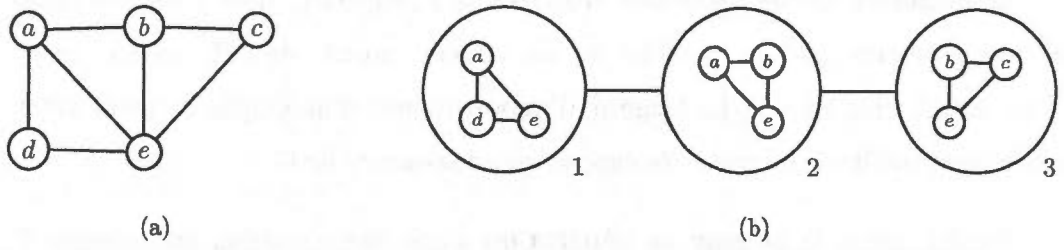
entre leurs variables peuvent toutefois être décrites à l'aide de graphes, et les techniques de décompositions arborescentes peuvent donc y être appliquées. Nous distinguerons deux grandes catégories d'algorithmes utilisant des décompositions arborescentes pour la résolution de SAT et CSP : les décompositions explicites, qui résolvent séparément des sous-problèmes et se servent de leurs solutions pour retrouver une solution du problème global, et les décompositions implicites, qui simulent une résolution indépendante des sous-problèmes à l'intérieur d'une recherche en profondeur sur le problème entier. Nous verrons notamment que les décompositions implicites ont pour effet de limiter la destructivité des sauts arrière dans une recherche en profondeur, sans toutefois l'éliminer totalement.

Les sous-sections 3.1.1 et 3.1.2 définissent les décompositions arborescentes et expliquent leur utilité pour la résolution de problèmes de graphes NP-complets. La sous-section 3.1.3 présente différentes variantes, cas particuliers et généralisations de la définition la plus courante de décomposition arborescente. La sous-section 3.1.4 introduit différentes représentations d'instances de problèmes SAT et CSP sous la forme de graphes. Les sous-sections 3.1.5 et 3.1.6 détaillent les algorithmes de résolution de SAT et CSP par décompositions respectivement explicites et implicites. La sous-section 3.1.7 explique la façon dont les décompositions implicites limitent la destructivité des sauts arrière.

3.1.1 Décomposition arborescente d'un graphe

Soit $G(\mathcal{S}, \mathcal{A})$ un graphe, orienté ou non (nous supposerons dans nos notations que G est non-orienté). Une **décomposition arborescente** de G (Robertson et Seymour, 1986) est un triplet $\mathcal{T}(T(\mathcal{N}, \mathcal{A}'), \varsigma, \alpha)$ tel que :

- $T(\mathcal{N}, \mathcal{A}')$ est un arbre ;
- $\varsigma : \mathcal{N} \rightarrow \mathcal{P}(\mathcal{S})$ est une fonction qui associe à tout nœud de T un sous-ensemble des sommets de G ;
- $\alpha : \mathcal{N} \rightarrow \mathcal{P}(\mathcal{A})$ est une fonction qui associe à tout nœud de T un sous-ensemble des arêtes de G ;



(a) (b)

FIGURE 3.1: Exemple de décomposition arborescente d'un graphe. La figure 3.1a présente un graphe à cinq sommets et sept arêtes. La figure 3.1b fournit une décomposition arborescente de ce graphe, qui comporte trois nœuds contenant chacun un sous-graphe à trois sommets et trois arêtes.

- Chaque sommet de G est associé à au moins un nœud de T : $\forall s \in \mathcal{S}$, $\exists n \in \mathcal{N} \mid s \in \varsigma(n)$;
- Chaque arête de G est associée à au moins un nœud de T : $\forall a \in \mathcal{A}$, $\exists n \in \mathcal{N} \mid a \in \alpha(n)$;
- Si une arête de G est associée à un nœud de T, ses deux sommets y sont aussi associés : $\forall a \in \mathcal{A}$, $n \in \mathcal{N}$, $a \in \alpha(n) \Rightarrow a \subseteq \varsigma(n)$;
- Pour tout sommet de G $s \in \mathcal{S}$, la restriction de T aux nœuds associés à s est connexe. Autrement dit, pour tout ensemble de trois nœuds $\{n_1, n_2, n_3\} \subseteq \mathcal{N}$ tels que $s \in \varsigma(n_1)$ et $s \in \varsigma(n_2)$, si n_3 fait partie du chemin unique entre n_1 et n_2, alors $s \in \varsigma(n_3)$.

La dernière propriété énoncée est appelée la **propriété de connectivité** des décompositions arborescentes. Elle implique que pour toute paire de nœuds $\{n_1, n_2\} \subseteq \mathcal{N}$ voisins dans T (tels que $\{n_1, n_2\} \in \mathcal{A}'$), l'ensemble de sommets $\varsigma(n_1) \cap \varsigma(n_2)$ est un séparateur du graphe G (pour tout nœud $n \in \mathcal{N}$, $\varsigma(n)$ est donc également un séparateur de G).

La figure 3.1 illustre cette définition par un exemple de décomposition arborescente d'un graphe. Cette décomposition est bien valide, car chaque sommet et chaque arête du graphe est présent dans au moins un noeud de la décomposition. De plus, e, le seul sommet présent à la fois dans les nœuds 1 et 3, la seule paire de nœuds non-voisins, est également présent dans le nœud 2, qui est l'unique nœud sur le chemin entre les nœuds 1 et 3 ; la propriété de connectivité est donc bien respectée.

La **largeur** d'une décomposition arborescente $\mathcal{T}$, notée $\omega(\mathcal{T})$, est le nombre maximal de sommets de G associé à un même nœud de T, moins un : $\omega(\mathcal{T}) = \max_{n \in \mathcal{N}}(|\varsigma(n)| - 1)$. La **largeur d'arborescence** d'un graphe G, notée $\omega(G)$, est la largeur minimale de toute décomposition arborescente de G.

Similairement, la **largeur de séparation** d'une décomposition arborescente $\mathcal{T}$ non-triviale (à au moins deux nœuds), notée $\omega_s(\mathcal{T})$, est le nombre maximal de sommets de G en commun entre deux nœuds voisins dans T : $\omega_s(\mathcal{T}) = \max_{\{n_1,n_2\} \in \mathcal{A}'}(|\varsigma(n_1) \cap \varsigma(n_2)|)$. La largeur de séparation d'un graphe G, notée $\omega_s(G)$, est la largeur de séparation minimale de toute décomposition arborescente de G. Il est évident que $\omega_s(\mathcal{T}) \leq \omega(\mathcal{T})$ et $\omega_s(G) \leq \omega(G)$.

La décomposition arborescente de la figure 3.1 a une largeur de 3 et une largeur de séparation de 2. Ces valeurs correspondent aussi respectivement aux largeurs d'arborescence et de séparation du graphe décomposé.

Notons que la définition de décomposition arborescente peut être naturellement étendue aux hypergraphes en considérant que la décomposition d'un hypergraphe correspond à la décomposition de son graphe primal.

3.1.2 Utilité des décompositions arborescentes

Intuitivement, une décomposition arborescente permet de donner une description arborescente d'un graphe quelconque en en extrayant des sous-parties dont les relations peuvent être décrites de façon arborescente. La largeur d'arborescence fournit quand à elle une mesure de la ressemblance du graphe à un arbre. En effet, un graphe a une largeur d'arborescence de 0 s'il ne contient aucune arête ou de 1 s'il est un arbre (ou une forêt). Pour un graphe à n sommets, la largeur d'arborescence maximale est $n - 1$ et elle est atteinte dans le cas d'un graphe complet, c'est-à-dire d'une clique de taille n.

Déterminer la largeur d'arborescence d'un graphe est un problème NP-complet

(Arnborg, Corneil et Proskurowski, 1987) [1] ; il est toutefois possible, pour une constante $k \in \mathbb{N}$ fixée, de déterminer si un graphe est de largeur d'arborescence au plus k et, le cas échéant, d'obtenir une décomposition arborescente de largeur au plus k, le tout en temps linéaire (Bodlaender, 1996). De plus, de nombreux problèmes de graphes NP-complets ou NP-difficiles peuvent être résolus en temps polynomial selon la taille des graphes sur l'ensemble des graphes de largeur d'arborescence au plus k, où $k \in \mathbb{N}$ est une borne considérée comme constante (Arnborg et Proskurowski, 1989; Bodlaender, 1988).

En effet, chaque nœud $n \in \mathcal{N}$ définit un sous-graphe $G_n(\varsigma(n), \alpha(n))$ de G comportant au plus $\omega(\mathcal{T})$ sommets. On peut alors généralement résoudre le problème considéré sur chaque sous-graphe associé à chaque nœud de la décomposition, puis s'en servir pour résoudre le problème d'origine par programmation dynamique. [2] Puisque la largeur de la décomposition est bornée par k, le nombre de variables des sous-problèmes à résoudre est borné par $k+1$.

Si l'on utilise un algorithme exponentiel selon le nombre de variables du problème traité, la résolution des sous-problèmes se fait en temps $2^{O(k)}$ et ne dépend donc pas du tout de la taille du graphe global. Si la réponse au problème sur le graphe global peut être retrouvée en temps polynomial par rapport à la taille de ce graphe en se servant des solutions des sous-problèmes, alors le problème est entièrement résolu en temps polynomial selon la taille du graphe pour tout graphe de largeur d'arborescence bornée.

Dans les cas les plus simples, il suffit de combiner des solutions compatibles des différents sous-problèmes afin d'obtenir une solution globale. Le problème NP-complet de clique maximale d'un graphe, qui consiste à déterminer la taille maximale des cliques

---

1. Plus exactement, Arnborg, Corneil et Proskurowski (1987) ont montré que le problème de déterminer pour un graphe donné le plus petit $k \in \mathbb{N}$ tel que ce graphe est un k-arbre partiel est NP-complet. Or, ce problème revient à déterminer la largeur d'arborescence du graphe van Leeuwen (1990, lemme 1.24).

2. Selon le problème, il est parfois nécessaire de résoudre un problème plus complexe sur les sous-graphes ; c'est par exemple le cas pour SAT et CSP, comme nous le verrons dans la sous-section 3.1.5.

du graphe, est un exemple extrême où la solution du problème global est très facile à trouver à partir des solutions des sous-problèmes. En effet, toute clique d'un graphe est obligatoirement incluse dans au moins un nœud d'une décomposition arborescente de ce graphe. Par conséquent, soient $G(\mathcal{S}, \mathcal{A})$ un graphe, $\mathcal{T}(T(\mathcal{N}, \mathcal{A}'), \varsigma, \alpha)$ une décomposition arborescente de G et $\{G_n\}_{n \in \mathcal{N}}$ les sous-graphes de G induits par la décomposition $\mathcal{T}$; alors la taille maximale des cliques dans G est le maximum des tailles maximales des cliques dans les sous-graphes $\{G_n\}_{n \in \mathcal{N}}$.

3.1.3 Variantes des décompositions arborescentes

La définition de décomposition arborescente de Robertson et Seymour est la plus couramment utilisée. Toutefois, il existe plusieurs autres définitions d'encapsulations arborescentes de graphes quelconques. Beaucoup sont en fait des cas particuliers de décompositions arborescentes, c'est-à-dire que toute encapsulation de graphe répondant à ces définitions sont également des décompositions arborescentes.

Parmi celles-ci, la **décomposition biconnexe** d'un graphe (Freuder, 1994) est une décomposition arborescente telle qu'à chaque nœud est associé une composante biconnexe du graphe et l'ensemble des arêtes à l'intérieur de cette composante biconnexe. Pour un graphe $G(\mathcal{S}, \mathcal{A})$, une composante biconnexe est un ensemble de sommets $B \subseteq \mathcal{S}$ tel que $\forall b \in B$, $G|_{B \setminus \{b\}}$ est connexe et B est maximal pour cette propriété. Les arêtes de la décomposition arborescente relient les paires de nœuds ayant un sommet de G en commun [3] ; l'arborescence de la décomposition est assurée par les propriétés des composantes biconnexes. Pour tout graphe G, il existe une unique décomposition biconnexe.

Une **décomposition en pivots** (*hinge decomposition*) (Gyssens, Jeavons et Cohen, 1994) est une décomposition arborescente d'un hypergraphe $H(\mathcal{S}, \mathcal{H})$ dont les nœuds correspondent aux sous-hypergraphes de H induits par les pivots minimaux de H. Un pivot de H est un ensemble de sommets $P \subseteq \mathcal{S}$ tel que pour toute composante connexe

---

3. Deux composantes biconnexes ont au plus un unique sommet en commun, appelé **point d'articulation**.

$C(\mathcal{S}_C, \mathcal{H}_C)$ de $H_{\setminus P}$, il existe une hyperarête $h \in \mathcal{H}_C$ qui contient toutes les variables en commun de P et $\mathcal{S}_C : P \cap \mathcal{S}_C \subseteq h$. Un pivot est minimal si aucun de ses sous-ensembles n'est lui-même un pivot. Les arêtes de la décomposition relient les nœuds ayant au moins un sommet de H en commun ; deux nœuds voisins ont exactement une hyperarête de H en commun, et l'arborescence de la décomposition est assurée par les propriétés des pivots. Il peut exister plusieurs décompositions en pivots pour un même hypergraphe, mais toutes ont exactement la même largeur d'arborescence.

Le ***tree clustering*** (Dechter et Pearl, 1989) est également un cas particulier de décomposition arborescente. Le graphe à décomposer est tout d'abord triangulé, c'est-à-dire que des arêtes y sont ajoutées afin de le rendre **cordal**. Un graphe est dit cordal si pour tout cycle $s_1, s_2, \ldots, s_n, s_1$ d'au moins 4 sommets, il existe une corde, c'est-à-dire une arête reliant 2 sommets non-consécutifs dans le cycle : $\exists \{i, j\} \in \{1, \ldots, n\} \,|\, \{s_i, s_j\} \in \mathcal{A}, |i - j| \geq 2$. Une fois le graphe triangulé, ses cliques maximales sont identifiées (cette identification est un problème NP-complet en général mais polynomial sur les graphes cordaux). La décomposition par *tree clustering* contient un nœud pour chaque clique maximale du graphe triangulé, associé aux sommets de cette clique et aux arêtes du graphe original reliant deux de ces sommets. Il est alors possible de relier les nœuds de la décomposition de façon arborescente tout en respectant la contrainte de connectivité. Il est évident que la décomposition arborescente obtenue est susceptible d'avoir une plus petite largeur si le nombre d'arêtes ajoutées lors de la triangulation du graphe est minimisé ; cependant, ce problème est lui-même NP-complet Yannakakis (1981). Par conséquent, Dechter et Pearl emploient un algorithme de triangulation heuristique conçu par Tarjan et Yannakakis (1984). Cet algorithme de triangulation est basé sur un ordre total sur les variables, qui peut lui-même être calculé heuristiquement.

Les ***dtrees*** (Darwiche, 2001) peuvent eux être considérés comme une présentation alternative des décompositions arborescentes, qui met plus l'emphase sur les séparateurs. Un *dtree* est un arbre enraciné strictement binaire où chaque feuille est associée à une hyperarête de l'hypergraphe décomposé. Chaque nœud interne $n \in \mathcal{N}$ représente un

séparateur du sous-hypergraphe induit par les sommets présents dans les feuilles du sous-arbre enraciné en n; ce séparateur est constitué des sommets partagées par les deux sous-arbres enracinés respectivement en n_1 et n_2, les deux fils de n. La taille d'une feuille est le nombre de variables de son hyperarête associée; la taille d'un nœud interne est le nombre de variables distinctes dans les feuilles du sous-arbre qui lui est enraciné qui font parties de son séparateur ou du séparateur d'un de ses ancêtres. La largeur d'un *dtree* est la taille maximale de ses nœuds moins 1. Le concept de dtree peut être considéré comme équivalent à celui de décomposition arborescente car pour tout $n \in \mathbb{N}$, un hypergraphe possède une décomposition arborescente de largeur n si et seulement s'il possède un dtree de même largeur (Darwiche, 2001).

Enfin, il existe également des définitions qui, au contraire, généralisent le concept de décomposition arborescente. Par exemple, la **décomposition hyperarborescente** (Gottlob, Leone et Scarcello, 2000) est similaire à la décomposition arborescente, à la différence que l'arborescence est enracinée et que pour un nœud $n \in \mathcal{N}$ et un sommet s appartenant à une hyperarête de $\alpha(n)$, il est possible d'avoir $s \notin \varsigma(n)$ si s n'est associé à aucun autre nœud du sous-arbre enraciné en n. La largeur d'une décomposition hyperarborescente est le nombre maximal d'hyperarêtes (et non pas de sommets) associées à un même nœud. Les décompositions hyperarborescentes sont elles-mêmes généralisées par les **décompositions gardées** (Cohen, Jeavons et Gyssens, 2008).

Pour toutes ces diverses définitions de décomposition, il est possible de déterminer en temps polynomial si la largeur correspondante d'un graphe ne dépasse pas une certaine borne, ainsi que de résoudre polynomialement des problèmes NP-complets sur les graphes de largeur bornée. Il existe en outre de nombreuses variations et hybridations de ces différentes définitions (par exemple Gottlob, Hutle et Wotawa, 2002; Zheng et Choueiry, 2005; Cohen et Green, 2006).

3.1.4 Décomposition arborescente d'une instance SAT ou CSP

L'intérêt principal des décompositions arborescentes, nous l'avons vu, est de permettre la résolution efficace de problèmes de graphes difficiles sur des graphes de largeur d'arborescence bornée. SAT et CSP ne sont pas des problèmes de graphes ; toutefois, les relations structurelles entre variables et clauses ou contraintes constituent une structure de graphe sous-jacente au problème. Dans le cas de CSP, pour toute instance de problème $R(\mathcal{X}, \mathcal{V}, \mathcal{C}, \delta, \pi, \tau)$, il est évident que $H_R^{\mathrm{P}}(\mathcal{X}, \{\pi(c) \,|\, c \in \mathcal{C}\})$ est un hypergraphe dans lequel chaque sommet représente une variable du problème et chaque hyperarête décrit la portée d'une contrainte. H_R^{P} est appelé l'**hypergraphe primal** du réseau de contraintes R. De même, on peut définir l'hypergraphe dual et les graphes primal et dual de R comme respectivement l'hypergraphe dual et les graphes primal et dual de H_R^{P} (Dechter, 2003, section 2.1.3). $H_R^{\mathrm{D}}(\mathcal{C}, \{\{c \in \mathcal{C} \,|\, x \in \pi(c)\} \,|\, x \in \mathcal{X}\})$, l'**hypergraphe dual** de R, comporte un sommet par contrainte de R et chaque hyperarête relie les contraintes dont la portée contient une variable $x \in \mathcal{X}$ donnée. $G_R^{\mathrm{P}}(\mathcal{X}, \{\{x_1, x_2\} \subseteq \mathcal{X} \,|\, \exists c \in \mathcal{C},\, \{x_1, x_2\} \subseteq \pi(c)\})$, le **graphe primal** de R, contient un sommet pour chaque variable et relie deux variables par une arête si la portée d'une contrainte de R les contient toutes les deux. Enfin, $G_R^{\mathrm{D}}(\mathcal{C}, \{\{c_1, c_2\} \subseteq \mathcal{C} \,|\, \pi(c_1) \cap \pi(c_2) \neq \emptyset\})$, le **graphe dual** de R, associe un sommet à chaque contrainte de R et relie deux contraintes par une arête si leurs portées comporte au moins une variable en commun.

De la même façon, on peut définir les hypergraphes et graphes primaux et duaux d'une formule propositionnelle $F(\mathcal{V}, \mathcal{C})$ en remplaçant les portées des contraintes d'une instance CSP par l'ensemble des variables présentes dans une clause. On a donc :

- $H_F^{\mathrm{P}}(\mathcal{V}, \{\nu(c) \,|\, c \in \mathcal{C}\})$;
- $H_F^{\mathrm{D}}(\mathcal{C}, \{\{c \in \mathcal{C} \,|\, v \in \nu(c)\} \,|\, v \in \mathcal{V}\})$;
- $G_F^{\mathrm{P}}(\mathcal{V}, \{\{v_1, v_2\} \subseteq \mathcal{V} \,|\, \exists c \in \mathcal{C},\, \{v_1, v_2\} \subseteq \nu(c)\})$;
- $G_F^{\mathrm{D}}(\mathcal{C}, \{\{c_1, c_2\} \subseteq \mathcal{C} \,|\, \nu(c_1) \cap \nu(c_2) \neq \emptyset\})$.

Exemple 3.1. Soit la formule propositionnelle en forme normale conjonctive $F = (a \vee b \vee e) \wedge (a \vee d) \wedge (\neg a \vee \neg e) \wedge (\neg b \vee \neg c \vee e) \wedge (\neg d \vee e)$; alors son graphe

primal, G_F^{P}, est identique au graphe représenté par la figure 3.1a.

Nous pouvons remarquer que pour toute instance I du problème SAT ou du problème CSP, H_I^{P}, H_I^{D}, G_I^{P} et G_I^{D} ont tous le même nombre de composantes connexes. Nous dirons alors que l'instance I est connexe si et seulement si ces graphes sont connexes. Pour toute formule propositionnelle $F(\mathcal{V},\mathcal{C})$ et toute composante connexe de son graphe primal $E \in G_F^{\mathrm{P}}$, la sous-formule connexe associée à E est $F_E(E, \{c \in \mathcal{C} \mid \nu(c) \subseteq E)$ et l'ensemble des sous-formules connexes de F est noté $\mathrm{Conn}(F) = \{F_E \mid E \in G_F^{\mathrm{P}}\}$. De même, pour tout réseau de contraintes $R(\mathcal{X},\mathcal{D},\mathcal{C},\delta,\pi,\tau)$ et toute composante connexe de son graphe primal $E \in G_R^{\mathrm{P}}$, le sous-réseau connexe associé à E est $R_E(\mathcal{X}_E,\mathcal{D}_E,\mathcal{C}_E,\delta_E,\pi_E,\tau_E)$, où $\mathcal{X}_E = E$, $\mathcal{D}_E = \bigcup_{x\in E}\delta(x)$, $\mathcal{C}_E = \{c \in \mathcal{C} \mid \pi(c) \subseteq E\}$, $\delta_E = \delta|_E$, $\pi_E = \pi|_{\mathcal{C}_E}$ et $\tau_E = \tau|_{\mathcal{C}_E}$, et l'ensemble des sous-réseaux connexes de R est noté $\mathrm{Conn}(R) = \{R_E \mid E \in G_R^{\mathrm{P}}\}$.

Ces représentations des problèmes CSP et SAT ne sont pas sans perte, puisqu'elles ne conservent aucune information sur les tuples autorisés par les contraintes et les instanciations interdites par les clauses, respectivement ; toutefois, elles permettent de développer des algorithmes polynomiaux pour résoudre ces problèmes sur les instances dont les graphes ou hypergraphes caractéristiques ont une largeur d'arborescence bornée, comme nous allons le voir dans les deux sous-sections suivantes.

3.1.5 Résolution par décomposition explicite

Étant donnée une décomposition arborescente d'une instance SAT ou CSP, une façon de l'utiliser pour résoudre efficacement le problème est d'adopter une stratégie de type programmation dynamique telle qu'évoquée dans la sous-section 3.1.2. Ce type d'algorithme a été décrit pour les CSP par Dechter et Pearl (1989) en utilisant le *tree clustering*, mais reste valide en utilisant une décomposition arborescente quelconque. Afin de le distinguer de la méthode exposée dans la sous-section 3.1.6, nous l'appellerons **décomposition explicite**, car il utilise la décomposition arborescente pour définir des sous-problèmes qu'il résout indépendemment avant de reconstituer une solution du problème global.

Soit $R(\mathcal{X}, \mathcal{V}, \mathcal{C}, \delta, \pi, \tau)$ un réseau de contraintes et $\mathcal{T} = (T(\mathcal{N}, \mathcal{A}), \varsigma, \alpha)$ une décomposition arborescente de son graphe primal (ou de son hypergraphe primal). Pour tout nœud $n \in \mathcal{N}$ de la décomposition, on peut définir le sous-réseau $R_n(\mathcal{X}_n, \mathcal{V}_n, \mathcal{C}_n, \delta_n, \pi_n, \tau_n)$ comme sa restriction aux variables associées à n et aux contraintes exprimées sur ses variables : $\mathcal{X}_n = \varsigma(n)$, $\mathcal{V}_n = \bigcup_{x \in \mathcal{X}_n} \delta(x)$, $\mathcal{C}_n = \{c \in \mathcal{C} \mid \pi(c) \subseteq \mathcal{X}_n\}$, $\delta_n = \delta|_{\mathcal{X}_n}$, $\pi_n = \pi|_{\mathcal{C}_n}$ et $\tau_n = \tau|_{\mathcal{C}_n}$. Soit $\eta : \mathcal{X} \to \mathcal{N}$ une fonction (non-unique) qui à toute variable associe un nœud de la décomposition qui la contient : $\forall x \in \mathcal{X}, x \in \varsigma(\eta(x))$. Comme toute clique du graphe primal (ou toute hyperarête de l'hypergraphe primal) est incluse dans au moins un nœud de la décomposition, toute contrainte est incluse dans au moins un sous-problème. Par conséquent, pour toute instanciation complète σ de $\mathcal{X}$, σ est une solution de R si et seulement si $\forall n \in \mathcal{N}$, $\sigma|_{\mathcal{X}_n}$ est une solution de R_n. De plus, grâce à la connectivité des décompositions arborescentes, si $\mathcal{N} = \{n_1, n_2, \ldots, n_{|\mathcal{N}|}\}$ et que $\sigma_1, \sigma_2, \ldots, \sigma_{|\mathcal{N}|}$ sont des solutions de $R_{n_1}, R_{n_2}, \ldots, R_{n_{|\mathcal{N}|}}$ qui coïncident sur les variables en commun des réseaux correspondant à des nœuds voisins, c'est-à-dire telles que $\forall \{n_i, n_j\} \in \mathcal{A}$, $\forall x \in \varsigma(n_i) \cap \varsigma(n_j)$, $\sigma_{n_i}(x) = \sigma_{n_j}(x)$, alors $\sigma : x \mapsto \sigma_{\eta(x)}(x)$ est une solution de R. Par conséquent, R est satisfaisable si et seulement s'il existe un tel ensemble de solutions des sous-problèmes.

L'algorithme proposé par Dechter et Pearl détermine pour chaque nœud $n \in \mathcal{N}$ l'ensemble S_n des solutions du sous-problème R_n correspondant. Ensuite, il définit un méta-réseau de contraintes binaire $M(\mathcal{X}_M, \mathcal{V}_M, \mathcal{C}_M, \delta_M, \pi_M, \tau_M)$ qui associe une variable à chaque sous-problème. Le domaine d'une variable est l'ensemble des solutions de ce sous-problème, et pour toute paire de sous-problèmes correspondant à des nœuds voisins de la décomposition arborescente, une contrainte interdit les combinaisons de solutions des sous-problèmes qui instancient des valeurs différentes à leurs variables de R en commun. On a donc $\mathcal{X}_M = \mathcal{N}$, $\mathcal{V}_M = \bigcup_{n \in \mathcal{N}} S_n$, $\mathcal{C}_M = \mathcal{A}$, $\forall n \in \mathcal{X}_M$, $\delta(n) = S_n$, $\pi = \mathrm{id}_{\mathcal{C}_M}$ et $\forall \{n_1, n_2\} \in \mathcal{C}_M$, $\tau(c) = \{(\sigma_1, \sigma_2) \in \delta(n_1) \times \delta(n_2) \mid \forall x \in \varsigma(n_1) \cap \varsigma(n_2), \sigma_1(x) = \sigma_2(x)\}$.

D'après les propriétés précédentes, les réseaux de contraintes R et M sont équisatisfaisables : si σ_R est une solution de R, alors $\forall n \in \mathcal{N}$, $\sigma_n = \sigma_R|_{\varsigma(n)}$ est une solution de R_n, donc $\sigma_M : n \mapsto \sigma_n$ est une solution de M. Inversement, si σ_M est une solution

de M et si $\forall n \in \mathcal{N}$ nous notons $\sigma_n = \sigma_M(n)$, alors $\sigma_R : x \mapsto \sigma_{\eta(x)}(x)$ est une solution de R.

Au final, la méthode de décomposition explicite utilise une décomposition arborescente du problème pour en définir des sous-problèmes résolus indépendamment, puis utilise les solutions de ces sous-problèmes pour construire un méta-problème dont la résolution fournit la réponse au problème de départ.

L'avantage de la décomposition explicite est de donner un algorithme dont la complexité temporelle n'est pas exponentielle selon la taille du problème traité, mais seulement selon $\omega(\mathcal{T})$, la largeur de la décomposition utilisée, qui peut être considérablement plus petite. En effet, puisque chaque sous-problème a au plus $\omega(\mathcal{T})$ variables, toutes ses solutions peuvent être énumérées en temps $2^{O(\omega(\mathcal{T}))}$, car le nombre de solutions d'un sous-problème est borné par $d^{\omega(\mathcal{T})}$ où d est la cardinalité maximale des domaines dans R. Le métaproblème M est un CSP binaire et arborescent, puisque son graphe primal est l'arbre $T(\mathcal{N}, \mathcal{A})$; il peut donc être résolu en temps $O(n_M d_M^2)$ où n_M et d_M sont respectivement le nombre de variables et la taille maximale des domaines de M (Dechter et Pearl, 1987). Cette dernière est égale au nombre maximal de solutions des sous-problèmes, donc $d_M \leq d^{\omega(\mathcal{T})}$. Par conséquent, le métaproblème peut également être résolu en temps $O(n_M(d^{\omega(\mathcal{T})})^2) = O(|\mathcal{N}|) \times 2^{O(\omega(\mathcal{T}))}$.

L'inconvénient principal de la résolution par décomposition explicite est qu'elle nécessite de résoudre le problème All-CSP sur les sous-problèmes, qui est potentiellement plus difficile que le problème CSP. De plus, toutes les solutions des sous-problèmes doivent être mémorisées pour constituer le métaproblème ; la complexité spatiale est donc également exponentielle selon la largeur de la décomposition. L'algorithme peut toutefois être optimisé afin que la complexité spatiale soit uniquement exponentielle selon la largeur de séparation de la décomposition (Dechter et Fattah, 2001). Cette exponentialité de la complexité spatiale rend la décomposition explicite des CSP très peu efficace en pratique, comme cela a été démontré expérimentalement par Jégou et Terrioux (2003). La décomposition explicite semble en pratique plus pertinente sur les problèmes

d'optimisation de CSP, qui sont par essence plus difficiles que le CSP simple (Koster, van Hoesel et Kolen, 2002; Sánchez, Larrosa et Meseguer, 2005)

Le schéma de décomposition explicite est également utilisable pour résoudre le problème SAT, bien que moins intuitif puisque le métaproblème n'est pas un problème SAT, mais un problème CSP. Là encore, le fait de devoir trouver et mémoriser toutes les solutions des sous-problèmes est un frein important à son utilisation en pratique. Cette stratégie a donc été très peu étudiée dans le cadre de SAT, essentiellement par Amir et McIlraith (2005) de façon théorique uniquement, et plus expérimentalement par Singer et Monnet (2008), qui ont proposé une résolution parallèle de SAT, basée sur la décomposition explicite, et ont également identifié l'explosion de l'espace mémoire nécessaire comme le principal obstacle à l'efficacité de leur implémentation.

Si nous reprenons la formule $F = (a \vee b \vee e) \wedge (a \vee d) \wedge (\neg a \vee \neg e) \wedge (\neg b \vee \neg c \vee e) \wedge (\neg d \vee e)$ définie dans l'exemple 3.1 et la décomposition arborescente de son graphe primal illustrée par la figure 3.1b, la décomposition explicite du problème SAT sur la formule F consiste à rechercher tous les modèles des sous-formules $F_1 = (a \vee d) \wedge (\neg a \vee \neg e) \wedge (\neg d \vee e)$, $F_2 = (a \vee b \vee e) \wedge (\neg a \vee \neg e)$ et $F_3 = (\neg b \vee \neg c \vee e)$, puis à vérifier s'il existe un ensemble de modèles compatibles sur leurs variables communes. F est bien satisfaisable car $\sigma_1 = \{a, \neg d, \neg e\}$, $\sigma_2 = \{a, \neg b, \neg e\}$ et $\sigma_3 = \{\neg b, c, \neg e\}$ sont des modèles respectivement de F_1, F_2 et F_3 qui affectent les mêmes valeurs aux variables qu'ils partagent.

3.1.6 Résolution par décomposition implicite

Si la résolution des problèmes SAT et CSP par décomposition explicite a une complexité temporelle uniquement exponentielle selon la largeur de la décomposition utilisée, elle rend également la complexité spatiale exponentielle selon cette largeur (ou selon la largeur de séparation). Cette complexité spatiale est en pratique un handicap important par rapport à la complexité polynomiale des algorithmes de recherche en profondeur tels que DPLL et CDCL pour SAT ou BT et CBJ pour CSP. Différentes méthodes, que nous regrouperons sous le nom de **décompositions implicites**, ont donc

été conçues afin de combiner les avantages des deux approches, c'est-à-dire de réduire le facteur exponentiel de la complexité temporelle tout en conservant une complexité spatiale polynomiale. Concrètement, lors de la résolution d'un problème par recherche en profondeur, les stratégies de décompositions implicites utilisent une décomposition arborescente de la structure du problème pour influer sur l'heuristique de choix des décisions.

Soit $F(\mathcal{V},\mathcal{C})$ un formule propositionnelle non-connexe et $\mathrm{Conn}(F)$ l'ensemble de ses sous-formules connexes. Il est évident que F est satisfaisable si et seulement si toutes ses sous-formules connexes sont satisfaisables, et que leur satisfaisabilité peut être testée indépendemment l'une de l'autre (Biere et Sinz, 2006). La satisfaisabilité de F peut ainsi être résolue en temps exponentiel selon le nombre maximal de variables des sous-formules connexes.

Une propriété similaire existe dans le cas d'une formule $F(\mathcal{V},\mathcal{C})$ quelconque mais qui peut être considérée comme déconnectée en tenant compte de l'instanciation courante au cours d'une recherche en profondeur. Pour toute instanciation partielle σ sur $\mathcal{V}$, nous pouvons définir le **sous-problème résiduel** de F et σ par $\mathrm{res}(F,\sigma) = F'(\mathcal{V}',\mathcal{C}')$ où $\mathcal{V}' = \mathcal{V} \setminus \mathcal{D}(\sigma)$ et $\mathcal{C}' = \{c \setminus \{\neg l \mid l \in \sigma\} \mid c \in \mathcal{C}, c \cap \sigma = \emptyset\}$. Le sous-problème résiduel $\mathrm{res}(F,\sigma)$ contient donc les variables qui ne sont pas instanciées par σ, ainsi que les clauses qui ne contiennent aucun littéral de σ et dans lesquelles on a retiré les littéraux opposés aux littéraux de σ. De plus, $\mathrm{res}(F,\sigma)$ est satisfaisable si et seulement si il existe un modèle de F qui étend σ. Par conséquent, pour tout sous-ensemble de variables $V \subseteq \mathcal{V}$, F est satisfaisable si et seulement s'il existe une instanciation complète sur V, σ_V, qui ne viole aucune clause de $\mathcal{C}$ et telle que $\mathrm{res}(F,\sigma_V)$ est satisfaisable.

Pour toute instanciation partielle σ sur $\mathcal{V}$, le graphe primal de $\mathrm{res}(F,\sigma)$ est un sous-graphe du graphe primal de F. Par conséquent, pour tout séparateur $S \subseteq \mathcal{V}$ de G_F^{P}, le graphe primal de F, et toute instanciation complète σ_S de S, le problème résiduel $\mathrm{res}(F,\sigma_S)$ est non-connexe. De plus, tout sous-problème connexe de $\mathrm{res}(F,\sigma_S)$ a ses variables incluses dans une composante connexe de $(G_F^{\mathrm{P}})_{\setminus S}$ (chaque composante

connexe de ce graphe correspond à une ou plusieurs sous-formules de $\mathrm{res}(F, \sigma_S)$). Soit $\{\mathrm{res}(F, \sigma_S)|_E \mid E \in \mathrm{Conn}((G_F^{\mathrm{P}})_{\setminus S})\}$ l'ensemble des sous-problèmes induits par le séparateur S (avec l'instanciation σ_S) ; il existe un modèle de F qui étend σ_S si et seulement si tous les sous-problèmes induits par S avec σ_S sont satisfaisables.

Une stratégie pour résoudre la satisfaisabilité d'une formule F consiste donc à identifier un séparateur du graphe primal, puis, pour toute instanciation consistante de ce séparateur, à résoudre indépendamment les sous-problèmes correspondant aux composantes connexes du graphe séparé. F est satisfaisable si et seulement s'il existe une instanciation de S telle que tous les sous-problèmes sont satisfaisables. Comme il existe au plus $2^{|S|}$ instanciations consistantes de S et, pour toute instanciation de S σ_S et tout $E \in \mathrm{Conn}((G_F^{\mathrm{P}})_{\setminus S})$, le sous-problème $\mathrm{res}(F, \sigma_S)|_E$ se résout en temps $2^{O(|E|)}$, F est résolu avec cette méthode en temps $2^{O(|S|+\max_{E \in \mathrm{Conn}((G_F^{\mathrm{P}})_{\setminus S})} |E|)}$. Cette stratégie a été décrite et implémentée par Park et van Gelder (1996) et Durairaj et Kalla (2004). Plus précisément, ces auteurs ont considéré une bipartition $\{E_1, E_2\}$ non-connectée de $\mathcal{V} \setminus S$. Chaque ensemble de cette bipartition contient donc une ou plusieurs composantes connexes de $(G_F^{\mathrm{P}})_{\setminus S}$.

Or, une telle stratégie est équivalente à utiliser un algorithme de recherche en profondeur à saut arrière sur la totalité de l'instance, avec certaines contraintes sur l'heuristique de choix des décisions. Plus précisément, définissons un **ordonnancement partiel** sur les variables $\theta = (E_1, E_2, E_3, \ldots, E_n)$ comme une partition ordonnée des variables telle que, lors de toute décision au cours de la recherche, l'algorithme doit choisir une variable dans le premier ensemble qui n'est pas encore totalement instancié. Si S est un séparateur de la formule F, une recherche en profondeur à saut arrière simule la résolution indépendante des sous-problèmes si elle est munie d'un ordonnancement partiel sur les variables dont le premier ensemble est S et tous les autres ensembles sont les composantes connexes de $(G_F^{\mathrm{P}})_{\setminus S}$ dans un ordre quelconque. En effet, si un conflit intervient après une décision dans l'une des composantes connexes, le retour arrière revient soit au niveau d'une décision de cette même composante, soit au niveau d'une décision de S. Les différents problèmes associés aux composantes connexes ne

peuvent donc aucunement interagir. Li et van Beek (2004) ont utilisé ce procédé en considérant des séparateurs de l'hypergraphe primal engendrant une partition en un nombre quelconque de composantes connexes.

Notons que pour un séparateur S séparant le graphe primal d'une formule F en différentes composantes $\text{Conn}((G_F^{\text{P}})_{\setminus S}) = \{E_1, E_2, \ldots, E_p\}$, on peut construire une décomposition arborescente $\mathcal{T}(T(\mathcal{N}, \mathcal{A}), \varsigma, \alpha)$ du graphe primal de F, où $\mathcal{N} = \{n_0, n_1, n_2, \ldots, n_p\}$, $\mathcal{A} = \{\{n_0, n_i\} \mid i \in \{1, \ldots, p\}\}$, $\varsigma(n_0) = S$, $\forall i \in \{1, \ldots, p\}$, $\varsigma(n_i) = S \cup E_i$ et $\forall i \in \{0, \ldots, p\}$, $\alpha(n_i) = \{c \in C \mid \nu(c) \subseteq \varsigma(n_i)\}$. Cette décomposition arborescente a $p+1$ nœuds et une largeur de $|S| + \max_{i \in \{1,\ldots,p\}}(|E_p|)$; notre méthode de résolution de F par séparation a donc une complexité exponentielle selon la largeur de la décomposition arborescente sous-jacente. Cependant, la largeur d'arborescence minimale d'une telle décomposition est $\frac{|V|}{p}$.

Cette propriété est généralisable dans le cas d'une décomposition arborescente quelconque : si $(n_1, n_2, \ldots, n_{|\mathcal{N}|})$ est un parcours préfixe des nœuds de l'arborescence (à partir d'une racine arbitraire), alors une recherche à saut arrière munie de l'ordonnancement partiel $(\varsigma(n_1), \varsigma(n_2) \setminus \varsigma(n_1), \ldots, \varsigma(n_{|\mathcal{N}|}) \setminus \bigcup_{i=1}^{|\mathcal{N}|-1} \varsigma(n_i))$ effectue récursivement une résolution indépendante des sous-problèmes résiduels séparés par l'instanciation des nœuds internes de la décomposition. Cette propriété a été montrée par Huang et Darwiche (2003) dans le cas d'un ordonnancement partiel de variables induit par le parcours préfixe des nœuds d'un *dtree*[4], mais elle est également valable dans le cas de décompositions arborescentes. Nous parlerons de décomposition implicite d'un problème SAT pour désigner cette tactique d'utilisation d'une décomposition arborescente pour influer sur la résolution globale du problème par un algorithme de recherche, par opposition à la décomposition explicite qui résout séparément les sous-problèmes associés aux nœuds de la décomposition. Notons que la décomposition implicite peut s'appliquer similairement au problème CSP.

---

4. On considère comme ensemble de variables associé à un nœud d'un *dtree* son séparateur, pour un nœud interne, ou les variables présentes dans sa clause, pour une feuille.

Exemple 3.2. Soit la formule $F = (a \vee b \vee e) \wedge (a \vee d) \wedge (\neg a \vee \neg e) \wedge (\neg b \vee \neg c \vee e) \wedge (\neg d \vee e)$ définie dans l'exemple 3.1 et la décomposition arborescente de son graphe primal illustrée par la figure 3.1b, que nous supposerons enracinée au nœud 2 ; il existe deux parcours préfixes distincts de cette arborescence : $(2, 1, 3)$ et $(2, 3, 1)$. Si nous utilisons le premier, la stratégie de décomposition implicite du problème SAT revient à instancier tout d'abord les variables a, b et e dans un ordre quelconque, puis la variable d, et enfin la variable c. $\{a, b, e\}$ est bien un séparateur de l'instance car aucune clause de la formule ne contient à la fois les variables c et d; le problème devient donc forcément non-connexe après l'instanciation de ces trois premières variables.

Durairaj et Kalla (2004)ont implémenté cette stratégie de décomposition implicite sans s'attacher à sa complexité mais en justifiant son efficacité par des résultats expérimentaux (ils se restreignent toutefois à des listes, c'est-à-dire à des décompositions arborescentes où chaque nœud a au plus deux voisins). Habbas, Amroun et Singer (2011) ont eux prouvé que leur algorithme de décomposition implicite à partir d'une décomposition hyperarborescente d'un problème CSP est en temps exponentiel à la fois selon la largeur de l'hyperarborescence et selon son nombre de nœuds, alors que sa complexité spatiale est linéaire.

D'autres stratégies ont rajouté certaines conditions afin de rendre la complexité temporelle exponentielle en temps uniquement selon la largeur de l'arborescence. Par exemple, Huang et Darwiche (2003) ont montré que l'utilisation d'un *dtree* équilibré, c'est-à-dire d'une profondeur en $O(\log(|\mathcal{N}|))$, suffit à obtenir une complexité temporelle de $O(|\mathcal{N}|^{\omega(\mathcal{T})+1})$ (par définition, le nombre de nœuds d'un *dtree* d'une instance SAT ou CSP est polynomial selon son nombre de clauses ou de contraintes). À partir d'un *dtree* quelconque de largeur ω, il est possible d'obtenir un *dtree* équilibré de largeur au plus $\omega + 1$ en temps $O(|\mathcal{N}| \log(|\mathcal{N}|))$. Par conséquent, la restriction de l'algorithme à un *dtree* équilibré ne perd pas de généralité.

L'algorithme BTD (*Backtracking with Decompositions*, Jégou et Terrioux, 2003) résout le problème CSP par décomposition implicite à l'aide de décompositions arbo-

rescentes. Pour tout paire de nœuds n_1 et n_2 tels que n_2 est le fils de n_1 et pour toute instanciation de leurs variables communes $\varsigma(n_1) \cap \varsigma(n_2)$, BTD retient si cette instanciation peut ou non être étendue sur les variables contenues dans le sous-arbre enraciné en n_2. Tout sous-problème résiduel associé à une instanciation donnée du séparateur est donc résolu au plus une fois au cours de la recherche. Cette mémorisation permet de rendre BTD exponentiel en temps selon la largeur de la décomposition uniquement ; elle permet également d'utiliser un algorithme de recherche à retour arrière (au lieu d'un algorithme à saut arrière) tout en conservant cette complexité. Cependant, la mémorisation des solutions aux sous-problèmes induit une complexité spatiale exponentielle selon la largeur de séparation, identique à la complexité spatiale du *tree clustering*. Les résultats expérimentaux obtenus par Jégou et Terrioux indiquent que cette complexité spatiale handicape cependant moins les performances de BTD que celles du *tree clustering*. Le BTD a été adapté au problème SAT (Habet, Paris et Terrioux, 2009) ainsi qu'à diverses variantes d'optimisation du problème CSP (Terrioux et Jégou, 2003; Jégou et Terrioux, 2004; de Givry, Schiex et Verfaillie, 2006). Il a également été modifié afin d'utiliser des décompositions hyperarborescentes (Jégou, Ndiaye et Terrioux, 2009).

Bjesse et al. (2004) ont remarqué que lors de tout conflit dans une décomposition implicite, il existe une clause de conflit dont toutes les variables appartiennent à un même nœud de la décomposition. En utilisant uniquement des clauses de conflits de cette forme, le nombre maximal de clauses de conflits distinctes est borné exponentiellement par la largeur de la décomposition. En supposant qu'aucune clause apprise n'est oubliée lors de la recherche, le nombre de conflits distincts, et donc la complexité en temps de la résolution décomposition implicite, sont également bornées exponentiellement par la largeur de la décomposition. La conservation exhaustive des clauses de conflits implique la même borne pour la complexité spatiale. Cette stratégie permet en outre d'assouplir l'ordonnancement des variables sans modifier les complexités temporelle et spatiale : à chaque décision, l'algorithme peut choisir une variable dans tout voisin d'un nœud déjà totalement instancié ; il est donc possible de choisir une variable dans un nouveau nœud sans avoir terminé d'instancier les variables du nœud de la décision précédente.

Notons que si l'efficacité des décompositions implicites pour SAT a été démontrée expérimentalement par différentes études, ces résultats ont été obtenus sur des formules relativement petites et une telle efficacité est difficile à généraliser pour des formules plus grandes. Le chapitre 4 est consacré à cette problématique.

3.1.7 Minimisation de la destructivité des sauts arrière par décomposition implicite

Nous avons vu que l'utilisation de décompositions arborescentes pour la résolution du problème SAT est principalement motivée par deux aspects liés entre eux : d'une part l'idée intuitive de résoudre séparément des sous-problèmes indépendants, et d'autre part la possible réduction de complexité temporelle qui en découle. Les décompositions implicites, qui sont un cas particulier de recherche en profondeur avec saut arrière, constituent également une stratégie de minimisation de la destructivité des sauts arrière.

En effet, l'objectif des décompositions implicites est de provoquer par l'instanciation de certaines variables la séparation du problème résiduel en différents sous-problèmes indépendants. De plus, l'ordre partiel sur les variables implique qu'un saut arrière à l'intérieur d'un sous-problème (qui retourne à un niveau dont la décision appartient au même sous-problème que la décision du niveau de conflit) peut uniquement défaire des variables de ce sous-problème particulier et ne peut affecter les instanciations des autres sous-problèmes. La contrainte sur les heuristiques de décision a donc pour conséquence de restreindre la destructivité des sauts arrière qui restent à l'intérieur d'un sous-problème donné.

Notons toutefois que si cette destructivité est limitée, elle n'est toutefois pas entièrement évitée. Tout saut arrière à l'intérieur d'un sous-problème peut désinstancier des niveaux de décision sans rapport avec le conflit s'ils font partie du même sous-problème. D'autre part, dans le cas du problème SAT, les décompositions arborescentes ne peuvent capturer dans le détail la connectivité d'un sous-problème résiduel pour une instanciation donnée. En effet, pour une instanciation σ_V d'un ensemble de variables V

d'une formule $F(\mathcal{V}, \mathcal{C})$, le graphe primal de $\text{res}(F, \sigma_V)$ est un sous-graphe couvrant de $(G_F^{\text{P}})_{\setminus V}$: il contient le même ensemble de variables, mais une arête $\{v_1, v_2\}$ de $(G_F^{\text{P}})_{\setminus V}$ peut ne pas appartenir à $\text{res}(F, \sigma_V)$ si toute clause $c \in \mathcal{C}$ telle que $\{v_1, v_2\} \subseteq \nu(c)$ est satisfaite par σ_V. Le graphe $(G_F^{\text{P}})_{\setminus V}$ représente donc une borne supérieure de la connectivité du graphe résiduel de F et σ_V mais ne peut assurer de représenter sa connectivité exacte. Par conséquent, une décomposition arborescente ne peut permettre de représenter la connectivité exacte du sous-problème résiduel pour une instanciation donnée. Pour toute formule F, tout séparateur S de F et toute instanciation complète σ_S de S, chaque sous-problème induit par le séparateur S avec σ_S peut contenir plusieurs composantes connexes du problème résiduel $\text{res}(F, \sigma_S)$. La recherche à saut arrière peut donc, même avec une décomposition implicite, détruite des niveaux de décision d'une composante totalement indépendante de la composante où se déroule le conflit, si ces deux composantes appartiennent au même sous-problème induit par les séparateurs successifs. Pour résoudre ce problème, Li et van Beek (2004) ont proposé de résoudre ce problème à l'aide d'une décomposition dynamique, qui consiste à calculer des séparateurs de l'instance en cours de recherche en tenant compte de la connectivité exacte du graphe résiduel ; cependant, leurs résultats expérimentaux indiquent que ce calcul très fréquent de séparateurs provoque un surcoût en temps d'exécution trop important.

En résumé, les décompositions implicites limitent, sans totalement les éliminer, les destructions de niveaux de décision par des sauts arrière à l'intérieur d'une autre composante connexe du sous-problème résiduel courant.

3.2 La détection dynamique des sous-problèmes résiduels connexes

Les techniques de décomposition implicite des problèmes SAT et CSP ont pour objectif de provoquer la déconnexion du sous-problème résiduel au cours d'une recherche en profondeur, puis de résoudre indépendamment les composantes déconnectées de ce sous-problème. Il existe d'autres méthodes qui exploitent également la déconnexion des sous-problèmes résiduels, mais sans chercher à provoquer cette déconnexion. Au lieu de forcer la séparation du sous-problème en contraignant l'heuristique de choix des variables,

la connectivité du sous-problème résiduel est périodiquement détectée par un parcours de son graphe primal. Si ce graphe est non-connexe, les différentes composantes du sous-graphe résiduel sont alors résolues indépendamment en contraignant l'heuristique de décision à instancier successivement toutes les variables d'une même composante connexe.

Cette stratégie de détection dynamique des composantes connexes a plusieurs avantages par rapport aux décompositions implicites. Tout d'abord, elle ne nécessite pas la construction préalable d'une décomposition arborescente du problème ; de plus, elle n'impose aucune contrainte sur l'ordre de décision des variables, hormis pour assurer la résolution indépendante des sous-problèmes. Elle permet également de connaître la connectivité exacte du sous-problème résiduel en tenant compte de l'effet des clauses satisfaites par l'instanciation courante, alors que les décomposition implicites ne fournissent qu'une borne supérieure à la connectivité des sous-problèmes. Enfin, si la détection de la connectivité est systématique (avant chaque décision), l'algorithme garantit qu'aucun conflit dans une composante connexe ne peut défaire d'instanciations dans une autre composante.

En contrepartie, la détection des sous-problèmes connexes n'a aucune utilité tant que le sous-problème résiduel reste connexe et ne permet pas d'influer sur cette connectivité. Elle ne permet donc pas d'améliorer la borne de complexité de l'algorithme, contrairement aux décompositions implicites. De plus, la détection de connectivité peut être effectuée avec une très bonne efficacité théorique, mais son temps d'exécution pratique est non-négligeable, particulièrement si elle est effectuée systématiquement. Si la détection n'est pas systématique, alors l'algorithme peut alterner des décisions entre plusieurs composantes connexes du sous-problème résiduel et n'empêche donc pas totalement les sauts arrière dans une composante connexe de détruire des instanciations d'une autre composante.

La détection dynamique des composantes a été relativement peu implémentée dans le cadre du problème SAT ; Biere et Sinz (2006) en ont implémenté une version qui détecte

la connectivité uniquement avant le début de la recherche ou lorsque toutes les décisions sont défaites (après un redémarrage ou un saut arrière dont le niveau d'assertion est le pseudo-niveau 0). Leurs résultats expérimentaux montrent que cette détection est trop restreinte pour avoir un impact significatif sur le temps de résolution du problème SAT. La détection dynamique des composantes a été étudiée plus fréquemment dans le cadre de problème plus complexes que SAT, par exemple #SAT (Bayardo Jr. et Pehoushek, 2000; Bacchus, Dalmao et Pitassi, 2003; Sang et al., 2004) et QBF (*quantified boolean formula*), le problème de satisfaisabilité des formules booléennes dont les variables peuvent être quantifiées existentiellement ou universellement (Samulowitz et Bacchus, 2007).

Mentionnons aussi l'heuristique proposée par Marques-Silva et Oliveira (1997) qui restreint le choix de la prochaine décision à des variables partageant des clauses avec des variables instanciées dans le niveau de décision précédent, s'il existe de telles variables. Cette heuristique a pour effet de garantir la résolution indépendante des composantes connexes du sous-problème résiduel sans avoir à détecter explicitement sa connectivité. En contrepartie, elle impose, comme les décompositions implicites, des restrictions à la stratégie de décision de l'algorithme.

3.3 La sauvegarde de phase

La sauvegarde de phase (*phase saving*, Pipatsrisawat et Darwiche, 2007) est une heuristique légère dont le but premier est de contrer la destructivité des sauts arrière dans l'algorithme CDCL en permettant de récupérer les instanciations détruites. L'accent est mis sur les instanciations sans rapport avec le conflit, notamment les instanciations appartenant à une composante connexe distincte de la composante du conflit : la sauvegarde de phase est présentée comme une variante plus heuristique de la détection de composantes connexes du sous-problème résiduel. Toutefois, elle permet de récupérer l'instanciation de toutes les variables défaites par un saut arrière antérieur, qu'elles aient eu ou non un rapport direct avec le conflit correspondant.

Concrètement, la sauvegarde de phase conserve pour toute variable la dernière

valeur booléenne qui lui a été assignée, que cela soit par une décision ou par propagation. Lors d'une décision, la variable choisie est réinstanciée à la même valeur booléenne que lors de son instanciation précédente, ou à une valeur arbitraire si elle est instanciée pour la première fois.

Si une variable de décision précédemment défaite est à nouveau choisie par l'heuristique de décision, elle sera donc à nouveau instanciée à la même valeur, et peut entraîner à nouveau la même chaîne de propagation que précédemment (ou une partie de cette chaîne, selon les autres modifications de l'instanciation partielle entre les deux décisions). Dans le cas d'une décision appartenant à une composante connexe distincte de celle du conflit qui a causé son effacement, le niveau de décision au complet peut être reconstitué si le reste de la composante connexe n'a pas été affecté, ce qui permet alors de totalement réparer l'effet du saut arrière sur l'instanciation de cette composante.

En résumé, la sauvegarde de phase n'empêche pas les sauts arrière de détruire des instanciations sans relation avec le conflit, mais peut aider à reconstruire une partie des instanciations détruites. Cette stratégie n'évite pas le surcoût nécessaire à la destruction, puis surtout à la reconstruction des instanciations, puisque chaque décision rétablie entraîne une nouvelle phase de propagations unitaires. Ce surcoût est cependant compensé par le faible coût de l'heuristique en elle-même ; elle se résume essentiellement à retenir une valeur booléenne pour chaque variable du problème, ce qui est évidemment bien moins lourd que la génération d'une décomposition arborescente du problème, ou même que la détection fréquente de sa connectivité.

La sauvegarde de phase peut cependant avoir des effets néfastes sur la recherche, puisqu'elle retient la polarité de toutes les variables désinstanciées, qu'elles soient ou non impliquées dans le conflit. Dans le premier cas, leur réinstanciation peut provoquer rapidement d'autres conflits non-identiques mais similaires. Les auteurs de l'heuristique ont d'ailleurs constaté que si la sauvegarde de phase améliore souvent les performances du CDCL, elle peut également les dégrader dans certains cas (Pipatsrisawat et Darwiche, 2007), qui pourraient correspondre à des réinstanciations fréquentes de variables impli-

quées dans les précédents conflits. La sauvegarde de phase est toutefois intégrée dans de nombreux solveurs récents, comme GLUCOSE (Audemard et Simon, 2009), en raison entre autres de son impact sur les redémarrages (voir sous-section 2.5.6).

3.4 Les algorithmes de recherche à saut arrière partiel pour CSP

Sous l'appellation d'algorithmes de recherche à saut arrière partiel, nous regrouperons une famille d'algorithmes de résolution de CSP qui, comme CBJ, effectuent un retour arrière intelligent, c'est-à-dire qui empêchent de redéclencher le même conflit dans la branche de recherche courante, tout en réduisant la destructivité des sauts arrière par rapport à CBJ.

Les sous-section 3.4.1 à 3.4.3 présentent les algorithmes à retour arrière dynamique, retour arrière dynamique généralisé et retour arrière à ordre partiel respectivement. La sous-section 3.4.4 introduit quelques généralisations supplémentaires de ces algorithmes, la sous-section 3.4.5 décrit l'ajout de la maintenance de l'arc consistance dans l'algorithme à retour arrière dynamique, et la sous-section 3.4.6 discute de la destructivité des sauts arrière dans ces différents algorithmes.

3.4.1 Le retour arrière dynamique

Pour résoudre un conflit, CBJ, l'algorithme classique de recherche en profondeur avec saut arrière sur les CSP, désinstancie la variable coupable et toutes les autres variables instanciées après cette variable coupable, puis rétablit toutes les valeurs de domaines éliminées dont la raison d'élimination contient une des variables défaites. Comme nous l'avons constaté précédemment, aucune des variables désinstanciées n'a de rapport direct avec le conflit, hormis la variable coupable. Ginsberg (1993) a remarqué que, dans un algorithme CBJ pur (sans inférence entre deux étapes de recherche), il est possible de modifier l'ordre des variables déjà instanciées sans rendre la recherche inconsistante : après avoir modifié l'ordre, il suffit pour toute variable instanciée $x \in \mathcal{D}(\sigma)$ de rétablir les valeurs éliminées dont la raison d'élimination contient une autre variable instanciée

$x' \in \mathcal{D}(\sigma)$ qui est désormais considérée comme instanciée après la variable x.

L'algorithme de retour arrière dynamique (*dynamic backtracking* ou DBT) utilise cette propriété pour réduire le nombre d'instanciations défaites et de valeurs rétablies lors d'un saut arrière. Lors de tout conflit, avant d'effectuer le saut arrière, la variable coupable est déplacée afin d'être considérée comme la dernière variable instanciée. Cette stratégie revient à résoudre le conflit en désinstanciant uniquement la variable coupable et en rétablissant uniquement les valeurs éliminées dont la raison d'élimination contient la variable coupable. Il est évident que tout retour arrière de DBT défait autant ou moins de variables et rétablit autant ou moins de valeurs qu'un saut arrière de CBJ dans le même contexte. DBT est donc une variante de CBJ qui réduit considérablement la destructivité du saut arrière. En particulier, il est dit additif sur les unions disjoints : pour un problème (ou un sous-problème résiduel) CSP non-connexe, le temps d'exécution de DBT est exponentiel selon le nombre de variables de la plus grande composante connexe (McAllester, 1993). Comme le CBJ, DBT est totalement correct et sa complexité spatiale est polynomiale (Ginsberg, 1993).

3.4.2 Le retour arrière dynamique généralisé

McAllester (1993) a proposé l'algorithme de retour arrière dynamique généralisé (que nous noterons GDBT pour *generalized dynamic backtracking*), qui est à mi-chemin entre DBT et CBJ. D'une part, DBT défait uniquement la variable coupable et rétablit les valeurs précédemment éliminées dont la raison d'élimination contient cette variable. D'autre part, CBJ défait la variable coupable et toutes les variables instanciées après la variable coupable, et rétablit toutes les valeurs dont la raison d'élimination contient une de ces variables désinstanciées.

GDBT défait toutes les variables instanciées à partir de la variable coupable incluse, comme CBJ, mais rétablit uniquement les valeurs éliminées dont la raison d'élimination contient la variable coupable, comme BTD. Par conséquent, si $x_i \mapsto v_i$ est une instanciation défaite lors d'un retour arrière de GDBT, si x_i n'est pas la variable

coupable et v_j est une valeur éliminée de la variable x_j telle que $x_i \mapsto v_i$ fait partie de sa raison d'élimination, alors v_j n'est pas rétablie (sauf si une instanciation de la variable coupable fait aussi partie de sa raison d'élimination). Cependant, v_j sera rétablie ultérieurement si x_i est réinstanciée à une valeur différente de v_i ou si x_j est choisie pour être instanciée alors que x_i n'est pas instanciée.

Cette conservation des valeurs éliminées par GDBT permet en quelque sorte de ne pas avoir à redécouvrir les inférences correspondantes si les variables défaites par le saut arrière sont réinstanciées avec les mêmes valeurs. Cependant, l'algorithme n'impose aucune contrainte sur l'ordre des variables à instancier après un saut arrière, ni sur les valeurs à choisir. En comparaison, le DBT classique est un cas particulier de GDBT qui impose, après chaque saut arrière, de réinstancier immédiatement toutes les variables défaites (hormis la variable coupable) dans le même ordre et avec les mêmes valeurs. Le GDBT a donc l'avantage de laisser plus de libertés ; en contrepartie, il n'est pas additif sur les unions disjoints, mais reste totalement correct et polynomial en espace (McAllester, 1993).

La conservation de valeurs éliminées alors que certaines de leurs raisons sont désinstanciées est plus facile à comprendre si l'on considère le GDBT non pas comme un algorithme qui construit progressivement une instanciation partielle, mais comme un algorithme qui manipule en tout temps une instanciation complète qu'il modifie tant qu'elle viole une contrainte (c'est d'ailleurs de cette façon que l'algorithme est présenté). Pour représenter sa progression, l'algorithme sépare les variables entre variables passées et variables futures. Au début de la recherche, toutes les variables du problèmes sont des variables futures et sont ordonnées de façon arbitraire. Tout au long de la recherche, toutes les variables passées sont positionnées avant les variables futures, et la raison d'élimination d'une valeur éliminée peut uniquement contenir des instanciations de variables antérieures. L'ordre entre les variables futures peut être modifié en cours de recherche, mais pas l'ordre entre les variables passées.

Pour effectuer une décision, GDBT sélectionne une variable future x qu'il déplace

avant toutes les autres variables futures, puis rétablit toutes les valeurs éliminées de x dont la raison d'élimination contient une autre variable future. L'instanciation de x peut être modifiée, auquel cas toutes les valeurs éliminées des autres variables futures dont la raison d'élimination contient l'ancienne instanciation de x sont rétablies. x est ensuite considérée comme une variable passée. Un conflit est déclenché dès qu'une contrainte $c \in \mathcal{C}$ est violée par l'instanciation courante σ et que toutes les variables de sa portée sont des variables passées. La variable coupable x est la variable de sa portée la plus récente ; soit v la valeur associée à x. Pour résoudre le conflit, GDBT considère x et toutes les variables ultérieures comme des variables futures, modifie l'instanciation de x et élimine la valeur v du domaine de x avec comme raison d'élimination $\sigma|_{\pi(c)\setminus\{x\}}$, puis rétablit les valeurs de domaines des variables futures dont la raison d'élimination implique x. Un conflit peut aussi être déclenché lorsque le domaine d'une variable $x \in \mathcal{X}$ devient vide. Un tel conflit est résolu de la même façon en considérant le *nogood* obtenu par l'union des raisons d'élimination des valeurs de x comme un tuple interdit pour une contrainte de portée correspondante.

Cette description de GDBT qui manipule une instanciation complète en tout temps est équivalente à la description précédente de construction progressive d'une instanciation partielle.

3.4.3 Le retour arrière à ordre partiel

Le retour arrière à ordre partiel (POB ou *partial order backtracking*, McAllester, 1993; Ginsberg et McAllester, 1994) est une variante de GDBT qui permet de choisir la variable coupable parmi plusieurs possibilités ; en contrepartie, des contraintes d'ordre entre les variables du problème sont ajoutées au fil de la recherche.

Si nous continuons à utiliser la description du GDBT comme une manipulation d'une instanciation en tout temps complète, la violation d'une contrainte $c \in \mathcal{C}$ par une instanciation σ y est résolue en identifiant la variable coupable x, qui est la plus grande variable dans la portée de c, puis en éliminant la valeur $\sigma(x)$ du domaine de x avec

$\sigma|_{\pi(c)\setminus\{x\}}$ comme raison d'élimination.

L'algorithme POB propose d'introduire une liberté supplémentaire en ordonnant partiellement les variables instanciées du problème. Par conséquent, toute variable $x \in \pi(c)$ maximale dans $\pi(c)$ (c'est-à-dire pour laquelle il n'existe pas une autre variable $x' \in \pi(c)$ telle que $x \prec x'$) peut être choisie comme variable coupable. Cet ordre partiel sur les variables remplace la notion de variables passées et futures dans GDBT.

POB démarre sans aucune contrainte de précédence entre les variables du problème ; pour la première contrainte violée détectée, on peut donc choisir arbitrairement la variable coupable dans la portée de cette contrainte. Chaque élimination de valeur ajoute une contrainte implicite sur l'ordre entre les variables : la variable dont la valeur est éliminée est postérieure à toutes les variables présentes dans la raison de cette élimination. Plus formellement, pour toute variable $x \in \mathcal{X}$ et toute valeur éliminée de son domaine $v \in \delta(x)$, $\forall x' \in \mathcal{D}(\rho(v))$, $x' \prec x$.

Le choix de la variable coupable lors d'un conflit est donc contraint par cet ordre partiel induit par les raisons d'élimination des valeurs présentement éliminées. Toutefois, cet ordre partiel ne suffit pas à assurer la terminaison de l'algorithme. Pour rétablir la terminaison, POB maintient également un ensemble S de conditions de sûretés, qui sont des contraintes de précédence supplémentaires entre variables. Lors d'un conflit, le choix de la variable coupable doit donc respecter à la fois les contraintes implicites de toutes les raisons d'élimination actives et les contraintes explicites représentées par S.

S est initialement vide. Lorsqu'une valeur v d'une variable x est éliminée, pour toute variable y que x doit précéder d'après les contraintes présentes, la condition de sûreté $x \prec y$ est ajoutée et toutes les contraintes de la forme $z \prec y$, $\forall z \in \mathcal{X} \setminus \{x\}$ sont retirées. Cette définition des conditions de sûreté et de leur évolution suffit à rétablir la terminaison de POB, qui est également comme DBT totalement correct, polynomial en espace et additif sur les unions disjoints (McAllester, 1993).

3.4.4 Variantes et généralisations des retours arrière dynamique et à ordre partiel

Le DBT et le POB ont été chacun légèrement généralisés par Ginsberg et McAllester (1994). La généralisation du retour arrière dynamique ne porte pas un nom distinct, mais est désigné par l'acronyme DB au lieu de DBT. La généralisation du retour arrière à ordre partiel est appelée retour arrière dynamique à ordre partiel et est notée PDB pour *Partial Order Dynamic Backtracking*.

La première différence de DB avec DBT est que, comme POB, DB est décrit comme un algorithme manipulant à tout moment une instanciation complète, sans notion de variables passées ou futures (DB généralise donc à la fois DBT et GDBT). De plus, alors que (G)DBT et POB modifient uniquement la variable coupable pour résoudre le problème, DB et PDB peuvent également modifier toutes les variables qui n'apparaissent dans aucune raison d'élimination présentement active. Cela revient en quelque sorte, du point de vue d'un algorithme construisant une instanciation partielle, à pouvoir continuer les instanciations alors qu'une contrainte est déjà violée. GDBT et POB sont alors du même point de vue des cas particuliers de DB et PDB respectivement qui obligent à s'occuper immédiatement des contraintes violées.

Une dernière variante d'algorithme à saut arrière partiel est le retour arrière à ordre partiel généralisé (GPB ou *Generalized Partial Order Backtracking*, Bliek, 1998), qui généralise à la fois DB et PDB. En effet, GPB gère comme PDB un ordre partiel impliqué à la fois par les raisons des variables éliminées et par un ensemble de conditions de sûreté. GPB laisse toutefois plus de libertés que PDB concernant la modification des conditions de sûreté lors de la gestion d'un conflit ; PDB est donc un cas particulier de GPB de façon évidente. DB peut quant à lui être considéré comme un cas particulier de GPB où les conditions de sûreté maintiennent en tout temps un ordre total sur les variables considérées comme instanciées et où toutes les variables instanciées précèdent les variables non-instanciées, mais sans aucune contrainte entre les variables non-instanciées. GPB reste totalement correct, polynomial en espace et additif sur les

unions disjoints (Bliek, 1998).

3.4.5 Maintenance de l'arc-consistance dans l'algorithme de retour arrière dynamique

Tous les algorithmes de résolution de CSP à saut arrière partiel décrits précédemment sont des algorithmes de recherche pure, c'est-à-dire sans aucun mécanisme d'inférence additionnel en cours d'exécution pour contribuer à élaguer l'espace de recherche. De tels mécanismes peuvent cependant être ajoutés, comme dans le cas de BT et de CBJ ; leur intégration est toutefois plus délicate dans le cas des méthodes à saut arrière partiel. En effet, pour retrouver un état consistant après un saut arrière partiel, il est nécessaire de rétablir les valeurs précédemment éliminées par inférence à partir d'une variable maintenant désinstanciée, sans quoi l'algorithme deviendrait incomplet. Cette conservation de la consistance est bien plus simple dans le cas de BT et de CBJ ; comme leurs sauts arrière défont l'ensemble de la recherche à partir d'un certain point, il suffit d'avoir mémorisé l'état de la recherche à ce point et d'y retourner sans possibilité d'inconsistance. Un saut arrière partiel peut au contraire aboutir à un état de la recherche qui n'a pas été exploré précédemment.

Le mécanisme d'inférence de maintenance de l'arc-consistance a été intégré à l'algorithme de retour arrière dynamique en tenant compte de ces contraintes (Jussien, Debruyne et Boizumault, 2000). Lorsqu'une variable $x \in \mathcal{X}$ est instanciée à une valeur $v \in \delta(x)$, toutes les autres valeurs de cette variable $v' \in \delta(x) \setminus \{v\}$ sont éliminées avec comme raison $\rho(v') = \{x \mapsto v\}$. Lorsqu'une valeur $v \in \delta(x)$ d'une variable $x \in \mathcal{X}$ est éliminée par maintenance de l'arc consistance, cela signifie qu'il existe une contrainte binaire $c \in \mathcal{C}$ de portée $\pi(c) = \{x, x'\}$ où toutes les valeurs de x' supports de v ont été éliminées. La raison d'élimination de v est alors obtenue par l'union des raisons d'élimination des supports de v dans c.

Grâce à cette extension des raisons d'élimination au mécanisme d'inférence, la résolution de conflit de DBT peut rétablir toutes les valeurs dont l'élimination dépend

de la variable coupable défaite. Toutefois, une fois ces valeurs rétablies, le problème peut ne pas être arc-consistant. L'arc-consistance doit donc être rétablie, ce qui peut alors provoquer une cascade de conflits.

3.4.6 Impact des méthodes à saut arrière partiel sur la destructivité des sauts arrière

Toutes les variantes de recherche à saut arrière partiel que nous avons présentées réduisent ou peuvent réduire la destructivité des sauts arrière par rapport à l'algorithme à saut arrière conventionnel CBJ. Cette propriété est particulièrement évidente dans le cas de DBT, qui a été conçu précisément dans le but de cette réduction. En effet, chaque saut arrière partiel de DBT défait uniquement l'instanciation de la variable coupable et rétablit les valeurs éliminées dont la raison dépend de cette instanciation défaite ; ce saut arrière partiel est donc strictement moins destructif qu'un saut arrière de CBJ lorsque la variable coupable n'est pas la dernière variable instanciée, et a une destructivité identique dans le cas contraire.

GDBT défait autant d'instanciations que CBJ mais il rétablit autant de valeurs que DBT. Il est donc au pire aussi destructif que CBJ et peut être moins destructif si certaines valeurs ont été éliminées à cause d'instanciations défaites sans que la variable coupable soit impliquée.

POB est plus difficile à comparer avec CBJ puisqu'il manipule uniquement des instanciations complètes au lieu d'étendre progressivement des instanciations partielles ; cependant, on peut remarquer qu'à chaque résolution de conflit, seule la variable coupable est modifiée, et seules les valeurs éliminées à cause de cette variable coupable sont rétablies. POB défait donc au plus autant de variables que CBJ. L'impact sur les valeurs éliminées est plus difficile à comparer, du fait de la liberté de choix de la variable coupable dans POB ; toutefois, dans tous les cas POB ne rétablit que les valeurs éliminées à cause de la variable coupable choisie, tandis que CBJ peut devoir rétablir les valeurs éliminées par plusieurs variables différentes.

DB, PDB et GPB sont encore plus difficiles à comparer puisqu'en plus de manipuler des instanciations complètes, ils permettent à chaque résolution de conflit de modifier plusieurs instanciations. Le nombre de variables modifiées peut potentiellement être plus grand que le nombre de variables désinstanciées par CBJ dans une situation comparable. Notons cependant qu'hormis la variable coupable, une instanciation ne peut être modifiée arbitrairement dans DB, PDB ou GPB que si elle n'invalide aucune élimination de valeur. L'impact de ces modifications d'instanciations sur la destructivité de la résolution du conflit est donc négligeable. De plus, il est possible de résoudre tout conflit en ne modifiant que l'instanciation de la variable coupable (dans le cas de DB et PDB, on obtient alors les algorithmes DBT et POB respectivement).

Les algorithmes de résolution des CSP à saut arrière partiel ont donc comme point commun de proposer des méthodes de résolution de conflit moins destructives qu'un saut arrière conventionnel dans CBJ. Notons toutefois que cette moindre destructivité ne se traduit pas toujours par une meilleure efficacité en pratique de ces algorithmes. Par exemple, DBT a été démontré expérimentalement plus efficace que CBJ sur certains problèmes (Ginsberg, 1993), mais au contraire bien moins efficace sur d'autres (Baker, 1994). Cette contre-performance relative de DBT s'explique selon Baker par l'impact du retour arrière dynamique sur les heuristiques de choix des variables. En effet, en défaisant certaines instanciations sans respecter leur ordre, DBT modifie a posteriori l'ordre d'instanciation des variables, ce qui peut avoir un impact négatif si l'heuristique de départ est efficace.

3.5 Les sauts arrière illimités

Les différentes variantes de recherche en profondeur à saut arrière présentées dans les sections 3.1 à 3.4 ont pour but ou pour effet de diminuer la destructivité des sauts arrière au cours de la recherche. Cependant, d'autres variantes permettent au contraire des sauts arrière plus destructeurs que les sauts arrière de DPLL ou CBJ respectivement. Plus concrètement, alors que les sauts arrière de DPLL (resp. de CBJ) reviennent toujours au niveau d'assertion (resp. à la variable coupable), ces variantes permettent aux

sauts arrière de retourner jusqu'à un niveau de décision (resp. une variable) quelconque. Nous appellerons **sauts arrière illimités** (d'après Lynce et Marques-Silva, 2002) de tels sauts arrière vers un niveau de décision ou une variable arbitraire. Par exemple, un redémarrage est un cas particulier de saut arrière illimité. Un saut arrière illimité est évidemment plus destructif qu'un saut arrière conventionnel si le niveau de décision (resp. la variable) de retour est antérieur(e) au niveau d'assertion (resp. à la variable coupable). Notons que pour résoudre le conflit, le saut arrière illimité doit au moins défaire le niveau courant dans le cas de SAT ou la dernière variable instanciée dans le cas de CSP.

La stratégie des sauts arrière illimités s'explique par analogie avec les algorithmes de recherche locale pour SAT et CSP. En effet, sans contrainte d'exhaustivité de la recherche, ceux-ci peuvent modifier librement les instanciations des variables du problème et donc se déplacer rapidement dans l'espace de recherche, ce qui peut leur permettre de trouver efficacement des solutions aux problèmes satisfaisables. En comparaison, pour défaire une affectation de variable, les algorithmes de recherche systématique doivent d'abord prouver que le sous-espace de recherche correspondant ne contient aucune solution au problème, c'est-à-dire que l'instanciation partielle constituée de cette affectation et des affectations précédentes ne peut être étendue en une solution. L'efficacité de la recherche est donc grandement conditionnée par les choix initiaux de l'algorithme qui peuvent le bloquer longtemps dans une sous-partie insatisfaisable de l'espace de recherche.

L'objectif des sauts arrière illimités est donc de résoudre ce problème en permettant de défaire arbitrairement des affectations qui semblent a posteriori peu pertinentes ou, de façon équivalente, dont la modification semble plus susceptible de mener à une solution que la modification de l'assertion ou de la variable coupable. Par exemple, dans le cas de SAT, Bhalla et al. (2005) proposent diverses heuristiques de sauts arrière illimités qui choisissent une décision à inverser parmi les niveaux de décision représentés dans la clause de conflit. Le niveau de la décision à inverser peut être choisi en fonction du nombre de variables instanciées à ce niveau de décision présentes dans la clause du

conflit, ou à partir de compteurs d'activités sur les variables similaires aux compteurs utilisés pour l'heuristique de décision.

Notons que, dans le cas de SAT, les sauts arrière illimités sont plutôt une généralisation de DPLL que de CDCL ; en effet, un saut plus profond que celui de CDCL défait le niveau d'assertion, donc la clause de conflit ne provoque aucune propagation et une nouvelle décision doit être prise immédiatement après le saut arrière. Au contraire, lorsque le saut arrière est moins profond que celui de CDCL, la clause de conflit peut être propagée, mais elle ne sera pas propagée au bon niveau, ce qui ressemble au cas de la pseudo-décision dans l'algorithme GRASP.

Comme les sauts arrière illimités peuvent défaire une partie de l'instanciation sans avoir vérifié exhaustivement le sous-espace de recherche associé, un algorithme utilisant de tels sauts arrière est incomplet, et il n'est pas assuré de terminer s'il peut réexplorer des sous-espaces précédemment abandonnées. Cependant, il existe différentes restrictions qui permettent de rétablir à la fois la terminaison et la complétude de la recherche (Lynce et Marques-Silva, 2002). Une solution consiste à alterner les sauts arrière illimités avec des sauts arrière ou retours arrière « conventionnels » et à progressivement augmenter le nombre de sauts et retours arrière conventionnels entre deux sauts arrière illimités (c'est la stratégie couramment employée pour garantir la complétude et la terminaison des algorithmes à sauts arrière avec redémarrages). La terminaison et la complétude sont également assurées si l'algorithme ne supprime jamais aucune clause de conflit pour laquelle le conflit a été résolu par un saut arrière illimité.

L'algorithme de recherche locale contrainte (Prestwich, 2000), qui a été appliqué à SAT, est encore plus général que les algorithmes à sauts arrière illimités (et est donc également incomplet), puisqu'à chaque conflit il défait un nombre arbitraire de décisions (et leurs conséquences) sans avoir à défaire les niveaux ultérieurs à ces décisions défaites.

L'algorithme de réparation de décisions (*decision repair*, Jussien et Lhomme, 2002) est un algorithme de recherche en profondeur non-systématique pour le problème CSP. Lorsqu'un conflit est détecté, l'algorithme le résout en défaisant l'affectation d'une des

variables de la portée de la contrainte violée. DBT et POB sont des cas particuliers de la réparation de décisions qui imposent certaines restrictions sur les choix de variables à désinstancier. Dans le cas général, l'algorithme de réparation de décision est incomplet ; toutefois, Pralet et Verfaillie (2005) ont montré que la complétude peut être rétablie à l'aide de différentes stratégies de restrictions sur le choix de la variable à défaire.

CHAPITRE IV

DÉCOMPOSITION IMPLICITE DE GRANDES INSTANCES SAT

Ce chapitre décrit une contribution secondaire de cette thèse. Il est principalement motivé par le constat que les implémentations existantes de décomposition implicites pour SAT, décrites dans la sous-section 3.1.6, semblent être uniquement capables de traiter des instances de taille modeste. Notre objectif est donc de concevoir une méthodologie de décomposition implicite dont la principale caractéristique est une robustesse accrue par rapport à la taille des instances traitées. Dans ce but, nous introduisons la notion de séparation arborescente, un formalisme simplifié par rapport aux décompositions arborescentes et aux *dtrees,* et proposons une heuristique capable de générer rapidement des séparations arborescentes sur des instances de grande taille. L'utilité de cette heuristique au sein d'un algorithme de décomposition implicite est évaluée en implémentant une modification d'un solveur CDCL.

La section 4.1 présente un aperçu des algorithmes de construction de décompositions arborescentes, plus particulièrement ceux utilisés dans les implémentations de décomposition implicite de SAT. La section 4.2 vérifie expérimentalement la limitation d'une de ces implémentations par rapport à la taille des instances traitées. Les sections 4.3 et 4.4 présentent respectivement la définition de séparation arborescente et notre heuristique de construction de séparation arborescente. La section 4.5 détaille et commente les résultats expérimentaux obtenus avec notre implémentation de décomposition implicite utilisant cette heuristique. La section 4.6 conclut en résumant les enjeux et résultats de ce chapitre.

4.1 Heuristiques de construction de décompositions arborescentes

Comme nous l'avons vu dans la sous-section 3.1.2, déterminer la largeur d'arborescence exacte d'un graphe est un problème NP-complet. Par conséquent, tout algorithme construisant une décomposition d'une instance SAT ou CSP de largeur minimale est exponentiel selon la taille de l'instance ; il est donc irréaliste d'utiliser un tel algorithme en pratique. Il est toutefois possible de rechercher des décompositions d'instances en relâchant la contrainte de largeur minimale ; on obtient alors des algorithmes d'approximation ou heuristiques selon l'intensité de ce relâchement.

Un **algorithme d'approximation** permet de construire plus efficacement une décomposition arborescente sans chercher absolument une largeur minimale, mais en garantissant tout de même une certaine proximité entre la largeur de l'arborescence obtenue et la largeur d'arborescence du graphe. L'algorithme peut rester exponentiel ou devenir polynomial selon la précision de l'approximation. Par exemple, Amir (2010) propose deux types d'algorithmes d'approximation. Pour un graphe G à n sommets et de largeur d'arborescence $\omega(G)$, le premier type construit une décomposition arborescence de largeur $\omega(G) \times \log(\omega(G))$ en temps polynomial. Le second type permet une approximation plus fine, puisqu'il construit une décomposition arborescente de largeur $k \times \omega(G)$ où $k \in \{3,66; 4; 4,5\}$ est une constante dont la valeur dépend de l'algorithme exact utilisé ; en contrepartie, cette famille d'algorithme a une complexité exponentielle.

Cependant, en pratique, même les algorithmes d'approximation polynomiaux sont trop peu efficaces pour être utilisés dans les méthodes de décomposition implicite pour SAT et CSP. Celles-ci ont donc recours à des **algorithmes heuristiques**, qui cherchent également à construire une décomposition arborescente de largeur la plus petite possible, mais ne donnent aucune garantie par rapport à la largeur d'arborescence du graphe décomposé. L'intérêt de cette absence de contrainte est de réduire encore le temps d'exécution par rapport aux algorithmes d'approximation, y compris polynomiaux. Par exemple, l'algorithme de construction d'un arbre de *tree clustering* à l'aide d'une triangulation heuristique, décrit dans la sous-section 3.1.3, est un algorithme de décomposition arbo-

rescente heuristique, qui est par exemple utilisé par l'implémentation de BTD (Jégou et Terrioux, 2003). La construction de décompositions arborescentes peut aussi employer diverses métaheuristiques telles que la recherche locale itérée (Musliu, 2008), la recherche tabou (Clautiaux et al., 2004) ou les algorithmes évolutionnistes (Hammerl et Musliu, 2010).

Park et van Gelder (1996) et Durairaj et Kalla (2004) effectuent tous deux une seule séparation globale d'une instance SAT en recherchant un ensemble coupant de l'hypergraphe dual de l'instance. Park et van Gelder (1996) utilisent un algorithme heuristique itératif pour le problème d'ensemble coupant minimal (Fiduccia et Mattheyses, 1982) ; Durairaj et Kalla (2004) et Li et van Beek (2004) utilisent une autre heuristique pour le même problème, hMETIS (Karypis et al., 1999), qui effectue des contractions successives de l'hypergraphe initial, c'est-à-dire des fusions de différents sommets et hyperarêtes, calcule un ensemble coupant de l'hypergraphe contracté et l'utilise pour retrouver un ensemble coupant de l'hypergraphe d'origine. Huang et Darwiche (2003) utilisent également une suite d'appels récursifs à hMETIS pour construire un *dtree* de l'instance SAT considérée (Darwiche et Hopkins, 2001), ainsi que Durairaj et Kalla (2004) pour la construction de leur décomposition arborescente. Enfin, Bjesse et al. (2004) se servent d'une heuristique de construction de décompositions arborescentes à partir d'un ordre d'élimination des variables.

4.2 Limites des implémentations de décomposition implicite pour SAT

L'efficacité de certaines implémentations de décomposition implicite pour SAT a été démontrée expérimentalement : sur certaines instances de problèmes, la construction heuristique d'une décomposition, puis l'exécution d'un solveur SAT implémentant une recherche en profondeur avec un ordre partiel sur les variables induit par la décomposition, est significativement plus rapide que l'exécution directe de ce solveur avec sa stratégie de décision par défaut (Park et van Gelder, 1996; Huang et Darwiche, 2003;

Bjesse et al., 2004; Durairaj et Kalla, 2004; Li et van Beek, 2004)[1]. Cependant, dans tous ces cas, les tests ont été effectués uniquement sur des instances de problèmes de taille relativement petite. En effet, le tableau 4.1 indique que seules des instances d'au plus 10 026 variables et 195 452 clauses ont été résolues expérimentalement à l'aide de ces implémentations (la grande majorité des instances traitées étant en-dessous de 6 000 variables et 100 000 clauses). En comparaison, la taille moyenne des instances dites **applicatives** (c'est-à-dire issues de l'encodage de problèmes tels que ceux énumérés dans la sous-section 2.2.6, par opposition à des instances « artificielles » générées aléatoirement) utilisées lors de la compétition SAT de 2009 est de 158 684 variables et 876 893 clauses. Ces moyennes sont fortement influencées par certaines instances particulièrement grandes (jusqu'à 10 950 109 variables et 32 697 150 clauses), par conséquent les médianes correspondantes sont plus réduites (60 215 variables et 253 404 clauses). Même en considérant ces médianes, les instances utilisées dans ces études expérimentales sont donc significativement petites par rapport aux dimensions typiques des instances considérées comme difficiles, qui sont donc celles sur lesquelles les bénéfices des décompositions sont potentiellement les plus importants.

Dans le but de vérifier la limite de taille des instances pouvant être résolues par les implémentations existantes de décomposition implicite, nous avons testé DTREE-ZCHAFF, l'implémentation de Huang et Darwiche (apparemment la seule à être publiquement accessible) sur un ensemble d'instances applicatives de tailles variées. Comme son nom l'indique, DTREE-ZCHAFF est une modification du solveur CDCL ZCHAFF (Moskewicz et al., 2001), plus précisément de sa version originale de 2001. Ces tests, comme tous ceux présentés dans cette thèse, ont été effectués sur un PC muni d'un processeur Intel Core 2 Duo cadencé à 3,16 GHz, de 3 GB de mémoire vive, et du système d'exploitation Ubuntu (la version du système utilisée variant de 9.04 à 12.04 selon les cas).

L'évaluation expérimentale a été effectuée sur 7 séries d'instances sélectionnées parmi les instances de diverses compétitions SAT (Le Berre et al., 2012), qui comportent

1. Les résultats expérimentaux de Bjesse et al. (2004) ne comparent pas les temps d'exécution des différents dispositifs, mais uniquement le nombre total de décisions prises.

TABLEAU 4.1: Taille maximale en variables (*var. max*) et en clauses (*cl. max*) des instances SAT résolues à l'aide de différentes *implémentations* de décomposition implicite. La taille maximale est une estimation basse dans le cas de Li et van Beek (2004), car les résultats expérimentaux fournissent uniquement une borne inférieure sur le nombre d'instances résolues avec succès dans chaque famille d'instances considérée. Les données reportées correspondent à la plus grande instance (à la fois en nombre de variables et de clauses) dont il est absolument certain qu'elle a été résolue avec succès étant données ces informations partielles. La plus grande instance incluse dans les familles d'instances considérées comporte 49 153 variables et 324 873 clauses.

implémentation	*var. max*	*cl. max*
Park et van Gelder (1996)	728	2 200
Huang et Darwiche (2003)	2 125	24 792
Bjesse et al. (2004)	6 000	16 000
Durairaj et Kalla (2004)	10 026	195 452
Li et van Beek (2004)	9 207	30 678

134 instances au total. Ces instances ont une moyenne de 194 067 variables et 723 589 clauses pour une médiane de 111 605 variables et 358 472 clauses, des statistiques relativement similaires à celles de l'ensemble des instances de la compétition SAT 2009. Une liste exhaustive des instances de test et de leur taille est disponible dans le tableau A.1 en annexe. Les différentes séries proviennent de champs applicatifs divers :

- les 34 instances dont le nom commence par AProVE sont des encodages de problèmes de terminaison de réécriture, générés par le prouveur de terminaison APROVE (Giesl, Schneider-Kamp et Thiemann, 2006) ;
- les 30 instances dont le nom commence par dspam_dump, hsat, itox ou xinetd proviennent de la vérification de logiciels par l'outil CALYPSO (Babić et Hu, 2007) ;
- les 20 instances dont le nom commence par q_query encodent des requêtes sur des réseaux de régulation de gènes (Corblin et al., 2011) ;
- les 7 instances dont le nom commence par eq.atree.braun encodent des problèmes d'équivalence de circuits électroniques (Järvisalo, 2007) ;
- les 27 instances commençant par blocks, cube, emptyroom, safe, sortnet ou uts sont des encodages de problèmes de planification conformante (Palacios et Geffner, 2006) ;

- les 11 instances de **abp1-1-k31** à **valves-gates-1-k617** encodent des problèmes de vérification de modèle bornée et ont été générées par le vérificateur CADENCE SMV (Schuppan et Biere, 2004) ;
- enfin, les 5 instances dont le nom commence par **clauses** sont des encodages de problèmes de configuration réseau, générés par le trouveur de modèle ALLOY (Narain, 2005) .

Les résultats expérimentaux obtenus sont présentés dans le tableau 4.2. Plus précisément, ce tableau présente les résultats sur les 36 instances comportant 140 000 clauses ou moins, DTREE-ZCHAFF n'étant parvenu à décomposer aucune des instances comportant plus de clauses. Dans le but de comparer le temps de la construction des décompositions par rapport au temps nécessaire à résoudre directement les instances, le tableau inclut également les temps d'exécution de la version 2.0 de MINISAT (Eén et Sörensson, 2004), un solveur CDCL qui a été parmi les plus performants dans les compétitions SAT pendant plusieurs années. Ce temps d'exécution n'est toutefois pas réellement comparable avec le temps d'exécution total de DTREE-ZCHAFF, car MINISAT est souvent plus performant que ZCHAFF. Chaque exécution de DTREE-ZCHAFF ou de MINISAT sur une instance est limitée à une heure.

Comme l'indiquent nos résultats, une large majorité des exécutions de DTREE-ZCHAFF (104 sur 134) a été interrompue avant la limite d'une heure sans retourner de résultat. Dans la quasi-totalité de ces cas d'échecs, DTREE-ZCHAFF a été interrompu pendant la construction du *dtree* de l'instance ; les seules exceptions sont les instances **AProVE07-26** et **AProVE09-20** que DTREE-ZCHAFF a réussi à décomposer avant d'être interrompu pendant la recherche CDCL. De plus, les échecs d'exécution sont fortement corrélés avec la taille des instances traitées : les plus grandes instances traitées avec succès par DTREE-ZCHAFF contiennent respectivement 37 821 variables et 128 361 clauses, des dimensions bien inférieures à la médiane de notre ensemble d'instances. La plus petite instance que DTREE-ZCHAFF a échoué à décomposer ne contient que 7 847 variables et 73 394 clauses. On peut en déduire que les échecs de l'implémentation sont certainement dus à un dépassement des capacités mémoires disponibles en raison de la taille des

TABLEAU 4.2: Résultats de l'exécution de DTREE-ZCHAFF et de MINISAT sur certaines instances du problème SAT. Les instances sont présentées par ordre croissant de nombre de clauses. Pour chaque *instance* sont indiqués son nombre de variables (*#var*) et de clauses (*#cl*), la largeur du *dtree* généré par DTREE-ZCHAFF (*l*), le temps nécessaire à cette génération (*tdc*), le temps total d'exécution de DTREE-ZCHAFF (*tdz*) et le temps d'exécution de MINISAT (*tmsat*). La mention « échec » signifie que l'exécution de DTREE-ZCHAFF a été interrompue sans retourner de réponse pendant la génération du *dtree* ou pendant la recherche CDCL selon le cas.

instance	*#var*	*#cl*	*l*	*tdc*	*tdz*	*tmsat*
eq.atree.braun.7	505	1 696	47	0.90	1.88	1.40
eq.atree.braun.8	684	2 300	57	1.27	31.07	7.37
eq.atree.braun.9	892	3 006	64	1.71	345.27	44.06
eq.atree.braun.10	1 111	3 756	71	2.27	>3 600.00	291.80
eq.atree.braun.11	1 400	4 732	79	2.96	>3 600.00	1 674.00
eq.atree.braun.12	1 694	5 726	86	3.68	>3 600.00	>3 600.00
eq.atree.braun.13	2 010	6 802	95	4.53	>3 600.00	>3 600.00
AProVE07-03	3 114	10 827	229	8.13	>3 600.00	727.46
AProVE07-21	3 189	11 039	68	6.61	151.21	244.91
AProVE07-08	4 614	16 637	239	12.34	>3 600.00	773.36
AProVE07-02	6 196	22 741	608	22.60	2 765.01	336.61
AProVE09-13	7 606	26 317	120	16.64	16.74	0.04
AProVE07-01	7 502	28 770	855	33.23	>3 600.00	>3 600.00
AProVE09-08	8 564	28 927	101	18.86	18.97	0.05
AProVE09-07	8 567	28 936	96	18.90	19.88	3.80
AProVE07-27	7 729	29 194	215	21.01	>3 600.00	1 733.10
AProVE07-25	8 920	31 884	266	24.81	>3 600.00	>3 600.00
AProVE09-22	11 557	38 505	288	27.69	27.95	0.06
abp1-1-k31	14 809	48 483	156	42.18	>3 600.00	31.21
abp4-1-k31	14 809	48 483	156	42.44	>3 600.00	31.33
AProVE09-05	14 685	49 510	111	35.15	35.53	0.07
AProVE07-22	15 589	54 263	227	42.64	179.19	54.32
AProVE07-20	7 847	73 394		échec		80.23
AProVE07-15	21 104	74 257	272	67.23	>3 600.00	135.38
AProVE09-11	20 192	78 082	240	60.43	61.05	0.23
AProVE07-26	21 734	79 766	662	103.35	échec	>3 600.00
AProVE09-21	29 964	91 044	468	98.18	98.84	244.91
sortnet-6-ipc5-h11	27 724	95 880		échec		2 733.18
AProVE09-12	27 495	102 011	409	84.41	84.93	0.32
AProVE09-20	33 054	108 377	387	117.18	échec	1 929.83
AProVE09-17	33 894	108 759	226	96.82	178.60	12.77
AProVE09-19	30 537	112 780	254	116.11	118.03	0.25
AProVE09-25	37 821	124 485	1 123	138.41	139.62	0.26
q_query_3_l37_lambda	26 241	128 361	492	121.63	139.77	1.57
q_query_3_l38_lambda	26 986	132 388		échec		1.88
q_query_3_l39_lambda	27 735	136 449		échec		13.95

instances traitées. Nous supposons que les deux exécutions qui ont échoué pendant la recherche CDCL ont disposé de suffisamment d'espace mémoire pour effectuer la décomposition, mais que l'espace restant n'a pas été suffisant pour la mémorisation des clauses apprises lors de la recherche.

Enfin, si le temps nécessaire à la décomposition est à peu près proportionnel à la taille de l'instance traitée, il est le plus souvent non-négligeable et rend souvent difficile, voire impossible, d'améliorer l'efficacité de la résolution du problème par rapport à une recherche CDCL directe. En effet, parmi les 32 instances que DTREE-ZCHAFF parvient à décomposer, la décomposition prend plus de temps que la résolution directe par MINISAT dans 14 cas, soit près de la moitié.

Ces résultats confirment clairement que seules les instances de taille modérée peuvent être traitées à l'aide de cette implémentation. Si l'implémentation de Durairaj et Kalla est parvenue à résoudre des instances sensiblement plus grandes, on peut cependant supposer qu'elle souffre d'une limitation comparable, car elle repose comme DTREE-ZCHAFF sur une utilisation d'HMETIS.

Nos résultats expérimentaux mettent donc en évidence deux handicaps important des implémentations existantes de décomposition implicite : d'une part, leur incapacité à traiter des instances de taille conséquente ; d'autre part, le temps nécessaire à la décomposition, qui est souvent bien trop important par rapport au temps nécessaire à résoudre directement l'instance.

4.3 Séparations arborescentes

Comme nous l'avons montré dans la section précédente, les implémentations existantes pour construire des décompositions arborescentes sont trop complexes pour être utilisables sur des grandes instances, ou même souvent pour être efficaces sur des instances de taille plus réduite. En effet, DTREE-ZCHAFF (Huang et Darwiche, 2003), l'implémentation que nous avons testée, utilise HMETIS (Karypis et al., 1999), qui transforme l'hypergraphe primal par contractions successives. Il n'est pas étonnant que

de telles manipulations sur des graphes de cette taille dépassent les capacités de la machine, notamment en mémoire vive.

Nous pouvons identifier deux sources principales qui expliquent cette complexité. La première est l'utilisation d'heuristiques relativement sophistiquées pour chercher à obtenir des décompositions de largeur aussi petite que possible. Pour contourner ce problème, il suffit d'utiliser des heuristiques de construction les plus simples possible, quitte à obtenir des décompositions moins optimisées. C'est ce que nous proposerons dans la section 4.4.

La seconde source de complexité est due simplement à la définition des décompositions arborescentes et des *dtrees*, qui imposent toutes deux des conditions de validité relativement compliquées. Pour permettre une décomposition implicite applicable à des instances de plus grande taille, il est donc logiquement nécessaire de simplifier les contraintes de constructions de décompositions. Nous allons donc proposer une nouvelle définition qui généralise à la fois les décompositions arborescentes et les *dtrees* en se focalisant uniquement sur l'objectif central des décompositions implicites, c'est-à-dire la séparation récursive du graphe primal de l'instance.

Définition 4.1. *Une* ***séparation arborescente généralisée*** *d'un hypergraphe $H(\mathcal{S}, \mathcal{H})$ est un couple $(\mathcal{T}, \varsigma)$ formé d'un arbre enraciné $\mathcal{T}(T(\mathcal{N}, \mathcal{A}), r)$ et d'une fonction $\varsigma : \mathcal{N} \to \mathcal{P}(\mathcal{S})$ tels que :*

- *Chaque sommet de H est associé à au moins un nœud de $\mathcal{T}$: $\bigcup_{n \in \mathcal{N}} \varsigma(n) = \mathcal{S}$.*
- *Pour tout ensemble de nœuds $N \subseteq \mathcal{N}$, notons $\varsigma(N) = \bigcup_{n \in N} \varsigma(n)$. $\forall n \in \mathcal{N}$, soit $\text{res}(H, n) = H|_{\varsigma(\{n\} \cup d(n)) \setminus \varsigma(a(n))}$ l'hypergraphe résiduel correspondant au sous-arbre enraciné en n, c'est-à-dire induit par l'ensemble des sommets présents dans n ou ses descendants sans être présent dans ses ancêtres. Alors, pour tout nœud interne $n \in \mathcal{N}$, $\varsigma(n)$ est un séparateur de $\text{res}(H, n)$ et pour tout fils de n, $n' \in f(n)$, $\varsigma(\{n'\} \cup d(n')) \setminus \varsigma(\{n\} \cup a(n))$ est une composante connexe ou l'union de plusieurs composantes connexes de $\text{res}(H, n)_{\setminus \varsigma(n)}$.*

Lemme 4.1. *Les décompositions arborescentes et les* dtrees *sont des cas particuliers de*

séparations arborescentes généralisées.

Démonstration. Une décomposition arborescente enracinée en un nœud quelconque respecte les propriétés d'une séparation arborescente généralisée, puisque pour tout nœud $n \in \mathcal{N}$, $\varsigma(n)$ est un séparateur de H et toute composante connexe de $H_{\setminus \varsigma(n)}$ est incluse dans l'ensemble des sommets associés à une composante connexe de $T_{\setminus \{n\}}$. Une décomposition arborescente est donc une séparation arborescente généralisée pour tout enracinement possible, qui vérifie en plus la couverture de toute hyperarête par au moins un nœud et la propriété de connectivité.

Un *dtree* est une séparation arborescente généralisée binaire qui vérifie de plus les propriétés suivantes :

- Pour tout ensemble de sommets $S \subseteq \mathcal{S}$, il existe une feuille $n \in \mathcal{N}$ telle que $\varsigma(n) = S$ si et seulement si S est une hyperarête de H, c'est-à-dire si $S \in \mathcal{H}$.
- Pour tout nœud interne $n \in \mathcal{N}$, soient n_1 et n_2 ses deux fils et $\varphi(n) \subseteq d(n)$ l'ensemble des feuilles du sous-arbre enraciné à n. Alors pour tout nœud interne $n \in \mathcal{N}$, l'ensemble des sommets associés à n est exactement l'ensemble des sommets associés à la fois à des feuilles des sous-arbres enracinés à n_1 et n_2 sans être associé à aucun ancêtre de n : $\varsigma(n) = (\varsigma(\varphi(n_1)) \cap \varsigma(\varphi(n_2))) \setminus \varsigma(a(n))$.

□

Comme les conditions de validité d'une séparation arborescente généralisée sont plus simples que celles d'une décomposition arborescente ou d'un *dtree*, il devrait être plus facile de construire une telle structure. Elle conserve toutefois la propriété essentielle pour une décomposition implicite : si l'on ordonne partiellement les variables d'une instance à l'aide d'un parcours préfixe d'une séparation arborescente de l'hypergraphe primal de l'instance, tel que décrit dans le cas particulier d'une décomposition arborescente ou d'un *dtree* dans la section 3.1.6, un algorithme CDCL dont l'heuristique de décision suit cet ordre partiel effectue bien une séparation récursive du problème et

simule la résolution indépendante des sous-problèmes.

Afin d'obtenir plus directement encore un ordre partiel, nous nous restreindrons à un cas particulier de séparations arborescentes :

Définition 4.2. *Une **séparation arborescente** $(\mathcal{T}(T(\mathcal{N},\mathcal{A}),r),\varsigma)$ d'un hypergraphe $H(\mathcal{S},\mathcal{H})$ est une séparation arborescente généralisée où tout sommet de l'hypergraphe est associé à exactement un nœud de l'arborescence. Autrement dit, $\{\varsigma(n) \mid n \in \mathcal{N}, \varsigma(n) \neq \emptyset\}$ est une partition de $\mathcal{N}$.*

Dans le cas d'une séparation arborescente, l'hypergraphe résiduel correspondant au sous-arbre enraciné en un nœud $n \in \mathcal{N}$ se définit par $\text{res}(H,n) = H|_{\varsigma(\{n\}\cup d(n))}$. Comme $\forall\{n_1,n_2\} \subseteq \mathcal{N}$, $\varsigma(n_1) \cap \varsigma(n_2) \neq \emptyset$, la propriété de décomposition implicite se simplifie de cette façon : si $(n_1,n_2,\ldots,n_{|\mathcal{N}|})$ est un parcours préfixe des nœuds de la séparation arborescente, alors une recherche à saut arrière munie de l'ordonnancement partiel $(\varsigma(n_1),\varsigma(n_2),\ldots,\varsigma(n_{|\mathcal{N}|}))$ effectue récursivement une résolution indépendante des sous-problèmes résiduels séparés par l'instanciation des nœuds internes de l'arborescence.

4.4 Séparation heuristique récursive d'un hypergraphe

Avec les séparations arborescentes, nous disposons d'une structure de description de séparations récursives très peu contraignante, qui nous permet d'utiliser des heuristiques de construction aussi simples que possible, dans le but de pouvoir traiter des instances SAT de taille importante avec des ressources en temps et en espace négligeables.

La stratégie de construction que nous proposons répond à cet objectif de simplicité et se base sur l'application récursive d'une heuristique de séparation d'hypergraphe nommée séparation par coupure linéaire. La sous-section 4.4.1 détaille cette heuristique de séparation, tandis que la sous-section 4.4.2 aborde la construction d'une séparation arborescente par applications récursives de cette technique de séparation.

4.4.1 Séparation d'hypergraphe par coupure linéaire

La séparation par **coupure linéaire** utilise un ordre linéaire (c'est-à-dire total) sur les sommets de l'hypergraphe et chercher à effectuer une coupure sur cet ordre, autrement dit à séparer le graphe en deux composantes connexes telles que tous les sommets d'une composante sont inférieurs à tous les sommets de l'autre composante. Pour cela, l'heuristique choisit un sommet arbitraire de coupure, $s_c \in \mathcal{S}$, pour servir de délimitation entre les deux composantes connexes. Une hyperarête $h \in \mathcal{H}$ **traverse** s_c si elle contient à la fois un sommet strictement inférieur à s_c et un sommet strictement supérieur à s_c. Si aucune arête de H ne traverse s_c, alors il est évident que si s_c n'est ni le premier ni le dernier sommet, $\{s_c\}$ est un séparateur de H tel que $\{s \in \mathcal{S} \mid s < s_c\}$ et $\{s \in \mathcal{S} \mid s > s_c\}$ sont des composantes déconnectées dans $H_{\setminus\{s_c\}}$. Sinon, pour éliminer une arête h qui traverse s_c, il suffit d'inclure toutes les variables de h dans le séparateur. Par conséquent, si $\mathrm{tr}(s_c) \subseteq \mathcal{H}$ désigne l'ensemble des hyperarêtes qui traversent s_c, alors $S_c = \{s_c\} \cup \bigcup_{h \in \mathrm{tr}(s_c)} h$ est un séparateur de H et $\{S_{\inf} = \{s \in \mathcal{S} \mid s < s_c,\ s \notin S_c\}$, $S_{\sup} = \{s \in \mathcal{S} \mid s > s_c,\ s \notin S_c\}\}$ est une partition de $\mathcal{S} \setminus S_c$ en deux composantes déconnectées dans $\mathcal{H}_{\setminus S_c}$, à condition que $S_{\inf}$ et $S_{\sup}$ soient non-vides.

L'algorithme 4.1, CoupureLinéaire, décrit cette heuristique de séparation. Notons qu'il consomme très peu de temps, puisqu'il ne parcourt qu'une seule fois chaque hyperarête. D'un point de vue asymptotique, il s'exécute en temps $O(|\mathcal{H}| \times k \times \log(|\mathcal{S}|))$ où $k \leq |\mathcal{S}|$ est le nombre maximal de sommets par hyperarête. $\log(|\mathcal{S}|)$ correspond à l'utilisation de structures telles que les arbres rouge et noir pour gérer les insertions dans l'ensemble S_c dont la taille est bornée par le nombre total de sommets.

Cet algorithme a deux paramètres importants : l'ordre total sur les sommets et le choix du sommet de coupure. L'ordre entre les sommets peut être choisi heuristiquement pour chercher à optimiser les propriétés de la séparation obtenue ; par exemple, on peut chercher à minimiser le nombre d'hyperarêtes qui traversent la variable de coupure en ordonnant les sommets de façon à minimiser heuristiquement les distances entre les variables connectées par des hyperarêtes. MINCE (Aloul, Markov et Sakallah, 2004)

Algorithme 4.1 COUPURELINÉAIRE$(H(\mathcal{S},\mathcal{H}), s_c)$

1: $s_c \in \mathcal{S}$
2: $S_c \leftarrow \{s_c\}$
3: **pour tout** $h \in \mathcal{H}$ **faire**
4: **si** $\exists s_1, s_2 \in h \mid s_1 < s_c < s_2$ **alors**
5: $S_c \leftarrow S_c \cup h$
6: $S_{\text{inf}} \leftarrow \{s \in \mathcal{S} \mid s < s_c\} \setminus S_c$
7: $S_{\text{sup}} \leftarrow \{s \in \mathcal{S} \mid s > s_c\} \setminus S_c$
8: **retourner** $(S_{\text{inf}}, S_c, S_{\text{sup}})$

est un exemple d'une telle heuristique d'ordonnancement. Il est également possible de choisir un ordre facile à déterminer afin d'économiser du temps de calcul. C'est cette dernière approche que nous avons adoptée : nous conservons l'ordre sur les sommets de la numérotation utilisée dans les fichiers de description d'instances SAT. En plus d'économiser le temps de calcul nécessaire à un réordonnancement heuristique, cette stratégie a comme avantage qu'en pratique, les variables sont généralement numérotées dans l'ordre de leur première apparition dans la description, et cette description semble être souvent structurée et introduire consécutivement des sommets liés entre eux.

Concernant le choix du sommet de coupure, pour un hypergraphe à $|\mathcal{S}|$ sommets, nous choisissons le $\lceil \frac{|\mathcal{S}|}{2} \rceil^{\text{ième}}$ sommet, afin de favoriser l'équilibre entre les tailles des composantes déconnectées S_{inf} et S_{sup}. Cet équilibre, comme nous l'avons vu dans la sous-section 3.1.6, est essentiel à l'efficacité asymptotique des algorithmes de décomposition implicite ; on peut donc s'attendre à ce que les performances en pratique en bénéficient également.

4.4.2 Construction d'une séparation arborescente par séparations récursives

La définition des séparations arborescentes permet de construire facilement une arborescence de façon progressive. En effet, on peut localement rajouter des fils à une feuille de l'arborescence sans tenir aucun compte des autres nœuds.

Proposition 4.1. *Soit $H(\mathcal{S},\mathcal{H})$ un hypergraphe et $(\mathcal{T}(T(\mathcal{N},\mathcal{A}),r),\varsigma)$ une séparation ar-*

Algorithme 4.2 SÉPARERNŒUD$(n, \mathcal{T}(T(\mathcal{N},\mathcal{A}),r), \varsigma, H(\mathcal{S},\mathcal{H}))$

Requis : $n \in \mathcal{N}$

1: /*séparer le sous-hypergraphe induit par $\varsigma(n)$*/
2: $\mathcal{S}_n \leftarrow \varsigma(n)$
3: $\mathcal{H}_n \leftarrow \{h \in \mathcal{H} \mid h \subseteq \mathcal{S}_n\}$
4: **choisir** $s_c \in \mathcal{S}_n$
5: $(S_{\text{inf}}, S_c, S_{\text{sup}}) \leftarrow$ COUPURELINÉAIRE$(H_n(\mathcal{S}_n, \mathcal{H}_n), s_c)$
6: **si** $S_{\text{inf}} \neq \emptyset$ **et** $S_{\text{sup}} \neq \emptyset$ **alors** /*si S_c est bien un séparateur*/
7: /*créer deux nœuds n_{inf} et n_{sup} fils de n*/
8: $n_{\text{inf}} \leftarrow$ CRÉERNOEUD()
9: $n_{\text{sup}} \leftarrow$ CRÉERNOEUD()
10: $\mathcal{N} \leftarrow \mathcal{N} \cup \{n_{\text{inf}}, n_{\text{sup}}\}$
11: $\mathcal{A} \leftarrow \mathcal{A} \cup \{\{n, n_{\text{inf}}\}, \{n, n_{\text{sup}}\}\}$
12: $\varsigma(n_{\text{inf}}) \leftarrow S_{\text{inf}}$
13: $\varsigma(n) \leftarrow S_c$
14: $\varsigma(n_{\text{sup}}) \leftarrow S_{\text{sup}}$
15: /*séparation récursive de n_{inf} et n_{sup}*/
16: **si** CONTINUERSÉPARATION(n_{inf}) **alors**
17: SÉPARERNŒUD$(n_{\text{inf}}, \mathcal{T}(T(\mathcal{N},\mathcal{A}),r), \varsigma, H(\mathcal{S},\mathcal{H}))$
18: **si** CONTINUERSÉPARATION(n_{sup}) **alors**
19: SÉPARERNŒUD$(n_{\text{sup}}, \mathcal{T}(T(\mathcal{N},\mathcal{A}),r), \varsigma, H(\mathcal{S},\mathcal{H}))$

borescente de H. Pour toute feuille $n \in \mathcal{N}$, soit $H|_{\varsigma(n)}(\varsigma(n), \mathcal{H}|_{\varsigma(n)})$ le sous-hypergraphe de H induit par $\varsigma(n)$. Soit $S \subseteq \varsigma(n)$ un séparateur de $H|_{\varsigma(n)}$ et $(S_1, S_2, \ldots, S_k)$ une partition de $\varsigma(n) \setminus S$ en composantes déconnectées dans $H|_{\varsigma(n)\setminus S}$. Alors $(\mathcal{T}'(T'(\mathcal{N}', \mathcal{A}'), r), \varsigma')$ est également une séparation arborescente de H, où :

- $\mathcal{N}' = \mathcal{N} \cup \{n_1, n_2, \ldots, n_k\}$, $n_1, n_2, \ldots, n_k \notin \mathcal{N}$;
- $\mathcal{A}' = \mathcal{A} \cup \bigcup_{i=1}^{k} \{n, n_i\}$;
- $\forall n' \in \mathcal{N}$, $\varsigma'(n') = \begin{cases} S & \text{si } n' = n \\ S_i & \text{si } n' = n_i,\ i \in \{1, 2, \ldots, k\} \\ \varsigma(n') & \text{sinon} \end{cases}$.

Grâce à cette propriété, il est possible de construire une séparation arborescente en partant d'un cas trivial, puis en séparant récursivement ses feuilles. L'algorithme 4.2, SÉPARERNŒUD, implémente la stratégie de la proposition 4.1 en utilisant la stratégie de séparation par COUPURELINÉAIRE décrite dans la sous-section précédente. Notons que

l'algorithme effectue la séparation uniquement si S_c est bien un séparateur, c'est-à-dire si S_{inf} et S_{sup} sont tous deux non-vide. En pratique, notre implémentation permet la séparation lorsqu'au moins un de ces ensembles est non-vide, afin d'éviter si possible un ordonnancement partiel des variables trop peu contraint ; l'arborescence n'est alors cependant plus une séparation arborescente.

SÉPARERNŒUD est bien récursive : après avoir rajouté les deux nouvelles feuilles n_{inf} et n_{sup}, celles-ci sont elles-mêmes séparées si elles vérifient la condition CONTINUERSÉPARATION, qui permet de paramétrer l'arrêt de la récursion. Le cas de base de la récursion est représenté par l'algorithme 4.3, SÉPARATIONARBORESCENTE : la séparation arborescente initiale comporte un seul nœud qui contient toutes les variables du problème. Il est possible de continuer les séparations récursives tant que les coupures linéaires produisent des séparations, mais nous n'avons pas remarqué d'amélioration des performances avec une telle stratégie. En effet, comme la coupure linéaire est une heuristique de séparation très peu optimisée, les séparateurs qu'elle produit sont souvent relativement gros, et par conséquent la taille des feuilles séparées récursivement décroît très vite. Si l'on ne limite pas les séparations, on obtient alors une arborescence de séparation beaucoup plus détaillée en profondeur. Or, ce sont les nœuds les moins profonds qui ont le plus d'impact sur la recherche, puisque les variables qui s'y trouvent seront instanciées beaucoup plus souvent et longtemps. En pratique, la stratégie que nous avons adoptée est de séparer une feuille uniquement si elle comporte plus de sommets que le séparateur à la racine de l'arborescence. Les résultats de la recherche CDCL avec un ordre induit de cette façon sont quasiment identiques à ceux obtenus avec une séparation récursive maximale, tout en ayant à construire et à gérer une arborescence moins volumineuse.

Il reste un paramètre de l'algorithme, qui est l'ordre des nœuds de l'arborescence dans le parcours préfixe. Il est en théorie possible d'effectuer ce parcours dynamiquement au cours de la recherche CDCL, c'est-à-dire de choisir quel fils d'un nœud parcourir en premier lorsque l'on a terminé d'instancier toutes les variables de ce nœud. Nous avons testé différentes heuristiques dynamiques, par exemple à partir de l'activité maximale ou

Algorithme 4.3 SÉPARATIONARBORESCENTE($H(\mathcal{S}, \mathcal{H})$)

1: $r \leftarrow$ CRÉERNŒUD()
2: $\mathcal{N} \leftarrow \{r\}$
3: $\mathcal{A} \leftarrow \emptyset$
4: $\varsigma(r) \leftarrow \mathcal{S}$
5: SÉPARERNŒUD($r, \mathcal{T}(T(\mathcal{N}, \mathcal{A}), r), \varsigma, H(\mathcal{S}, \mathcal{H})$)
6: **retourner** $(\mathcal{T}(T(\mathcal{N}, \mathcal{A}), r), \varsigma)$

moyenne des nœuds suivants possibles. Toutefois nos résultats expérimentaux préliminaires n'ont pas indiqué de différence de performance significative par rapport à un ordre statique arbitraire, que nous avons donc adopté par soucis d'efficacité. Nous suspectons que cette faible différence est encore une fois due à la grande taille des séparateurs et notamment du séparateur à la racine ; si une grande partie de la recherche se situe dans ce nœud, les choix des nœuds suivants auront logiquement un impact limité.

4.5 Résultats expérimentaux

Afin de vérifier expérimentalement l'efficacité de notre heuristique de séparation arborescente et de son utilisation pour ordonnancer les variables d'un algorithme CDCL, nous l'avons implémentée dans la version 2.0 du solveur MINISAT (Eén et Sörensson, 2004). Nous appellerons cette modification MINISEP. Notre heuristique de séparation utilise directement la représentation interne de la formule construite par MINISAT et évite donc la construction explicite d'un hypergraphe. La formule est séparée après les propagations unitaires initiales qui précèdent la première décision. Ces propagations éliminent en moyenne (sur notre ensemble d'instances) environ 0,75% des variables et 11,25% des clauses, ce qui nous permet donc de séparer un hypergraphe légèrement simplifié.

L'heuristique de décision de MINISAT utilise une file de priorité qui ordonne toutes les variables non-instanciées selon leur activité. MINISEP maintient une file de priorité par nœud de la séparation, chaque file contenant uniquement les variables associées à ce nœud. Cela facilite l'obtention de la variable non-instanciée la plus active dans un nœud donné. De plus, comme la complexité de l'insertion ou de la modification d'un élément

dans une file est fonction de la taille de cette file, on peut gagner un peu d'efficacité en séparant cette file en parties indépendantes.

MINISEP et MINISAT ont tous deux été testés sur le même ensemble d'instances utilisé dans la section 4.2 avec une limite d'une heure par instance et par implémentation. Les résultats sont visibles dans le tableau A.1 (en annexe).

Le constat le plus immédiat est que notre heuristique de construction de séparations arborescentes est applicable à la quasi-totalité des instances considérées. En effet, si l'on exclut les 3 instances résolues par les propagations unitaires initiales et qui n'ont donc pas eu à être décomposées, MINISEP a échoué à construire une séparation arborescente sur seulement 5 des 131 instances restantes. Il s'agit exactement des 5 instances dont le nombre de variables après les propagations unitaires initiales est le plus élevé (de 842 195 à 1 685 481). Nous suspectons que MINISEP pourrait être applicable au moins à une partie de ces instances en optimisant davantage les structures de données utilisées pendant la construction de la séparation arborescente ; toutefois, notre implémentation permet en l'état de traiter toutes les instances de notre ensemble de test dont la taille après simplification est égale ou inférieure à 640 012 variables et 2 194 684 clauses. Ces dimensions sont bien au-delà des médianes et moyennes de notre ensemble de test ainsi que de l'ensemble des instances de la compétition SAT 2009 ; notre objectif d'implémenter une méthode de décomposition applicable à des instances de significativement grande taille est donc atteint. De plus, l'extrême simplicité de notre heuristique de séparation arborescente a pour conséquence un temps de construction tout à fait négligeable (au plus un quart de seconde).

Cette simplicité a en contrepartie un impact négatif sur niveau de détail des décompositions obtenues. Sur les 32 instances décomposées par DTREE-ZCHAFF, la taille maximale des nœuds des séparations arborescentes obtenues par MINISEP est de 2,3 à 18,3 fois supérieure (8 fois en moyenne) à la largeur du *dtree* construit par DTREE-ZCHAFF, qui est une borne supérieure à la taille maximale des nœuds du *dtree*. En particulier, le séparateur à la racine de l'arborescence construite par MINISEP, qui se-

lon notre heuristique conditionne la profondeur de l'arborescence, comprend en moyenne 26% des variables du problème (pour une médiane de 21,5%). Pour 32 des instances, l'arborescence ne comporte que 3 nœuds car le séparateur à la racine contient plus de nœuds que chacune des 2 composantes déconnectées qu'il sépare. La grande taille du premier séparateur est un inconvénient, car cela signifie que de nombreuses variables doivent être instanciées avant de pouvoir obtenir une résolution indépendante de sous-problèmes.

Notons néanmoins que dans 60 des 100 instances résolues par MINISEP dans le temps imparti, la résolution a été effectuée sans prendre aucune décision dans un ou plusieurs nœuds de la séparation arborescente. Cela signifie que la stratégie de décomposition implicite a permis dans ces cas de réduire en pratique la taille de l'espace de recherche considéré.

L'impact des séparations arborescentes sur les performances de MINISAT est très variable. Globalement, MINISEP est sensiblement moins performant que l'implémentation MINISAT originale ; il parvient à résoudre légèrement moins d'instances dans le temps imparti (100 au lieu de 104) et les instances résolues le sont un peu plus lentement en moyenne (4 minutes 56 contre 4 minutes 25). MINISEP parvient toutefois à résoudre 34 instances plus rapidement que MINISAT avec une différence d'au moins 1 seconde, dont 4 instances que MINISAT ne parvient pas à résoudre en moins d'une heure.

On peut aussi remarquer que la différence de performances entre MINISAT et MINISEP, dans un sens ou dans l'autre, augmente avec la finesse de la décomposition. C'est particulièrement flagrant dans le cas des instances **dspam**, dont la décomposition est particulièrement détaillée (dans 8 des 10 instances, le plus gros nœud contient moins de 2% des variables). En effet, 3 de ces instances sont résolues en moins de deux secondes par MINISEP, tandis que MINISAT en résout une en plus de deux minutes et prend plus d'une heure pour les deux autres. Au contraire, 4 autres instances de cette famille sont résolues en moins de 2 minutes chacune par MINISAT tandis que MINISEP ne peut les résoudre en moins d'une heure.

Globalement, l'impact de nos séparations arborescentes sur la recherche CDCL

est donc difficile à prévoir et est peu bénéfique en moyenne. Nous pensons que cette faible efficacité est due à deux facteurs contradictoires. D'une part, le fait même de poser des contraintes fermes sur les décisions de CDCL, qui est à la base des stratégies de décomposition implicite, va à l'encontre des heuristiques de décision dynamique dont l'efficacité est reconnue comme supérieure à celle des heuristiques statiques. Malgré les bénéfices que l'on peut attendre du fait de provoquer la séparation de l'instance et d'en résoudre les composantes résiduelles séparément, l'impact négatif sur la liberté des heuristiques de décision, qui est difficile à estimer, peut surpasser ces bénéfices. D'autre part, comme nos séparations heuristiques sont peu détaillées, l'indépendance des sous-problèmes ne pourra être exploitée que tardivement au cours de la recherche. Il semblerait donc que, dans une grande partie des cas, nos séparations arborescentes sont trop peu détaillées pour exploiter efficacement l'indépendance des sous-problèmes, mais suffisamment pour perturber les heuristiques de choix des variables.

4.6 Conclusion

Dans ce chapitre, nous avons présenté une méthode de décomposition d'hypergraphes simple, rapide et applicable en pratique à des problèmes de grande taille. Cette méthode repose sur la notion de séparation arborescente, moins contraignante que les définitions de décomposition arborescente et de *dtree*, et permet d'appliquer la stratégie de décomposition implicite à des instances SAT de grande taille.

L'applicabilité de notre méthode a été prouvée expérimentalement à l'aide d'une implémentation dans un solveur CDCL. Toutefois, la simplicité de l'heuristique de séparation arborescente utilisée rend difficile la réduction du temps de résolution des instances. Étant donné le temps d'exécution quasi-nul de la construction de la séparation arborescence, il serait possible d'utiliser une heuristique de construction plus complexe et de rechercher un équilibre entre la qualité des décompositions obtenues et un contrôle des ressources en temps et en espace nécessaires. On pourrait alors trouver un compromis à mi-chemin entre MINISEP et les autres implémentations de décomposition implicite existantes, suffisamment léger pour être applicable à des instances de taille considérable

mais fournissant des décompositions assez détaillées pour exploiter rapidement l'indépendance des sous-problèmes. Il resterait toutefois à résoudre le problème de la restriction des heuristiques dynamiques de décision, inhérente au principe de décomposition implicite.

Notons que cette problématique de la décomposition de grandes instances est beaucoup moins pertinente dans le cadre du problème CSP. En effet, comme les domaines des variables d'un CSP peuvent contenir un nombre quelconque de valeurs et que chaque contrainte peut éliminer un grand nombre de tuples, l'hypergraphe primal d'un encodage CSP d'un problème donné peut comporter bien moins de sommets et d'hyperarêtes que l'hypergraphe primal d'un encodage SAT de ce même problème. Par exemple, sur les 3 289 instances CSP utilisées lors de la compétition de solveurs de contraintes 2009 (*Constraint Solver Competition*, van Dongen, Lecoutre et Roussel, 2009), la taille moyenne est de seulement 750 variables et 172 contraintes, pour des tailles maximales de 62 704 variables et 4 559 contraintes. Ces tailles de problèmes bien plus réduites permettent donc le plus souvent de décomposer les instances CSP à l'aide d'heuristiques sophistiquées en un temps raisonnable. Cet écart important de taille des hypergraphes caractéristiques des instances, ainsi que la grande efficacité expérimentale des solveurs SAT, nous tendent à penser que les techniques de décomposition implicites ont un meilleur potentiel dans le cadre de la résolution du problème CSP que du problème SAT.

CHAPITRE V

PROPRIÉTÉS DES ALGORITHMES CDCL

La contribution principale de cette thèse est la présentation de variantes de l'algorithme CDCL, le CDCL sans saut arrière et le CDCL à ordre partiel, qui sont introduits par les chapitres 6 et 7 respectivement. Nous verrons que chacun de ces algorithmes admet lui-même différentes variantes. Nous parlerons dorénavant « des algorithmes CDCL » pour désigner l'ensemble des variantes du CDCL, et d'« un algorithme CDCL » pour désigner une variante quelconque parmi cet ensemble. Au contraire, nous utiliserons « le CDCL » ou « l'algorithme CDCL » pour mentionner l'algorithme CDCL original tel que décrit dans la section 2.5. Nous parlerons parfois de « CDCL classique » ou de « CDCL ordinaire » pour éviter certaines ambiguïtés. Sauf cas particulier, nous négligerons généralement la présence ou l'absence du mécanisme de littéraux bloqués dans le CDCL classique.

Ce chapitre a pour objectif d'exposer formellement de nouvelles définitions et propriétés pouvant ou non s'appliquer aux algorithmes CDCL. Ces propriétés nous permettront lors des chapitres suivants de comparer différentes caractéristiques de nos nouvelles variantes avec celles du CDCL classique et du CDCL de type GRASP. Nous verrons en effet que la modification du CDCL peut facilement altérer les propriétés de l'algorithme : le CDCL de type GRASP lui-même n'observe pas les mêmes propriétés que le CDCL classique.

Les sections 5.1 et 5.2 définissent respectivement des propriétés concernant la pro-

cédure de propagation unitaire et, dans le cas de solveurs l'implémentant, le mécanisme de littéraux surveillés. La section 5.3 démontre des relations entre ces deux catégories de propriétés. La section 5.4 introduit différentes propriétés sur le type de conflits rencontrés lors d'une recherche et prouve certaines relations avec les propriétés précédemment énoncées. La section 5.5 propose une interprétation de ces différentes propriétés. La section 5.6 détermine quelles propriétés sont vérifiées par le CDCL classique et le CDCL de type GRASP. Enfin, la section 5.7 est un court résumé des résultats de ce chapitre.

5.1 Propriétés de la procédure de propagation unitaire

La procédure PROPAGER est une composante cruciale des algorithmes CDCL. En effet, si sa fonction première est de trouver les clauses unitaires à propager, c'est également cette procédure qui détecte les éventuelles clauses rendues fausses par l'instanciation actuelle ; elle est donc essentielle à la correction de l'algorithme. On attend d'une procédure de propagation idéale qu'elle détecte exhaustivement toutes les propagations et tous les conflits présents. Nous verrons toutefois que ça n'est pas toujours le cas : dans certains algorithmes CDCL, la procédure de propagation ne peut détecter exhaustivement ni les clauses unitaires ni les conflits. Dans d'autres cas, les conflits sont assurés d'être détectés, mais pas les propagations. Nous définirons donc ici deux niveaux d'exhaustivité :

Définition 5.1. *Un algorithme CDCL est* ***à conflits exhaustifs*** *si, lorsque la procédure de propagation unitaire termine sans détecter de conflit, il ne peut y avoir aucune clause fausse.*

Définition 5.2. *Un algorithme CDCL est* ***à propagations exhaustives*** *si, lorsque la procédure de propagation unitaire termine sans détecter de conflit, il ne peut y avoir aucune clause unitaire ou fausse.*

Il est évident que la première définition est incluse dans la seconde. Par conséquent :

Corollaire 5.1. *Un algorithme CDCL à propagations exhaustives est aussi à conflits exhaustifs.*

Le rapport entre la correction d'un algorithme CDCL et sa procédure de propagation peut être formalisé par la proposition suivante :

Proposition 5.1. *Un algorithme CDCL à conflits exhaustifs est correct.*

Démonstration. Supposons que notre algorithme est à conflits exhaustifs. Si cet algorithme décide que la formule testée est satisfaisable, cela signifie que la dernière procédure de propagation a terminé sans détecter de conflits ; selon notre hypothèse de départ, aucune clause de la formule n'est fausse. De plus, toutes les variables de la formule sont instanciées. On a donc bien un modèle de la formule. Par conséquent, l'algorithme est correct. □

La correction est une propriété essentielle du CDCL, il est naturel de vouloir la préserver dans toutes les variantes de l'algorithme. Cette proposition nous assure donc que la détection exhaustive des conflits est une condition suffisante pour obtenir cette propriété. Il s'agit également d'une condition nécessaire dans la plupart des cas, si l'algorithme n'implémente pas de mécanisme de détection des conflits supplémentaire ; en effet, si un algorithme qui n'est pas à conflits exhaustifs décide que la formule est satisfaisable, cela signifie que toutes les variables sont instanciées et que la dernière procédure de propagation a terminé sans détecter de conflit, mais il peut toutefois exister une clause fausse dans la formule. Pour qu'un algorithme qui n'assure pas la détection exhaustive des conflits soit tout de même correct, il faudrait donc que dans toute trace d'exécution toutes les clauses fausses soient détectées lors du dernier appel à la procédure PROPAGER, ce qui est très peu probable mais difficile à prouver formellement dans le cas général. Nous verrons toutefois, à la fois par des preuves formelles et des résultats expérimentaux, que nos exemples d'algorithmes CDCL sans exhaustivité des conflits, CDCL-SSA$_{\text{LB}}$ (section 6.7) et CDCL-OP$_{\text{LB}}$ (section 7.2), ne sont pas corrects à moins d'ajouter un mécanisme supplémentaire de détection des clauses fausses.

5.2 Propriétés des littéraux surveillés

Toutes les variantes du CDCL présentées dans cette thèse, excepté GRASP, utilisent le mécanisme des littéraux surveillés pour détecter efficacement les clauses unitaires et fausses. Dans ce cas, la façon dont les clauses sont surveillées conditionne évidemment les propriétés de la procédure de propagation. Ce mécanisme assure une propagation exhaustive dans un CDCL classique, tout en minimisant le nombre de modifications des littéraux surveillés. Il est toutefois très sensible aux variations de l'algorithme dans sa globalité, et nos différentes variantes assouplissent plus ou moins les conditions de surveillance des clauses. Ici encore nous définirons deux niveaux de surveillance, qui correspondent intuitivement au nombre de littéraux surveillés valides (c'est-à-dire vrais, indéfinis ou pas encore propagés) par clause.

Définition 5.3. *Une clause est* ***partiellement surveillée*** *si soit au moins un de ses deux littéraux surveillés est vrai, indéfini, ou faux mais pas encore propagé, soit l'algorithme utilise les littéraux bloqués et le littéral bloqué associé à l'un des littéraux surveillés est vrai.*

Rappelons qu'un littéral instancié $l \in \sigma$ (ou son opposé $\neg l$) est dit propagé lorsque la procédure de propagation unitaire a terminé de vérifier toutes les clauses dans lesquelles $\neg l$ est surveillé.

Définition 5.4. *Un algorithme CDCL utilisant les littéraux surveillés est dit* ***à surveillance partielle*** *si toutes les clauses du problème sont partiellement surveillées en tout temps.*

Dans ce contexte, « en tout temps » signifie que la propriété de surveillance est vraie avant et après chaque décision, avant chaque vérification de clause par la procédure PROPAGER, après chaque vérification de clause qui ne provoque pas de conflit et après chaque résolution de conflit.

Définition 5.5. *Une clause est* ***entièrement surveillée*** *si elle est dans l'un des deux cas suivants :*

1. *chacun de ses deux littéraux surveillés est soit vrai, soit indéfini, soit faux mais pas encore propagé, ou bien son littéral bloqué associé est vrai si le mécanisme des littéraux bloqués est utilisé ;*
2. *un de ses deux littéraux surveillés est faux et déjà propagé, et le second est vrai.*

Définition 5.6. *Un algorithme DPLL utilisant les littéraux surveillés est dit* ***à surveillance entière*** *si toutes les clauses du problème sont entièrement surveillées en tout temps.*

Corollaire 5.2. *Une clause entièrement surveillée est aussi partiellement surveillée. Un algorithme à surveillance entière est également à surveillance partielle.*

Démonstration. Dans le premier cas de la définition de surveillance entière, considérons arbitrairement un des deux littéraux surveillés. S'il est vrai, indéfini ou faux mais pas encore propagé, il vérifie le premier cas de la définition de surveillance partielle. Sinon, son littéral bloqué est vrai, et il vérifie donc le second cas de la définition de surveillance partielle. Dans le second cas de la définition de surveillance entière, le littéral surveillé vrai vérifie le premier cas de la définition de surveillance partielle. □

5.3 Relations entre propriétés de la propagation unitaire et des littéraux surveillés

Les deux sections précédentes ont défini des propriétés respectivement sur l'exhaustivité des détections de la procédure de propagation et sur la surveillance des littéraux. Cette nouvelle section prouve les relations entre ces propriétés.

Proposition 5.2. *Un algorithme CDCL utilisant les littéraux surveillés et à surveillance partielle est aussi à conflits exhaustifs.*

Démonstration. Supposons qu'un algorithme surveille toutes les clauses partiellement en tout temps. Supposons qu'une clause devient fausse avant ou pendant une procédure de propagation unitaire. Tous ses littéraux sont faux, y compris les littéraux bloqués

associés aux deux littéraux surveillés si le mécanisme des littéraux bloqués est utilisé. Cependant, puisque la clause est surveillée partiellement, au moins un de ses littéraux surveillés n'est pas encore propagé. La clause fausse sera donc forcément détectée pendant les propagations unitaires si ce littéral est propagé. S'il n'est pas propagé, cela signifie qu'un autre conflit a été détecté avant d'arriver à ce littéral. L'algorithme est donc à conflits exhaustifs. □

Comme dans le cas de la proposition 5.1, la réciproque est certainement vraie mais difficile à démontrer dans le cas général. En effet, si une clause fausse n'est pas surveillée partiellement, cela signifie que ses deux littéraux surveillés sont faux et ont déjà été propagés. Si une procédure de propagation unitaire ne rencontre aucun autre conflit, elle termine alors sans avoir détecté la clause fausse, ce qui prouve la non-exhaustivité des conflits. Pour qu'un algorithme qui n'assure pas la surveillance partielle soit tout de même à conflits exhaustifs, il faudrait alors que dans chaque cas où une clause devient fausse sans être partiellement surveillée, la procédure PROPAGER en cours ou qui suit immédiatement provoque un conflit en détectant une autre clause fausse, ce qui semble improbable. Tous les algorithmes CDCL sans surveillance partielle que nous décrirons dans cette thèse, soit CDCL-SSA$_{\text{LB}}$ (section 6.7) et CDCL-OP$_{\text{LB}}$ (section 7.2), ne sont pas à conflits exhaustifs.

Corollaire 5.3. *Un algorithme CDCL utilisant les littéraux surveillés et à surveillance partielle est correct.*

Démonstration. Par transitivité des propositions 5.1 et 5.2. □

Proposition 5.3. *Un algorithme CDCL utilisant les littéraux surveillés et à surveillance entière est également à propagations exhaustives.*

Démonstration. Supposons qu'un algorithme surveille toutes les clauses entièrement en tout temps. Supposons qu'une clause c devient unitaire avant ou pendant une procédure de propagation unitaire. Puisque c est unitaire, les littéraux bloqués associés aux deux

littéraux surveillés sont faux ou indéfinis si ce mécanisme est utilisé, et au moins un des littéraux surveillés est faux. Soit w_1 ce littéral surveillé faux. Puisque c est entièrement surveillée, w_1 n'est pas encore propagé. Supposons que c est vérifiée par propagation de w_1 ; dans le cas contraire, cela signifie que la procédure de propagation unitaire a rencontré un conflit et que la propriété de propagations exhaustives n'est pas brisée. Si le second littéral surveillé w_2 est le seul littéral indéfini de c, alors w_1 ne peut être remplacé et c est détectée comme unitaire. Sinon, w_2 est faux et n'est pas encore propagé car c est entièrement surveillée et w_1 n'est pas vrai. Le littéral indéfini w_3 est alors surveillé à la place de w_1 et les propagations unitaires continuent. Si aucun conflit n'est découvert entretemps, c est à nouveau vérifiée à partir du littéral surveillé w_2. Puisque le second littéral surveillé est alors w_3, w_2 ne peut être remplacé et c est détectée comme unitaire. La détection d'une clause fausse peut être prouvée de la même façon que pour la proposition 5.2. L'algorithme est donc à propagations exhaustives. □

Ici encore, la réciproque de la propriété a de grandes chances d'être vraie sans qu'il nous soit possible de le prouver formellement dans le cas général. En effet, une clause unitaire qui n'est pas surveillée entièrement ne peut pas être détectée par la procédure de propagation unitaire, car au moins un de ses deux littéraux surveillés w_1 est faux et a déjà été propagé, tandis que le second littéral surveillé w_2 est indéfini ou faux. Si w_2 est indéfini et le reste jusqu'à la fin de la procédure de propagation unitaire, ou s'il est faux et déjà propagé, c ne sera pas vérifiée, et donc pas détectée. Si w_2 est faux mais pas encore propagé, alors l, le seul littéral non-faux de c, est différent de w_1 et w_2. Si la procédure de propagation unitaire ne détecte pas de conflit auparavant, c sera vérifiée lors de la propagation de w_2. l sera alors surveillé à la place de w_2, et comme w_1 a déjà été propagé, la clause unitaire c ne sera pas détectée.

Par conséquent, pour qu'un algorithme sans surveillance entière des clauses soit tout de même à propagations exhaustives, il faudrait que pour toute clause qui devient unitaire sans être entièrement surveillée, l'appel suivant ou en cours de PROPAGER déclenche un conflit qui défasse un des littéraux surveillés, ou alors que son seul littéral

indéfini soit instancié de façon à satisfaire la clause avant la fin de cet appel à PROPAGER. Là encore, un tel concours de circonstances systématique est hautement improbable. En pratique, nous verrons que les deux algorithmes que nous présenterons et qui sont à surveillance partielle mais non entière, CDCL-SSA et CDCL-OP, ne sont pas à propagations exhaustives (voir sections 6.4 et 7.4 respectivement).

En résumé, on peut donc partitionner les algorithmes CDCL utilisant les littéraux surveillés en trois catégories mutuellement exclusives :

1. Les algorithmes sans surveillance partielle, probablement sans conflits exhaustifs ;
2. Les algorithmes à surveillance partielle et conflits exhaustifs, mais sans surveillance entière et probablement sans exhaustivité des propagations ;
3. Les algorithmes à surveillance entière et propagations exhaustives.

Les algorithmes de la première catégorie ne sont probablement pas corrects, à moins de comporter une méthode additionnelle assurant la correction. Les algorithmes des catégories 2 et 3 sont corrects, et cette correction est assurée uniquement par la surveillance partielle des clauses.

5.4 Propriétés des conflits

Les conflits constituent un aspect central des algorithmes CDCL. Ils permettent notamment d'enrichir la connaissance du problème en dérivant une clause de conflit. Il est donc souhaitable que cette clause de conflit soit différente de toutes les clauses déjà connues. Nous verrons que, dans certaines conditions, l'assurance d'une clause de conflit inédite dépend de la quantité de littéraux situés au niveau de conflit.

Définition 5.7. *Un conflit est* ***trivial*** *si la clause fausse l'ayant déclenché a un seul littéral au niveau de conflit. Un conflit est* ***redondant*** *si la clause de conflit qu'il produit est identique à une clause originale ou précédemment apprise.*

Définition 5.8. *Un algorithme CDCL est* ***non-trivial*** *s'il ne peut déclencher aucun conflit trivial. Il est* ***non-redondant*** *s'il ne peut déclencher aucun conflit redondant.*

Proposition 5.4. *Tout conflit non-redondant dans un algorithme utilisant la stratégie d'apprentissage du premier point d'implication unique est non-trivial.*

Démonstration. Si un conflit est trivial, la clause fausse c qui l'a déclenché ne contient qu'un seul littéral au niveau de conflit. Par conséquent, lors de l'analyse de c par l'algorithme 2.9, la boucle **tant que** commençant à la ligne 4 n'est pas exécutée. La clause de conflit γ est donc strictement identique à la clause fausse c; le conflit est redondant. Cela prouve la proposition par contraposée. □

Corollaire 5.4. *Si un algorithme CDCL utilisant la stratégie d'apprentissage du premier point d'implication unique est non-redondant, il est aussi non-trivial.*

Si la stratégie d'apprentissage est différente, la proposition 5.4 est fausse : certains conflits triviaux peuvent être non-redondants. Par exemple, la stratégie d'apprentissage du dernier point d'implication unique requiert que la clause apprise contienne la décision du niveau de conflit. Si la clause fausse d'un conflit trivial ne contient pas cette décision mais une propagation du même niveau, au moins une résolution sera appliquée à cette clause, et la clause de conflit sera donc potentiellement différente de toutes les clauses déjà connues.

Le corollaire 5.4 restera toutefois généralement vrai : en effet, si un algorithme est trivial, il restera sans doute toujours quelques cas de conflits où aucune résolution n'est effectuée durant l'analyse, peu importe la stratégie d'apprentissage utilisée. Par conséquent, il est peu réaliste d'envisager un algorithme CDCL qui soit non-redondant sans être non-trivial. Il est cependant difficile de prouver ce corollaire indépendamment de la stratégie d'apprentissage utilisée; il est généralement nécessaire d'exhiber un contre-exemple particulier, c'est-à-dire un conflit trivial qui est également redondant en utilisant la stratégie d'apprentissage considérée. Nous nous restreignons ici au cas du premier point d'implication unique car il s'agit de la stratégie la plus répandue et aucune variante du CDCL présentée dans cette thèse n'utilise de stratégie différente, hormis GRASP.

Le corollaire 5.4 implique donc qu'avec cette stratégie d'apprentissage, la non-trivialité des conflits est une condition nécessaire pour éviter les conflits redondants. La proposition 5.5, quant à elle, démontre que la non-trivialité de l'algorithme requiert l'exhaustivité des propagations, quelque soit la stratégie d'apprentissage utilisée.

Proposition 5.5. *Si un algorithme CDCL est non-trivial, alors il est à propagations exhaustives.*

Démonstration. Considérons un algorithme CDCL qui n'est pas à propagations exhaustives. Cela signifie qu'à la fin d'une procédure de propagation unitaire qui n'a pas détecté de conflit, il peut rester une clause c unitaire. Soit l le littéral indéfini dans c. L'étape suivante de l'algorithme est de choisir une nouvelle décision. Supposons que cette décision est $\neg l$. La décision rend la clause c fausse, ce qui est détecté lors de la propagation de $\neg l$. Supposons que c est la première clause fausse détectée lors de cette propagation. $\nu(l)$ est donc l'unique variable instanciée au niveau de décision courant de la clause fausse. Par conséquent, le conflit est trivial, donc l'algorithme n'est pas non-trivial, ce qui prouve la proposition par contraposée. □

La preuve de cette proposition suppose que dans tout algorithme CDCL, lors de tout conflit survenant immédiatement après une nouvelle décision, le niveau de conflit est le niveau de cette décision. C'est effectivement le cas dans toutes les variantes de CDCL que nous décrivons dans cette thèse. Nous supposons également que le choix de la prochaine décision ainsi que l'ordre de vérification des clauses surveillées sont non-déterministes, ce qui implique qu'à partir de n'importe quelle occurrence de clause unitaire omise par PROPAGER on peut dériver un conflit trivial. Ce non-déterminisme est vrai dans les descriptions algorithmiques du CDCL et de ces variantes, mais il ne se retrouve généralement pas dans les implémentations, puisque le choix des décisions est effectué par des heuristiques et l'ordre de vérification des clauses est généralement déterminé par la structure de donnée utilisée pour garder la trace des clauses où un littéral donné est surveillé.

La réciproque de la proposition 5.5 est fausse : la section 6.8 fournit des exemples d'algorithmes à propagations exhaustives qui ne sont pourtant pas non-triviaux. La proposition 5.6 démontre que cette réciproque est toutefois vraie dans le cas d'algorithmes à niveau de propagation unique, que nous définissons ci-après. Pour la prouver, nous aurons également besoin du lemme 5.1.

Définition 5.9. *Un algorithme CDCL est dit* ***à niveau de propagation unique*** *si, lors de toute exécution de* PROPAGER, *il existe un niveau de décision courant λ_c tel que toute instanciation qui n'est pas encore propagée est au niveau de décision λ_c, et tout conflit découvert est au même niveau λ_c.*

Le CDCL classique, GRASP et le CDCL à ordre partiel (chapitre 7) sont à niveau de propagation unique, contrairement au CDCL sans saut arrière (chapitre 6).

Lemme 5.1. *Si un algorithme CDCL est à propagations exhaustives et à niveau de propagation unique, aucune clause n'est unitaire ou fausse après une résolution de conflit, exceptée la nouvelle clause de conflit qui est unitaire.*

Démonstration. Par récurrence sur les conflits.

Case de base : Supposons que le premier conflit de l'exécution ne se déroule pas au pseudo-niveau de décision 0 (sinon la formule est insatisfaisable). Nommons i le $i^{\text{ème}}$ niveau de décision créé, et soit n le niveau de décision où survient le premier conflit, $n \geq 1$. La décision au niveau n a suivi une procédure de propagation au niveau $n-1$ qui a terminé sans rencontrer de conflit. Puisque l'algorithme est à propagations exhaustives, aucune clause n'est unitaire ou fausse avant la décision du niveau n. Comme l'algorithme est à niveau de propagation unique, l'appel à PROPAGER suivant la décision propage uniquement des littéraux au niveau n, car au début de cet appel la décision n'est pas encore propagée. Lors de la résolution du conflit, le niveau n est entièrement défait, donc toutes les propagations ayant suivi la $n^{\text{ième}}$ décision sont défaites. Aucune clause ne contient alors plus de littéraux instanciés qu'avant cette décision. En particulier, aucune clause n'est unitaire ou fausse, hormis la nouvelle clause de conflit.

Récursion : Supposons qu'aucune clause n'est unitaire ou fausse après le $k^{\text{ième}}$ conflit, exceptée la clause de conflit associée γ_k, qui contient un littéral au nouveau niveau de décision courant λ_c. Si une ou plusieurs décisions sont prises entre le $k^{\text{ième}}$ et le $k+1^{\text{ième}}$ conflit, alors on peut prouver la propriété de la même façon que pour le cas de base, en s'appuyant sur le fait qu'aucune clause n'est unitaire ni fausse juste après la décision précédant le $k+1^{\text{ième}}$ conflit.

Si aucune décision n'est prise entre le $k^{\text{ième}}$ et le $k+1^{\text{ième}}$ conflit, cela signifie que le $k+1^{\text{ième}}$ conflit intervient dès le premier appel à PROPAGER après le $k^{\text{ième}}$ conflit. Au début de cet appel, l'assertion résultant du $k^{\text{ième}}$ conflit est instanciée au niveau λ_c mais pas encore propagée. Comme l'algorithme est à niveau de propagation unique, toutes les propagations effectuées lors de la procédure sont instanciées au niveau λ_c. Lors de la résolution de conflit, le niveau λ_c est entièrement défait ; en particulier, toutes les instanciations ayant suivi le $k^{\text{ième}}$ conflit sont défaites. Aucune clause ne contient alors plus de littéraux qu'avant ce conflit. En particulier, aucune clause n'est unitaire ou fausse, hormis la nouvelle clause de conflit γ_{k+1}, car la clause du conflit précédent γ_k contenait après propagation de l'assertion deux littéraux de niveau λ_c. □

Dans cette preuve, nous supposons que tout algorithme CDCL défait entièrement le niveau de conflit lors d'une résolution de conflit. Il est possible d'imaginer des algorithmes pouvant ne défaire que partiellement le niveau de conflit, toutefois ce n'est le cas d'aucune des variantes décrites dans cette thèse.

Proposition 5.6. *Si un algorithme CDCL est à propagations exhaustives et à niveau de propagation unique, alors cet algorithme est non-trivial.*

Démonstration. Considérons un conflit dans un algorithme CDCL à propagations exhaustives et à niveau de propagation unique. La procédure de propagation ayant détecté le conflit a suivi soit une décision, soit un conflit antérieur (sinon, cela signifie que le premier conflit détecté est au niveau 0, ce qui termine l'algorithme en prouvant l'insatisfaisabilité de la formule).

Dans le premier cas, aucune clause n'était unitaire ou fausse avant la décision, puisque l'algorithme est à propagations exhaustives. Toute clause a donc au moins deux littéraux indéfinis. Puisque l'algorithme est à niveau de propagation unique, tous les littéraux propagés après la dernière décision sont du même niveau λ_c que cette décision. λ_c est également le niveau de conflit. Par conséquent, si une clause fausse est détectée, elle contient au moins deux littéraux du niveau de conflit : le conflit est non-trivial.

Dans le second cas, supposons que le conflit que nous considérons est le $k^{\text{ième}}$ conflit de la recherche, $k \geq 2$. L'appel à PROPAGER ayant provoqué ce conflit a immédiatement suivi la résolution du $k-1^{\text{ième}}$ conflit et l'instanciation de son assertion. D'après le lemme 5.1, après la résolution du $k-1^{\text{ième}}$ conflit, la seule clause unitaire était la clause de conflit correspondante, dont un des littéraux était du niveau d'assertion du $k-1^{\text{ième}}$ conflit. Soit i ce niveau. Comme l'algorithme est à niveau de propagation unique, toutes les propagations effectuées lors de l'appel à PROPAGER qui a provoqué le $k^{\text{ième}}$ conflit ont été instanciées au niveau i. En effet, au début de cet appel, l'assertion du $k-1^{\text{ième}}$ conflit est instanciée au niveau i et pas encore propagée. Par conséquent, la clause fausse déclenchant le $k^{\text{ième}}$ conflit contient au moins 2 littéraux du niveau de conflit i. Le conflit est donc non-trivial.

Puisque tout conflit est non-trivial, l'algorithme est non-trivial. □

Nous avons donc montré qu'un algorithme CDCL est non-trivial s'il est à propagations exhaustives et à niveau de propagation unique. On peut remarquer qu'au contraire, un algorithme CDCL qui n'est pas à niveau de propagation unique a peu de chances d'être non-trivial, peu importe l'exhaustivité des propagations. En effet, cette non-unicité du niveau de propagation signifie que la procédure de propagation ayant précédé un conflit a pu effectuer des propagations à divers niveaux de décision. En particulier, la clause fausse détectée peut ne contenir qu'un seul littéral au niveau de conflit, auquel cas le conflit est trivial. Pour rester non-trivial, un algorithme à niveau de propagation non-unique devrait donc garantir que toute clause fausse détectée par la procédure de propagation unitaire a au moins deux littéraux au niveau de conflit, ce qui

semble difficilement réalisable. Nous verrons que parmi toutes les variantes de CDCL sans saut arrière, un algorithme à niveau de propagation non-unique que nous définirons dans le chapitre 6, aucune n'est non-triviale.

Nous allons maintenant montrer que la non-trivialité d'un algorithme CDCL à niveau de propagation unique est une condition suffisante pour assurer sa non-redondance.

Proposition 5.7. *Si un algorithme CDCL à niveau de propagation unique est non-trivial, alors il est non-redondant.*

Démonstration. D'après la proposition 5.5, l'algorithme est à propagations exhaustives, puisqu'il est non-trivial. D'après le lemme 5.1, après la résolution d'un conflit, la clause de conflit nouvellement apprise est la seule à être unitaire. Elle est donc différente de toutes les autres clauses du problème. Par conséquent le conflit est non-redondant. Puisque cette démonstration s'applique à tous les conflits de l'exécution, l'algorithme est non-redondant. □

Corollaire 5.5. *Soit un algorithme CDCL à niveau de propagation unique utilisant la technique d'apprentissage du premier point d'implication unique. Alors les trois propriétés suivantes sont équivalentes :*

1. *L'algorithme est non-trivial ;*
2. *L'algorithme est non-redondant ;*
3. *L'algorithme est à propagations exhaustives.*

Démonstration. Le corollaire 5.4 et la proposition 5.7 prouvent l'équivalence de la non-trivialité et de la non-redondance dans ces conditions. Les propositions 5.4 et 5.5 prouvent quant à elles l'équivalence de la non-trivialité et de l'exhaustivité des propagations. □

5.5 Interprétation des propriétés

Les propriétés énoncées et reliées dans les sections précédentes nous permettent non seulement de comparer les différentes variantes de CDCL entre elles, mais aussi d'interpréter la qualité de chaque algorithme.

L'exhaustivité de la procédure de propagation unitaire apparaît comme un élément clé des propriétés de l'algorithme tout entier. En particulier, la détection exhaustive des conflits assure la correction de l'algorithme. Dans le cas d'algorithmes à littéraux surveillés, l'exhaustivité de la procédure dépend logiquement de la qualité de surveillance de chaque clause, la détection exhaustive des propagations requérant une surveillance plus forte que la détection exhaustive des conflits.

L'utilité des conflits est une propriété qui n'est pas essentielle à la correction de l'algorithme, mais qui est toutefois importante pour son efficacité. En effet, si un conflit est redondant, cela signifie que la partie de la recherche qui a conduit à ce conflit aurait pu être évitée par inférence à partir des clauses connues précédemment. De plus, la clause de conflit étant déjà connue, elle ne contribue pas à l'élagage de l'espace de recherche. Éviter les conflits redondants est donc une propriété souhaitable d'un algorithme CDCL, qui peut être assurée en interdisant les conflits triviaux. Les conflits triviaux eux-mêmes sont évitables en assurant l'exhaustivité des propagations et l'unicité de leur niveau. Un algorithme sans propagation à niveau unique peut difficilement être non-trivial, et donc non-redondant. Toutefois, même dans ce cas, maintenir l'exhaustivité des propagations peut contribuer à réduire le nombre de conflits redondants, comme cela est démontré expérimentalement dans le chapitre 6.

5.6 Propriétés du CDCL classique et de GRASP

Dans cette section, nous allons analyser les algorithmes du CDCL classique et de GRASP à la lumière des propriétés décrites dans les sections précédentes. Nous verrons que si le CDCL classique vérifie les meilleures propriétés possibles, ce n'est pas le cas de

GRASP, ce qui peut partiellement expliquer la différence d'efficacité constatée entre les deux algorithmes.

Proposition 5.8. *CDCL est à surveillance entière, avec ou sans gestion des littéraux bloqués.*

Démonstration. Nous allons tout d'abord prouver la surveillance entière sans gestion des littéraux bloqués.

La propriété est trivialement vraie au début de l'exécution, puisqu'aucune variable n'est instanciée : toutes les clauses sont surveillées par deux littéraux indéfinis et se trouvent donc dans le premier cas de la définition 5.5.

Lorsqu'un littéral l est instancié, cela ne peut modifier la surveillance que pour les clauses où l ou $\neg l$ est surveillé. Toutes ces clauses appartiennent avant l'instanciation au premier cas. Elles restent dans ce premier cas, puisqu'on y remplace un littéral indéfini par un littéral vrai ou par un littéral faux mais pas encore propagé.

Lors de la vérification d'une clause c pendant la propagation de son littéral faux l, c est initialement dans le premier cas. Si l est remplacé par un nouveau littéral surveillé indéfini ou vrai, c reste dans le premier cas. Sinon, soit w le second littéral surveillé de c. Si w est déjà vrai, ou si c est unitaire et w est instancié à vrai, l n'est pas remplacé et c passe dans le second cas. Comme l'ordre des niveaux de décision est fixé chronologiquement et que le CDCL est à niveau de propagation unique, on a $\lambda(w) \leq \lambda(l)$ pour toute clause dans le second cas, où w est le littéral surveillé vrai et l est le littéral surveillé faux.

Si l ne peut pas être remplacé et w est faux, alors un conflit est rencontré et un saut arrière est déclenché. Lors d'un saut arrière, les clauses du premier cas restent dans le premier cas si un ou deux littéraux surveillés y sont désinstanciés. Les clauses du second cas passent dans le premier cas si le littéral surveillé faux est désinstancié ou si les deux littéraux surveillés sont désinstanciés. Elles pourraient passer dans un cas incorrect si le littéral surveillé vrai était désinstancié mais pas le littéral surveillé faux. Cependant, le saut arrière du CDCL défait les niveaux de décision dans l'ordre

inversement chronologique, et tout niveau de décision défait est défait entièrement. Par conséquent, puisque le niveau de décision du littéral surveillé faux est toujours supérieur ou égal au niveau du littéral surveillé vrai, il ne peut être conservé alors que le littéral surveillé vrai est défait. Par conséquent, toutes les clauses restent entièrement surveillées lors de l'exécution sans littéraux bloqués.

Si les littéraux bloqués sont utilisés, la surveillance d'une clause c peut être modifiée si le littéral bloqué $\beta(c, w)$ associé à l'un des deux littéraux surveillés $w \in \omega(c)$ devient vrai. Dans ce cas, après l'instanciation de $\beta(c, w)$, $\neg w$ peut être instancié et propagé sans que le littéral surveillé w soit remplacé dans c, quelque soient les valeurs du second littéral surveillé de c et de son littéral bloqué associé. Cependant, on a $\lambda(\beta(c, w)) \leq \lambda(w)$, donc si $\beta(c, w)$ est désinstancié par un saut arrière, $\neg w$ sera également désinstancié. La surveillance entière de c ne peut donc pas être rompue par le mécanisme des littéraux bloqués dans le CDCL classique. Celui-ci est donc bien à surveillance entière, avec ou sans utilisation des littéraux bloqués. □

Corollaire 5.6. *Le CDCL classique, avec ou sans gestion des littéraux bloqués, est à propagations exhaustives, non-trivial et non-redondant.*

Démonstration. Comme le CDCL classique est à surveillance entière, d'après la proposition 5.8, il est aussi à propagations exhaustive. L'exhaustivité des propagations implique la non-trivialité et la redondance par le corollaire 5.5, puisque le CDCL classique est à niveau de propagation unique et utilise l'apprentissage par premier point d'implication unique. □

Proposition 5.9. *GRASP est à conflits exhaustifs.*

Démonstration. La surveillance des clauses par mise à jour systématique de compteurs de littéraux vrais et faux assure cette exhaustivité de la détection des conflits. □

Proposition 5.10. *GRASP n'est pas non-trivial ni non-redondant.*

Démonstration. Nous allons montrer cette propriété sur un contre-exemple qui provoque un conflit à la fois trivial et redondant.

Considérons une formule propositionnelle comportant les 5 variables a, b, c, d et e et les 6 clauses $c_1 = \neg a \vee \neg c \vee d$, $c_2 = \neg a \vee \neg c \vee \neg d$, $c_3 = \neg b \vee \neg d \vee e$, $c_4 = \neg b \vee \neg d \vee \neg e$, $c_5 = \neg b \vee d \vee e$ et $c_6 = \neg b \vee d \vee \neg e$. Supposons la séquence de décisions suivante :

- Le littéral a est instancié comme décision au niveau 1.
- Le littéral b est instancié comme décision au niveau 2.
- Le littéral c est instancié comme décision au niveau 3. Les clauses c_1 et c_2 sont désormais unitaires. Supposons que c_1 est détectée la première. Le littéral d est alors instancié comme propagation au niveau 3.
- La clause c_2 est désormais fausse. La décision du niveau 3, c, n'a pas encore été inversée, donc GRASP calcule la clause de conflit $c_7 = \neg a \vee \neg c$ à l'aide de la stratégie du premier point d'implication unique, puis défait uniquement le niveau 3 et instancie $\neg c$ comme pseudo-décision du niveau 3.
- Le littéral d est instancié comme décision au niveau 4. Les clauses c_3 et c_4 sont désormais unitaires. Supposons que c_3 est détectée la première. Le littéral e est alors instancié comme propagation au niveau 4.
- La clause c_4 est désormais fausse. La décision du niveau 4, d, n'a pas encore été inversée, donc GRASP calcule la clause de conflit $c_8 = \neg b \vee \neg d$ à l'aide de la stratégie du premier point d'implication unique, puis défait uniquement le niveau 4 et instancie $\neg d$ comme pseudo-décision du niveau 4.
- Les clauses c_5 et c_6 sont désormais unitaires. Supposons que c_5 est détectée la première. e est alors instancié comme propagation au niveau 4.
- La clause c_6 est maintenant fausse. La pseudo-décision du niveau 4, $\neg d$, a déjà été inversée, donc GRASP effectue un saut arrière. La clause de conflit est $c_9 = \neg b$, de niveau maximal 2 ; les niveaux de décision 3 et 4 sont donc défaits.
- Comme c_9 est toujours fausse, elle déclenche un nouveau conflit, qui est trivial puisque c_9 est une clause unaire. La décision du niveau 2, b, n'a pas encore été inversée, donc GRASP calcule la clause de conflit $c_{10} = c_9 = \neg b$ à l'aide de la

stratégie du premier point d'implication unique. Ce conflit est donc également redondant.

□

En plus de cette preuve formelle, nous avons également vérifié expérimentalement l'existence de conflits triviaux en utilisant l'implémentation originale de GRASP.

Corollaire 5.7. *GRASP n'est pas à propagations exhaustives.*

Démonstration. Par la propriété 5.10, GRASP n'est pas non-trivial. Or, il est à niveau de propagation unique. Il n'est donc pas à propagations exhaustives, par contraposée de la propriété 5.6. □

La non-exhaustivité des propagations dans GRASP peut aussi se prouver par un contre-exemple, mais celui-ci est plus long que la démonstration de la propriété 5.10.

Cette preuve de l'existence de conflits redondants dans GRASP et de leur absence dans le CDCL classique fournit une explication supplémentaire des meilleures performances de ce dernier en pratique. Cette supériorité avait été tout d'abord démontrée par un dispositif expérimental implémentant à la fois les stratégies de saut arrière et d'apprentissage de chaque algorithme (Zhang et al., 2001), où elle était avant tout expliquée par le fait que contrairement au CDCL, GRASP apprend en général plusieurs clauses par conflit, ce qui permet un meilleur élagage de l'espace de recherche mais ralentit significativement les propagations unitaires puisqu'un plus grand nombre de clauses doit être vérifié pour chaque instanciation à propager. Nos résultats sur l'utilité des conflits nous poussent donc à considérer que la stratégie de pseudo-décisions est également en partie responsable de la moins bonne efficacité de GRASP, puisqu'elle empêche la découverte de certaines clauses unitaires et provoque des conflits qui auraient pu être évitées par un élagage plus complet de l'espace de recherche.

5.7 Résumé

Dans ce chapitre, nous avons défini différentes propriétés liées à l'exhaustivité des propagations et conflits détectés au cours des propagations unitaires, à l'intégrité des littéraux surveillés, ainsi qu'à la trivialité des conflits et à l'originalité des clauses apprises. Nous avons établi différents liens d'implication entre ces propriétés, ainsi qu'avec la correction de l'algorithme dans son ensemble. Enfin, nous avons déterminé les propriétés vérifiées par le CDCL classique et par GRASP. Nous avons constaté que les propriétés de GRASP étaient plus faibles que celles du CDCL classique, ce qui contribue à expliquer la meilleure efficacité observée en pratique pour ce dernier.

CHAPITRE VI

CDCL SANS SAUT ARRIÈRE

Ce chapitre présente l'algorithme de CDCL sans saut arrière, conçu pour réduire la quantité d'instanciations détruites par l'étape de saut arrière de l'algorithme CDCL. Cette variation du CDCL part du constat que le saut arrière jusqu'au niveau d'assertion n'est pas essentiel à la correction de l'algorithme. Ce saut arrière est plutôt un héritage des origines de recherche en profondeur de l'algorithme, et est une façon de simplifier la gestion en assurant que toutes les propagations et conflits se déroulent en tout temps au niveau de décision maximal. Cependant, tous les niveaux de décision défaits par le saut arrière, excepté le niveau maximal où le conflit a eu lieu, n'ont par définition aucun rapport direct avec le conflit ; on peut donc considérer que le saut arrière a pour conséquence de défaire inutilement une partie parfois non-négligeable de l'instanciation courante construite jusqu'ici par l'algorithme.

Les différentes variations de CDCL introduites dans ce chapitre proposent par conséquent de supprimer cette étape de saut arrière. Plus précisément, chaque conflit entraînera uniquement la désinstanciation des variables du niveau de conflit.

Supprimer l'étape de saut arrière permet donc de minimiser la quantité d'informations détruites lors de la résolution d'un conflit. Nous verrons qu'en contrepartie, cette modification complique la gestion globale de l'algorithme : les propagations, et donc les conflits, peuvent survenir à n'importe quel niveau de décision. Cela implique entre autres que lorsque le niveau de conflit n'est pas le niveau maximal, la résolution du conflit doit

non seulement défaire le niveau de conflit, mais également détecter et défaire les variables d'autres niveaux propagées directement ou indirectement à l'aide du niveau de conflit. De plus, ces modifications altèrent les propriétés de l'algorithme : il n'est pas possible d'éviter l'occurrence de conflits triviaux et redondants, et obtenir une exhaustivité des propagations ou même des conflits requiert certaines conditions, voire des manipulations supplémentaires.

La section 6.1 décrit informellement le concept du CDCL sans saut arrière. La section 6.2 discute plus en détail son implémentation. La section 6.3 compare le CDCL sans saut arrière avec les autres stratégies de réduction de la destructivité des sauts arrière introduites dans le chapitre 3. La section 6.4 prouve les propriétés du CDCL sans saut arrière : cet algorithme est totalement correct, il détecte exhaustivement les conflits mais pas les propagations unitaires. La section 6.5 propose différentes méthodes de résolution de conflit. La section 6.6 compare expérimentalement l'efficacité du CDCL sans saut arrière à celle du CDCL classique. La section 6.7 présente l'introduction des littéraux bloqués dans le CDCL sans saut arrière, montre qu'ils provoquent la perte de l'exhaustivité des conflits et évalue expérimentalement leur impact sur les performances de l'algorithme. La section 6.8 décrit et compare expérimentalement différentes stratégies de détection des propagations alternatives, et montre notamment qu'elles permettent l'exhaustivité des propagations unitaires, toutefois sans éviter les conflits triviaux et redondants. La section 6.9 propose une variante de ces stratégies de détection qui peut éviter certaines désinstanciations lors des résolutions de conflits et l'évalue expérimentalement. Enfin, la section 6.10 conclut le chapitre en résumant ses résultats.

6.1 Concept de base

Le but premier du saut arrière dans l'algorithme CDCL classique est de résoudre le conflit rencontré, c'est-à-dire de revenir à un état consistant de la recherche où la clause qui a déclenché le conflit n'est plus fausse. Or, pour cela, il suffit simplement de défaire le niveau de conflit ; c'est d'ailleurs la stratégie utilisée par DPLL. Si CDCL effectue un saut arrière jusqu'au niveau d'assertion, c'est dans le but de propager la clause de

conflit au niveau de décision le plus bas possible, afin d'élaguer un maximum l'espace de recherche, tout en évitant la destruction future de propagations toujours valides qui survient dans GRASP (voir section 5.6). Or, dans le CDCL classique, toute propagation est instanciée au niveau de décision courant, car elle a au moins un antécédent à ce niveau de décision courant. Si GRASP contourne le problème en ignorant cette dernière condition, le CDCL classique s'y conforme en défaisant toutes les instanciations dont le niveau de décision est ultérieur au niveau d'assertion. La clause de conflit contient alors bien au moins un littéral instancié au nouveau niveau courant. En contrepartie, cette destruction de niveaux de décision sans rapport direct avec le conflit est la source de la destructivité du mécanisme de saut arrière, expliquée dans la sous-section 2.5.9.

Effectuer toutes les propagations au niveau de décision courant a un avantage pratique certain : pour trouver toutes les instanciations à propager, il suffit de parcourir dans l'ordre la trace des instanciations à ce niveau, et lorsqu'un conflit est déclenché toute cette file de propagations est invalidée en même temps que le niveau maximal est détruit. Cependant, la stratégie du saut arrière de CDCL peut aussi s'interpréter comme une contrainte implicite de dépendance totale entre les différents niveaux de décision, due à l'aspect de recherche en profondeur hérité du DPLL, qui suppose que toute décision prise à un instant donné, ainsi que ses conséquences, dépend de toutes les décisions prises antérieurement. Ainsi, lorsque l'algorithme revient à un niveau de décision antérieur pour y propager la clause de conflit, il est alors contraint de défaire tous les niveaux de décision qui le suivent. En effet, selon cette hypothèse de dépendance systématique, le changement apporté au niveau d'assertion provoquera possiblement un déroulement différent de la recherche lors des niveaux de décision suivants.

La variante du CDCL que nous proposons, le CDCL sans saut arrière (ou CDCL-SSA), consiste essentiellement à permettre d'effectuer une nouvelle propagation unitaire à un niveau de décision quelconque. En particulier, lors d'un conflit, il n'est pas nécessaire de défaire tous les niveaux de décision ultérieurs au niveau d'assertion pour propager la clause de conflit ; seul le niveau de conflit et ses conséquences doivent être désinstanciés afin de résoudre le conflit. Le CDCL-SSA n'effectue donc pas de réel saut

arrière : toute résolution de conflit défait une et une seule décision, qui est celle du niveau de conflit. Aucune autre décision ne peut être défaite, puisqu'une décision ne dépend d'aucune autre instanciation. Nous conservons cependant un ordre total sur les niveaux de décision, correspondant à l'ordre d'instanciation de leur décision. Une propagation unitaire se fait donc au plus grand des niveaux de décision des antécédents de la propagation. Plus formellement, soit $c = \{l_1, l_2, \ldots, l_{n-1}, l_n\}$ une clause unitaire telle que $\sigma(l_1) = \sigma(l_2) = \ldots = \sigma(l_{n-1}) =$ faux et $\sigma(l_n) =$ indéfini. Alors l_n sera propagé au niveau $\lambda(c) = \max\{\lambda(l) \mid l \in c \setminus \{l_n\}\}$. En particulier, l'assertion d'un conflit est toujours propagée au niveau d'assertion.

Comme les conflits surviennent à la suite de propagations unitaires, le CDCL-SSA permet l'apparition de conflits où le niveau de conflit, soit le niveau maximal dans la clause fausse, n'est pas le niveau maximal courant. Par conséquent, lors de la gestion d'un conflit dans le cadre de cet algorithme, il sera nécessaire de défaire non seulement le niveau de conflit, mais aussi toutes les propagations des niveaux de décision supérieurs dont un des antécédents appartient au niveau de conflit, puis récursivement toutes les autres propagations dont un des antécédents a été défait.

La figure 6.1 donne un exemple d'exécution du CDCL sans saut arrière sur la même instance utilisée dans les sous-sections 2.4.4 et 3.4.6 pour illustrer respectivement les algorithmes DPLL et CDCL. Nous supposerons ici que les mêmes décisions sont prises dans le même ordre jusqu'au premier conflit. L'exécution jusqu'à ce conflit est strictement identique à celle du CDCL ordinaire ; l'état atteint est représenté par la figure 6.1a. Comme celui-ci, le CDCL sans saut arrière désinstancie le niveau de conflit et effectue la propagation de l'assertion au niveau d'assertion. La différence est que le CDCL sans saut arrière, contrairement au CDCL ordinaire, ne défait pas les niveaux situés entre le niveau de conflit et le niveau d'assertion. Les décisions a et b sont donc conservées (voir figure 6.1b). Par conséquent, l'assertion $\neg y$ n'est pas propagée au niveau de décision maximal.

Par la suite, $\neg y$ pourra provoquer des propagations à des niveaux de décision

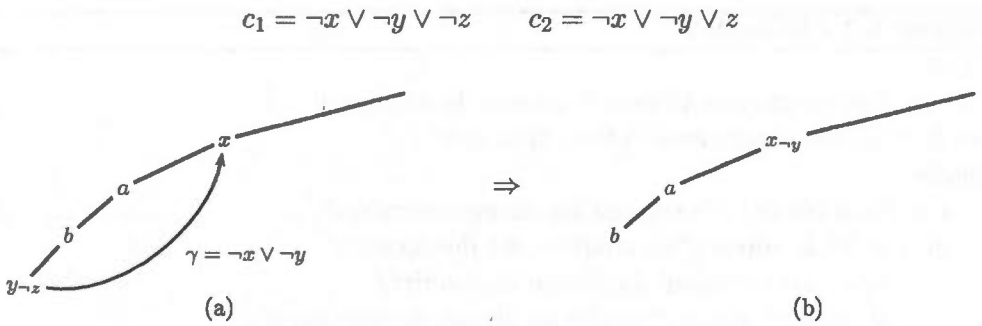


FIGURE 6.1: Exemple d'exécution du CDCL sans saut arrière. La figure 6.1a décrit l'arbre de recherche produit par CDCL-SSA sur la formule $\mathcal{F}(\{a, b, x, y, z\}, \{c_1, c_2\})$ après les décisions consécutives x, a, b et y et la propagation de $\neg z$ par la clause unitaire c_1, ainsi que la clause γ résultant de l'analyse du conflit déclenché par la clause fausse c_2 et le niveau d'assertion de la clause unitaire γ. La figure 6.1b représente l'arbre de recherche après la résolution du conflit.

divers. Supposons par exemple que la formule propositionnelle contient également les variables d, e et f ainsi que les clauses $c_3 = y \vee d$, $c_4 = y \vee \neg a \vee e$ et $c_5 = y \vee \neg a \vee \neg b \vee f$; alors, la vérification de $\neg y$ par le mécanisme de propagation unitaire provoquera la propagation des littéraux d, e et f par les clauses c_4, c_5 et c_6 aux niveaux de décision $\lambda(x)$, $\lambda(a)$ et $\lambda(b)$ respectivement.

6.2 Implémentation du CDCL sans saut arrière

Cette section aborde l'implémentation de l'algorithme de CDCL sans saut arrière présenté informellement dans la section précédente. Le pseudo-code global de l'algorithme est présenté dans la sous-section 6.2.1, tandis que les sous-sections 6.2.2 et 6.2.3 détaillent certains points plus précis de l'implémentation, respectivement la structure de données utilisée pour gérer la trace des instanciations et la représentation de la file des propagations en attente.

6.2.1 Pseudo-code global

L'algorithme 6.1, CDCL-SSA, décrit le pseudo-code global du CDCL sans saut arrière. Il y a relativement peu de différences avec le pseudo-code du CDCL original

Algorithme 6.1 CDCL-SSA

```
1: Λ ← 0
2: λ_c ← 0  /*on commence au pseudo-niveau de décision 0*/
3: σ ← ∅  /*on commence avec l'affectation vide*/
4: boucle
5:     c ←PROPAGER() /*propager les clauses unitaires*/
6:     si c ≠ NUL alors /*un conflit a été découvert*/
7:         λ_γ ← λ(c) /*calcul du niveau de conflit*/
8:         si λ_γ = 0 alors /*conflit au niveau de décision 0*/
9:             retourner faux  /*F est insatisfaisable*/
10:        sinon
11:            γ ← ANALYSER(c)  /*déduire la clause de conflit γ*/
12:            a ← l ∈ γ | λ(a) = λ_γ
13:            /*a est unique ; c'est la future assertion*/
14:            RÉSOUDRECONFLIT(λ_γ)
15:            /*défaire λ_γ et les propagations qui en dépendent*/
16:            C ← C ∪ {γ}  /*γ est apprise*/
17:            PROPAGERASSERTION(a, γ)
18:    sinon /*aucun conflit pendant les propagations*/
19:        si D(σ) = V alors /*toutes les variables sont instanciées*/
20:            retourner σ  /*σ est un modèle de F*/
21:        sinon
22:            λ_c ← NOUVEAUNIVEAU()
23:            Λ ← Λ ∪ {λ_c}
24:            choisir v ∈ V\D(σ)  /*v est une variable non-instanciée*/
25:            choisir l ∈ {v, ¬v}
26:            INSTANCIER(l, NUL)
27:            /*l est une décision, il n'a pas d'antécédent*/
```

(algorithme 2.8). La seule différence visible à ce niveau de détail est qu'il est nécessaire de calculer explicitement le niveau de conflit à la ligne 7, car le niveau d'un conflit n'est pas forcément le niveau courant. Cette définition du niveau de conflit est également la seule différence dans la procédure ANALYSER (algorithme 6.2, lignes 4 et 6). Les procédures PROPAGER et PROPAGERASSERTION sont identiques aux versions du CDCL original (algorithmes 2.15 et 2.12 respectivement).

La clause de conflit obtenue par la procédure d'analyse est toujours notée γ. Le niveau de conflit, noté λ_γ, est le plus grand niveau de décision présent dans γ. Contrairement au CDCL classique, il n'y a dans CDCL-SSA pas de notion de niveau

Algorithme 6.2 ANALYSER(c) *[CDCL-SSA]*

1: /*c est la clause fausse détectée pendant les propagations unitaires*/
2: /*γ est la clause de conflit générée par l'analyse*/
3: $\gamma \leftarrow c$
4: **tant que** $|\{l \in \gamma \,|\, \lambda(l) = \lambda_\gamma\}| > 1$ **faire**
5: /*il reste plus d'un littéral de niveau maximal dans γ*/
6: $l \leftarrow$ DERNIER(γ, λ_γ)
7: /*l est le littéral de niveau λ_γ dans γ instancié le plus récemment*/
8: $\gamma \leftarrow \gamma \otimes_{\nu(l)} \alpha(l)$ /*résolution de γ et $\alpha(l)$ sur la variable de l*/

d'assertion, puisque pas de saut arrière. Le niveau de décision de l'assertion propagée après résolution du conflit est défini, au travers des procédures PROPAGERASSERTION et INSTANCIER, comme le niveau de décision maximal dans la clause de conflit γ après désinstanciation entière du niveau de conflit λ_γ. Le SAUTARRIÈRE est lui remplacé par RÉSOUDRECONFLIT, dont deux stratégies d'implémentation sont décrites par la section 6.5.

6.2.2 Trace des instanciations

L'implémentation originale de CDCL utilise une structure de données simple et efficace pour mémoriser l'ordre dans lequel les variables ont été instanciées et leur organisation en niveaux de décision. Les instanciations sont simplement enregistrées dans l'ordre dans un vecteur unidimensionnel. Un second vecteur contient une liste de pointeurs vers les décisions consécutives. Lors d'un saut arrière, il suffit alors de détruire entièrement la fin du vecteur d'instanciations à partir de la décision du premier niveau de décision à défaire, ainsi que la fin du vecteur de pointeurs de décisions. Le saut arrière est donc implémenté très efficacement dans le CDCL classique.

On ne peut utiliser une implémentation aussi simple pour le CDCL sans retour arrière, puisqu'il permet de revenir à un niveau de décision antérieur et d'y rajouter de nouvelles instanciations. Garder la même structure de données nécessiterait dans ce cas de décaler les instanciations des niveaux supérieurs et de mettre à jour les pointeurs vers les décisions de ces niveaux à chaque nouvelle instanciation à un niveau de décision

non-maximal, ce qui alourdirait considérablement la gestion. Nous avons contourné ce problème en utilisant une structure de données bidimensionnelle, soit un vecteur de vecteurs. Chaque vecteur interne représente un niveau de décision, qui peut donc être mis à jour sans incidence sur les autres niveaux. Notons qu'à aucun moment le CDCL sans retour arrière ne requiert d'insérer un nouveau niveau de décision entre deux niveaux existants.

Lorsqu'un niveau de décision non-maximal est entièrement effacé, nous laissons un vecteur vide à son emplacement pour éviter un déplacement des niveaux supérieurs et la mise à jour correspondante. Similairement, lorsqu'une propagation est effacée dans un niveau de décision, nous marquons simplement l'emplacement correspondant du niveau comme vide afin de ne pas avoir à déplacer les propagations suivantes dans le même niveau. Si ce procédé risque en théorie d'entraîner une surconsommation d'espace mémoire, nous avons constaté qu'en pratique la place occupée par les espaces vides est négligeable. De plus, cette tactique est bien plus efficace en temps de calcul que le déplacement systématique des littéraux suivants, que nous avons implémenté et évalué lors de tests préliminaires.

6.2.3 Gestion des propagations

Dans le CDCL classique, la liste des instanciations qui n'ont pas encore été propagées est simplement représentée par un pointeur sur une instanciation dans la trace unidimensionnelle. En effet, dans cet algorithme, toutes les instanciations à propager sont forcément au niveau de décision courant. Il suffit donc de parcourir ce niveau entièrement, dans l'ordre dans lequel les variables sont instanciées, jusqu'à atteindre la fin de la trace ou un conflit. Si un conflit survient, le niveau maximal est défait, et par conséquent toutes les propagations en attente sont invalidées.

La gestion des propagations est plus complexe dans le cadre d'un CDCL sans saut arrière. Entre autres, un conflit n'implique pas que toutes les variables en attente d'être propagées sont défaites : en effet, l'algorithme peut déclencher des propagations à un

niveau arbitraire, qui elles-mêmes peuvent entraîner de nouvelles propagations au même niveau ou à un niveau supérieur. Or, le seul niveau de décision entièrement défait par la résolution de conflit du CDCL sans saut arrière est le niveau de conflit. Il convient donc de gérer une file d'instanciations à propager par niveau de décision, c'est-à-dire, pour chaque niveau, un pointeur vers la première instanciation pas encore totalement propagée dans ce niveau.

Notre implémentation examine les instanciations à propager dans l'ordre croissant de leur niveau de décision, dans le but de favoriser la détection de conflits à des niveaux de décision bas. L'ordre de parcours des instanciations à propager n'a cependant pas d'incidence sur la correction de l'algorithme. De plus, notre ordre de parcours ne nous garantit pas de détecter les propagations unitaires ou les conflits de plus bas niveau possible en premier. En effet, lorsqu'une clause est examinée parce qu'un de ses littéraux surveillés w_1 est faux, elle peut contenir d'autres variables instanciées à des niveaux de décision plus grands. Dans ce cas, si cette clause s'avère unitaire ou fausse, le niveau de propagation ou le niveau de conflit est donc plus grand que $\lambda(w_1)$.

6.3 Comparaison avec d'autres stratégies pour limiter la destructivité des sauts arrière

CDCL-SSA est en quelque sorte l'équivalent du retour arrière dynamique pour le problème SAT. En effet, les deux algorithmes ont pour point commun de limiter la destructivité de la résolution de conflit en désinstanciant uniquement une seule décision et ses conséquences (éliminations de valeurs dans le cas de DBT, propagations unitaires dans le cas de CDCL-SSA). De plus, les variables défaites par DBT et CDCL-SSA ont un rôle similaire : DBT défait la variable coupable, la dernière variable, dans l'ordre d'instanciation, impliquée dans l'élimination des valeurs d'une variable dont le domaine est vidé, tandis que CDCL-SSA défait la décision du niveau de conflit, la dernière décision impliquée dans une clause rendue fausse. La différence essentielle est que la résolution de conflits de CDCL-SSA provoque des propagations unitaires à des niveaux de décision quelconques, ce qui n'est pas possible dans un CDCL ordinaire et cause une modification

des propriétés de l'algorithme (voir section 6.4).

Du fait de la similarité entre CDCL-SSA et DBT, les différences entre CDCL-SSA et POB sont comparables aux différences entre DBT et POB : dans CDCL-SSA, le niveau de conflit est toujours défini comme le dernier niveau de décision instancié à être impliqué dans le conflit, tandis que POB permet un choix potentiellement arbitraire de la variable coupable parmi toutes les variables impliquées. Toutefois, contrairement à POB, CDCL-SSA n'impose aucune contraintes sur l'ordre de choix des variables à décider.

CDCL-SSA remplit l'objectif des stratégies de résolution indépendante des sous-problèmes connexes, car la résolution de conflit dans une composante connexe du sous-problème résiduel ne peut défaire aucune instanciation dans les autres composantes, avec l'avantage de ne pas nécessiter de détection explicite des sous-problèmes connexes. CDCL-SSA permet de plus de limiter la destructivité des sauts arrière à l'intérieur des composantes connexes. En fait, alors que les stratégies de résolution indépendante des sous-problèmes connexes tiennent compte de la connectivité du graphe résiduel, et donc des relations futures potentielles entre les variables pas encore instanciées, CDCL-SSA considère au contraire uniquement les relations effectives entre les variables déjà instanciées, c'est-à-dire l'ensemble des clauses qui ont été utilisées par des propagations unitaires, qui est un sous-ensemble de la connectivité du graphe résiduel avant leur instanciation.

Par rapport aux décompositions implicites de problèmes SAT, CDCL-SSA a l'avantage d'économiser la construction d'une décomposition arborescente de l'instance et de ne poser aucune contrainte sur l'heuristique de choix des variables, ainsi que d'éviter une plus grande proportion des destructions d'instanciations par rapport à un saut arrière classique. Toutefois, comme les techniques de détection des composantes connexes, CDCL-SSA n'améliore pas la complexité théorique par rapport CDCL.

Enfin, CDCL-SSA est plus complexe que la sauvegarde de phase, mais cette dernière, si elle permet de retrouver une partie des instanciations défaites par un saut arrière,

n'évite pas le surcoût dû à la désinstanciation, puis à la réinstanciation de la variable défaite, ainsi que des propagations unitaires en partie répétées à l'identique. CDCL-SSA permet aussi de détruire toutes les variables du niveau de conflit, qui risqueraient de redéclencher un conflit similaire, alors que la sauvegarde de phase peut récupérer toutes les instanciations défaites sans distinction.

6.4 Propriétés du CDCL sans saut arrière

Cette section établit les propriétés théoriques du CDCL sans saut arrière, c'est-à-dire à la fois les propriétés « classiques » de correction, complétude et terminaison et les propriétés spécifiques aux variantes du CDCL introduites dans le chapitre 5. Nous verrons notamment que le CDCL sans saut arrière conserve la détection exhaustive des conflits, mais pas la détection exhaustive des propagations, ce qui permet l'occurrence de conflits triviaux et redondants.

Proposition 6.1. *CDCL-SSA est complet, termine et a une complexité temporelle exponentielle au pire des cas.*

Démonstration. Les preuves des propositions 2.6, 2.7 et 2.8, qui démontrent ces propriétés pour le CDCL classique, restent valides pour le CDCL-SSA, ainsi que pour toutes les variantes du CDCL-SSA étudiées dans la suite de ce chapitre. □

Proposition 6.2. *CDCL-SSA est à surveillance partielle.*

Démonstration. La propriété est trivialement vraie au début de la recherche, puisqu'aucune variable n'est instanciée.

L'instanciation d'un littéral l préserve cet invariant : elle n'a aucune incidence sur les clauses où ni l, ni $\neg l$ n'est un littéral surveillé. Si l est un littéral surveillé dans une clause, alors cette clause reste partiellement surveillée puisque l est vrai ; si $\neg l$ est surveillé dans une clause, alors elle reste également partiellement surveillée puisque $\neg l$ est faux, mais pas encore propagé.

La propagation d'une instanciation l n'a d'incidence que sur les clauses où $\neg l$ est surveillé. Soit c une telle clause et w le second littéral surveillé de la clause. Si w est vrai, la clause reste partiellement surveillée ; sinon, l'algorithme cherche à remplacer $\neg l$ par un autre littéral de c indéfini ou vrai. Si un tel littéral existe, c reste partiellement surveillée ; sinon si w est indéfini, il est propagé par c qui reste partiellement surveillée puisque w devient vrai. Si w est faux, c est détectée comme une clause fausse, un conflit est déclenché et l reste considéré comme non-propagé ; c est donc toujours partiellement surveillée, même si l n'est pas défait par la résolution du conflit. □

Corollaire 6.1. *CDCL-SSA est totalement correct et à conflits exhaustifs.*

Démonstration. D'après la proposition 6.2, CDCL-SSA est à surveillance partielle ; il est donc à conflits exhaustifs selon la proposition 5.2, cette propriété impliquant sa correction d'après la proposition 5.1. Puisque la proposition 6.1 montre que CDCL-SSA est également complet et termine, l'algorithme est donc totalement correct. □

Ces différentes propositions montrent que le CDCL sans saut arrière conserve les propriétés essentielles de CDCL que sont la correction, la complétude et la terminaison. De plus, la procédure de propagation unitaire y détecte exhaustivement tous les conflits. La surveillance des clauses est toutefois de qualité inférieure à celle du CDCL ordinaire, et implique que certaines clauses unitaires ne sont pas détectées, comme le démontre la proposition suivante.

Proposition 6.3. *CDCL-SSA n'est pas à surveillance entière ni à propagations exhaustives.*

Démonstration. Prouvons la proposition par contre-exemple. Considérons une formule propositionnelle contenant 7 variables nommées a, b, c, d, e, f et g ainsi que les 6 clauses suivantes : $c_1 = \neg a \vee \neg d \vee \neg e$, $c_2 = \neg a \vee \neg d \vee e$, $c_3 = \neg b \vee d \vee f$, $c_4 = \neg c \vee d \vee f$, $c_5 = \neg a \vee \neg b \vee \neg f \vee \neg g$ et $c_6 = \neg a \vee \neg b \vee \neg f \vee g$. Soit σ l'instanciation courante,

initialement vide. Supposons que chaque clause est initialement surveillée par ses deux premiers littéraux et que les opérations suivantes sont effectuées :

- Le littéral a est instancié par décision au niveau 1. Après propagation de a, les clauses c_1 et c_2 sont surveillées respectivement par $\{\neg d, \neg e\}$ et $\{\neg d, e\}$, c_5 et c_6 sont toutes deux surveillées par $\{\neg b, \neg f\}$.
- Le littéral b est instancié par décision au niveau 2. Après propagation de b, c_3 est surveillée par $\{d, f\}$.
- Le littéral c est instancié par décision au niveau 3. Après propagation de c, c_4, c_5 et c_6 sont surveillées respectivement par $\{d, f\}$, $\{\neg f, \neg g\}$ et $\{\neg f, g\}$.
- Le littéral d est instancié par décision au niveau 4, c_1 et c_2 deviennent toutes deux unitaires et sont détectées pendant la propagation de d.
- Supposons que c_1 est détectée la première ; alors $\neg e$ est instancié par propagation au niveau 4, ce qui rend c_2 fausse. La clause de conflit est $c_7 = \neg a \vee \neg d$, les niveaux 4 et 1 sont respectivement le niveau de conflit et le niveau d'assertion. Les littéraux d et $\neg e$ sont défaits, $\neg d$ est instancié par propagation de c_7 au niveau 1. Toutes les clauses sont toujours entièrement surveillées avant que $\neg d$ soit propagé.
- Les clauses c_3 et c_4 sont toutes deux unitaires et détectées lors de la propagation de $\neg d$; supposons que c_3 est détectée la première. Le littéral f est instancié par propagation au niveau 2, c_3 et c_4, qui sont toutes deux surveillées par $\{d, f\}$, restent donc entièrement surveillées. La propagation de $\neg d$ est terminée.
- Les clauses c_5 et c_6 sont toutes deux unitaires et détectées pendant la propagation de f ; supposons que c_5 est détectée la première. Le littéral $\neg g$ est instancié par propagation unitaire au niveau 2, ce qui rend c_6 fausse. La clause de conflit est $c_8 = \neg a \vee \neg b$, de niveau de conflit 2 et de niveau d'assertion 1. Avant résolution du conflit, c_4 est entièrement surveillée car d est faux et propagé mais f est vrai.
- Le conflit est résolu en désinstanciant b et $\neg f$, puis en instanciant $\neg b$ par propagation de c_8 au niveau 1. La clause c_4 n'est maintenant plus entièrement

surveillée, puisque d est faux et déjà propagé et f est indéfini ; CDCL-SSA n'est donc pas à surveillance entière. La propagation de $\neg b$ ne provoque aucune vérification de clause, puisque b n'est surveillé dans aucune clause. La procédure de propagation unitaire termine donc sans détecter c_4, qui est unitaire. Par conséquent, CDCL-SSA n'est pas à propagations exhaustives.

□

Comme GRASP, le CDCL sans saut arrière ne permet donc pas d'assurer la détection exhaustive de toutes les clauses unitaires. Les causes en sont cependant différentes : dans GRASP, l'omission des propagations unitaires est due à la présence de propagations à des niveaux de décision strictement supérieurs aux niveaux de tous leurs littéraux antécédents, ce qui implique qu'un saut arrière conventionnel peut défaire cette propagation sans défaire aucun des antécédents. Dans le cas du CDCL sans saut arrière, toutes les propagations unitaires sont instanciées exactement au niveau de décision le plus élevé parmi leurs littéraux antécédents, et toute propagation unitaire ne peut être défaite que si au moins un autre littéral de son antécédent est défait. Toutefois, il peut subsister après la résolution de conflit une autre clause unitaire pouvant servir de nouvel antécédent à la propagation défaite, mais qui ne sera pas détectée par le mécanisme de propagation ; c'est le cas qui est illustré par la preuve de la proposition 6.3. Un tel cas ne peut pas survenir dans un algorithme CDCL classique, car l'antécédent retenu pour une propagation a toujours le niveau de décision le plus petit parmi tous les antécédents possibles pour cette propagation. Comme le CDCL défait les instanciations dans l'ordre inverse de leur niveau de décision et défait tous les niveaux de décision entièrement, si une propagation unitaire est défaite, alors il ne peut rester aucune clause unitaire dont l'unique littéral indéfini est cette propagation défaite.

Cette surveillance seulement partielle des clauses a pour conséquence que le CDCL sans saut arrière ne peut empêcher les conflits triviaux ou redondants, comme le prouve la proposition suivante :

Proposition 6.4. *CDCL-SSA n'est pas non-trivial, ni non-redondant.*

Démonstration. Reprenons la preuve de la proposition 6.3 et supposons que le littéral $\neg f$ est instancié comme décision au niveau 4. Cela entraîne la vérification des clauses c_3 et c_4, toutes deux surveillées par f. La clause c_3 est satisfaite par le littéral $\neg b$ mais c_4 est fausse et comporte un seul littéral, $\neg f$, à son niveau de décision maximal, soit le niveau 4. Le conflit est donc trivial, ce qui prouve que CDCL-SSA n'est pas non-trivial. Comme l'algorithme utilise la stratégie d'apprentissage du premier point d'implication unique, la clause apprise est identique à c_4, ce qui montre que le conflit est redondant et que CDCL-SSA n'est pas non-redondant. □

En résumé, dans le CDCL sans saut arrière, les propagations unitaires peuvent ne pas être détectées, ce qui rend possible des conflits triviaux et redondants. Cependant, au moins un des deux littéraux surveillés de chaque clause reste valide. Par conséquent, tous les conflits sont détectés, même si certains auraient pu être évités par des propagations exhaustives.

La section 6.8 propose des variantes du CDCL sans saut arrière qui permettent d'assurer l'exhaustivité des propagations. Nous verrons que cela ne suffit malheureusement pas à éviter les conflits triviaux et redondants, bien que cela réduise leur proportion en pratique.

6.5 Stratégies de résolution des conflits

La phase de résolution des conflits du CDCL sans saut arrière nécessite d'identifier toutes les instanciations dépendant de la décision du niveau de conflit, directement ou non. Les sous-sections 6.5.1 et 6.5.2 proposent deux stratégies d'implémentation différentes pour cette résolution de conflit, respectivement par construction d'un graphe de dépendances entre variables et par parcours intégral des instanciations de niveaux supérieurs.

6.5.1 Représentation explicite des dépendances entre instanciations

Algorithme 6.3 INSTANCIER(l, c) *[CDCL-SSA-Graphe]*

Requis : $\nu(l) \notin \mathcal{D}(\sigma)$

1: $v \leftarrow \nu(l)$
2: $\sigma(v) \leftarrow \rho(l)$
3: $\alpha(v) \leftarrow c$
4: **si** $c = \text{NUL}$ **alors** /*si l est une décision*/
5: $\quad \lambda(v) \leftarrow \lambda_c$
6: **sinon** /*si l est une propagation*/
7: $\quad \lambda(v) \leftarrow \lambda(c)$
8: $\quad$ **pour** $l' \in c \setminus \{l\}$ **faire**
9: $\quad\quad \Delta \leftarrow \Delta \cup \{(\nu(l'), \nu(l))\}$ /*l dépend de l'*/

Algorithme 6.4 DÉFAIRE(v) *[CDCL-SSA-Graphe]*

1: $\sigma(v) \leftarrow$ indéfini
2: $\lambda(v) \leftarrow$ indéfini
3: $\alpha(v) \leftarrow$ indéfini
4: *propagé*$(v) \leftarrow$ faux
5: $\Delta \leftarrow \Delta \setminus \{(v, v'), (v', v) \mid v' \in \mathcal{V}\}$

La phase de résolution de conflit du CDCL sans saut arrière consiste à défaire toutes les instanciations du niveau de conflit, puis récursivement défaire toutes les propagations qui dépendaient d'une ou plusieurs instanciations préalablement défaites. Cette phase nécessite donc de connaître les relations de causalité entre les différentes instanciations. Avec les structures de l'algorithme CDCL classique, il est possible de retrouver l'ensemble des variables dont dépend directement ou indirectement une propagation, par le parcours récursif des antécédents des propagations ; cependant, pour résoudre un conflit dans CDCL-SSA, nous avons besoin de l'information inverse, c'est-à-dire de trouver, pour une instanciation donnée, la liste des propagations qui dépendent directement ou indirectement de cette instanciation.

Nous avons conçu une première implémentation naïve (nommée CDCL-SSA-Graphe) qui maintient un graphe orienté $G_\Delta(\mathcal{V}, \Delta)$ où $\Delta \subseteq \mathcal{V} \times \mathcal{V}$ représente les dépendances entre variables : lorsqu'une propagation unitaire a lieu, on ajoute un arc orienté de chaque antécédent vers la variable propagée (algorithme 6.3).[1] Ces arcs sont

1. Notons que dans le cas d'une propagation unitaire, INSTANCIER détermine son niveau de dé-

défaits lorsque la variable précédemment propagée est désinstanciée (algorithme 6.4). Comme la relation Δ est irréflexive et asymétrique, sa fermeture transitive, notée Δ^+ ou $\prec_\Delta$, est un ordre partiel strict sur les variables représentant les dépendances récursives de leurs instanciations actuelles : si $v_1 \prec_\Delta v_2$, alors il existe un ensemble de littéraux actuellement instanciés $\{l_1, l_2, \ldots, l_i\} \subseteq \sigma$, $i \geq 2$, tels que $\nu(l_1) = v_1$, $\nu(l_i) = v_2$ et $\forall k \in 1, \ldots, i-1$, $l_i \in \alpha(l_{i+1})$.

La résolution du conflit (algorithme 6.5) s'effectue en défaisant tout d'abord la décision du niveau de conflit, puis, récursivement, en défaisant toutes les propagations dépendant d'instanciations précédemment défaites, ce qui équivaut donc à défaire toutes les instanciations qui lui sont supérieures selon Δ^+.

L'inconvénient principal de cette implémentation est la lourdeur de la gestion du graphe explicite. Il est potentiellement très grand puisqu'il peut contenir toutes les variables du problème et de nombreux antécédents par propagation (soit un nombre d'arcs quadratique selon le nombre de variables au pire des cas). Ce graphe doit être mis à jour à chaque propagation unitaire ainsi qu'à chaque résolution de conflit. De plus, le parcours du sous-graphe enraciné à la décision du niveau de conflit est en pratique inefficace, car chaque variable à défaire peut avoir plusieurs antécédents eux-mêmes défaits, on va donc la rencontrer plusieurs fois par différents chemins. Notons qu'il est également possible de stocker directement le graphe de la fermeture transitive de la relation, $G_{\Delta^+}(\mathcal{V}, \Delta^+)$, ce qui facilite la recherche des variables à défaire lors de la résolution de conflit mais alourdit énormément la mise à jour du graphe lors d'une propagation ; nous nous restreindrons donc à la version de l'algorithme qui mémorise uniquement G_Δ.

6.5.2 Parcours des niveaux de décision supérieurs

Il est possible d'effectuer la résolution des conflits sans recourir à une représentation explicite des dépendances entre instanciations. Au vu de la définition du niveau de

---

cision en fonction du niveau maximal dans son antécédent, alors que toute nouvelle instanciation est systématiquement assignée au niveau courant dans le CDCL classique (voir algorithme 2.13).

Algorithme 6.5 RÉSOUDRECONFLIT(λ_γ) *[CDCL-SSA-Graphe]*

```
1: Θ ← ∅  /*variables défaites pendant le traitement*/
2: Γ ← {δ(λ_γ)}  /*variables à défaire*/
3: /*Γ contient initialement la décision du niveau de conflit*/
4: tant que Γ ≠ ∅ faire
5:     choisir v ∈ Γ
6:     DÉFAIRE(v)
7:     pour v' ∈ V | v ≺_Δ v', v' ∉ Γ ∪ Θ faire
8:         Γ ← Γ ∪ {v'}
9:     Θ ← Θ ∪ {v}
10:    Γ ← Γ \ {v}
11: Λ ← Λ \ λ_γ  /*λ_γ est le seul niveau entièrement défait*/
```

Algorithme 6.6 RÉSOUDRECONFLIT(λ_γ) *[CDCL-SSA-Parcours]*

```
1: pour λ_γ ≤ i ≤ λ_c faire
2:     /*on parcourt tous les niveaux à partir du niveau de conflit*/
3:     pour l ∈ σ | λ(l) = i faire
4:         /*parcours des littéraux du niveau dans l'ordre d'instanciation*/
5:         si i = λ_γ alors
6:             /*défaire toutes les instanciations du niveau de conflit*/
7:             DÉFAIRE(l)
8:         sinon si ∃l' ∈ α(l) | σ(l') = indéfini alors
9:             /*défaire l si un de ses antécédents a été défait*/
10:            DÉFAIRE(l)
11: Λ ← Λ \ λ_γ  /*λ_γ est le seul niveau entièrement défait*/
```

décision d'une propagation, on peut noter que si une variable a dépend d'une variable b, alors $\lambda(a) \geq \lambda(b)$. Par conséquent, toutes les instanciations défaites lors d'une résolution de conflit sont forcément d'un niveau supérieur ou égal au niveau de conflit. De plus, l'implémentation de la trace des instanciations est telle que les instanciations d'un même niveau de décision sont conservées dans l'ordre chronologique. Par conséquent, si a et b sont du même niveau et a dépend de b, alors a est stockée après b. Pour résoudre un conflit, il suffit alors de procéder de la façon suivante :

- Défaire le niveau de conflit en entier ;
- Parcourir tous les niveaux de décision supérieurs au niveau de conflit dans l'ordre croissant ;
- Pour chaque niveau, parcourir toutes les propagations dans l'ordre dans lequel

Algorithme 6.7 INSTANCIER(l, c) *[CDCL-SSA-Graphe]*

Requis : $\nu(l) \notin \mathcal{D}(\sigma)$

1: $v \leftarrow \nu(l)$
2: $\sigma(v) \leftarrow \rho(l)$
3: $\alpha(v) \leftarrow c$
4: **si** $c = \text{NUL}$ **alors** /*si l est une décision*/
5: $\quad \lambda(v) \leftarrow \lambda_c$
6: **sinon** /*si l est une propagation*/
7: $\quad \lambda(v) \leftarrow \lambda(c)$

elles sont conservées ;

- Défaire les propagations si et seulement si un ou plusieurs de leurs antécédents ont préalablement été défaits.

Nous annoterons cette variante par CDCL-SSA-Parcours ; le pseudo-code correspondant à la phase de résolution du conflit est décrit par l'algorithme 6.6. INSTANCIER (algorithme 6.7) est implémentée de la même façon que pour CDCL-SSA-Graphe (algorithme 6.3), à la différence près qu'il n'est pas nécessaire de gérer le graphe G_Δ. DÉFAIRE est implémentée de la même façon que pour un CDCL classique (algorithme 2.14). Ce procédé a l'inconvénient de visiter des propagations qui ne seront pas défaites. En contrepartie, il ne visite chaque propagation qu'une seule fois et, surtout, il évite toute gestion explicite des dépendances entre les instanciations. Le parcours potentiellement plus long des instanciations a lieu à chaque conflit, tandis que maintenir une représentation explicite des dépendances nécessite des opérations supplémentaires à chaque propagation ; or, les propagations sont bien plus fréquentes que les conflits.

6.6 Évaluation expérimentale du CDCL sans saut arrière

Cette section a pour objet l'évaluation expérimentale de l'efficacité du CDCL sans saut arrière par rapport au CDCL classique, et compare également l'utilisation des deux stratégies de résolution de conflit pour le CDCL sans saut arrière introduites dans la section 6.5. La sous-section 6.6.1 détaille le dispositif expérimental utilisé et présente les diverses données expérimentales recueillies, puis la sous-section 6.6.2 propose une interprétation de ces résultats expérimentaux.

6.6.1 Présentation de l'évaluation et des résultats

Afin de vérifier expérimentalement l'efficacité du CDCL sans saut arrière et des deux variantes proposées de résolution de conflit, nous avons implémenté les différentes variantes de notre algorithme en modifiant la version 1.0 du solveur CDCL GLUCOSE (Audemard et Simon, 2009). Ce solveur est lui-même une modification de MINISAT (Eén et Sörensson, 2004) et a fait partie des solveurs les plus performants lors des dernières compétitions SAT. Notons que GLUCOSE utilise l'heuristique de sauvegarde de phase décrite dans la section 3.3. Comme le CDCL sans saut arrière a été conçu comme une alternative à cette heuristique, nous l'avons désactivée dans toutes nos modifications. Des résultats préliminaires nous indiquaient de toute façon que la combinaison de la sauvegarde de phase avec le CDCL sans saut arrière dégrade généralement les performances de ce dernier.

Toutes les expériences présentées dans ce chapitre ont été réalisées sur une sélection de 62 instances parmi les 300 instances applicatives utilisées lors de la compétition SAT 2011. Nous avons essayé de sélectionner une ou plusieurs instances représentatives pour chaque famille d'instances de cet ensemble, tout en cherchant à privilégier les instances de difficulté moyenne (dont le temps d'exécution est supérieur à quelques secondes mais inférieur à une heure) pour l'implémentation originale de GLUCOSE. La liste des instances retenues est disponible dans le tableau A.2 (en annexe).

L'efficacité de nos deux variantes de CDCL sans saut arrière a été évaluée en les exécutant chacune sur ces 62 instances avec une limite d'une heure par exécution. Les résultats obtenus ont été comparés avec ceux de l'exécution de l'implémentation originale de GLUCOSE dans les mêmes conditions. Nous avons également testé une quatrième variante, strictement identique à l'implémentation originale de GLUCOSE, à l'exception de la désactivation des littéraux bloqués. Cette variante nous permettra d'évaluer l'importance de la désactivation des littéraux bloqués dans les performances des deux variantes du CDCL sans saut arrière, qui n'utilisent pas ce mécanisme.

Dans un soucis de compacité, nous ne présenterons pas les résultats détaillés de chaque exécution individuelle d'une implémentation sur une instance, mais uniquement pour chaque implémentation des résultats agglomérés sur toutes les instances (ou une partie d'entre elles). Le tableau 6.1, notamment, fournit pour chaque implémentation des informations agglomérées sur l'ensemble des instances pour nos deux variantes de CDCL classique et nos deux variantes de CDCL sans saut arrière, ainsi que pour d'autres variantes du CDCL sans saut arrière que nous décrirons ultérieurement au cours du chapitre.

Pour chaque implémentation testée, les deux premières colonnes indiquent respectivement le nombre total d'instances résolues à l'intérieur de la limite d'une heure par exécution, ainsi que le temps total d'exécution de l'implémentation sur l'ensemble des instances, y compris celles pour lesquelles la résolution n'a pas été accomplie dans le temps imparti.

La troisième colonne, quant à elle, fournit le nombre total, sur l'ensemble des instances, de clauses vérifiées pendant la procédure PROPAGER en raison de la propagation d'un de ses littéraux surveillés. La vérification de clauses est en effet l'étape la plus élémentaire de la procédure PROPAGER, et c'est dans cette procédure qu'est généralement passé l'essentiel du temps d'exécution du CDCL classique. Nous considérerons donc ce nombre total de clauses vérifiées, que nous appellerons le nombre d'**étapes de propagation**, comme un indicateur de la difficulté de résolution indépendant de l'implémentation en elle-même et de l'efficacité en temps des différentes opérations. Lorsque nous parlerons de la longueur ou de la taille d'une exécution, nous parlerons du nombre d'étapes de propagation qu'elle contient.

La quatrième colonne indique ce que nous appellerons la **fréquence de traitement** d'une implémentation, c'est-à-dire le nombre moyen d'étapes de propagation effectuées par seconde au cours d'une exécution. Cette donnée nous permettra d'estimer la rapidité d'exécution des différentes implémentations ; elle est complémentaire au nombre total d'étapes de propagation, qui fournit plutôt une indication sur la longueur

TABLEAU 6.1: Comparaison expérimentale de différentes implémentations (*impl*) du CDCL classique et du CDCL sans saut arrière dans le solveur GLUCOSE. Les implémentations du CDCL classique sont l'implémentation originale de GLUCOSE ($CDCL_{LB}$) et une variante sans littéraux bloqués (CDCL). Les implémentations du CDCL sans saut arrière sont CDCL-SSA-Graphe (abrégé par Graphe), CDCL-SSA-Parcours (Parcours), CDCL-SSA_{LB} (SSA_{LB}), CDCL-SSA-PropAltTard (PropAltTard), CDCL-SSA-PropAltAnt (PropAltAnt) et CDCL-SSA-Conservation (Conservation). Chaque implémentation a été exécutée sur les 62 instances énumérées dans le tableau A.2 avec une limite d'une heure. Pour chaque implémentation, nous indiquons le nombre d'instances résolues à l'intérieur de ce délai (*rés*), le temps total d'exécution en secondes (*t*), le nombre total d'étapes de propagation en millions (*ep*), le nombre moyen d'étapes de propagation par secondes (*#ep/sec*), la destructivité moyenne des conflits (*d/c*), la proportion moyenne de conflits triviaux (*triv*) et l'évolution moyenne de la distance entre les conflits par rapport à l'implémentation originale de GLUCOSE (*dist*).

impl	*rés*	*t*	*ep*	*#ep/sec*	*d/c*	*triv*	*dist*
$CDCL_{LB}$	61	36 500	1 143 301	29 240 778	6,69%	0,00%	100,00%
CDCL	57	44 282	838 267	21 203 876	6,46%	0,00%	96,90%
Graphe	49	91 071	832 419	12 105 521	4,47%	32,02%	65,92%
Parcours	51	83 048	1 092 735	16 125 504	4,53%	31,63%	68,22%
SSA_{LB}	53	68 713	1 092 304	20 759 144	4,34%	32,04%	67,44%
PropAltTard	37	115 506	495 315	5 071 897	5,14%	14,81%	75,84%
PropAltAnt	48	91 124	489 014	6 167 998	4,96%	15,05%	65,77%
Conservation	44	100 683	500 089	5 882 930	5,13%	14,88%	68,78%

des recherches effectuées. Pour chaque implémentation, nous indiquons la moyenne de la fréquence de traitement sur l'ensemble des instances testées.

La cinquième colonne quantifie la destructivité des conflits, c'est-à-dire la quantité d'instanciations défaites au cours d'un conflit. Cette quantité est exprimée en pourcentage par rapport au nombre total de variables du problème, afin de donner le même poids à toutes les instances, quelque soit leur nombre de variables. La quantité d'instanciation défaites par conflit sur une instance est la moyenne de cette quantité sur tous les conflits survenus pendant l'exécution sur cette instance (redémarrages non-compris) ; le tableau 6.1 indique pour chaque implémentation une moyenne sur l'ensemble des instances. Cette donnée nous permettra de vérifier à quel point le CDCL sans saut arrière permet de réduire la destructivité moyenne des conflits par rapport au CDCL classique. Notons que le CDCL sans saut arrière produit une recherche différente par rapport

au CDCL classique ; il est donc possible, sur certaines instances, d'avoir une quantité moyenne d'instanciations défaites par conflit dans une implémentation du CDCL sans saut arrière supérieure à cette même quantité dans une implémentation du CDCL classique, bien que, pour tout conflit donné, une résolution du conflit par le CDCL sans saut arrière ne puisse pas défaire plus d'instanciations qu'un saut arrière classique.

La sixième colonne indique la proportion de conflits triviaux par rapport au nombre de conflits total dans une exécution. Là encore, nous indiquons pour chaque implémentation une moyenne de cette proportion sur la résolution de toutes les instances.

Enfin, la septième colonne fournit des informations sur la **distance** entre les conflits successifs, c'est-à-dire le nombre moyen d'étapes de propagation nécessaires pour déclencher un conflit. Comme les conflits (du moins les conflits non-redondants) permettent d'élaguer l'espace de recherche, on peut supposer qu'une distance plus courte entre les conflits aide à élaguer l'espace de recherche plus rapidement et donc à raccourcir la taille globale de la recherche en nombre d'étapes. Dans ce cas également, il est nécessaire de normaliser cette information car, pour une même implémentation, certaines instances peuvent avoir des distances moyennes entre conflits mille fois supérieures à d'autres. Par conséquent, nous indiquerons pour toute exécution d'une implémentation sur une instance le rapport entre la distance moyenne dans cette exécution et la distance moyenne dans l'exécution de l'implémentation originale de GLUCOSE sur la même instance. Ceci quantifie en quelque sorte l'évolution de cette distance moyenne par rapport à GLUCOSE. Cette quantité est donc toujours égale à 100% pour l'implémentation CDCL_{LB}, c'est-à-dire la version originale de GLUCOSE. Le tableau présente pour chaque implémentation une moyenne de l'évolution de la distance des conflits sur la résolution de toutes les instances.

On peut remarquer que certaines des données présentées par le tableau 6.1 sont plus ou moins significatives selon que l'implémentation parvient ou non à résoudre l'instanciation dans la limite de temps donnée, notamment le nombre d'étapes de propagation

TABLEAU 6.2: Comparaison entre différentes implémentations de CDCL et CDCL-SSA sur le nombre d'instances qu'elles résolvent plus rapidement. Pour chaque paire d'implémentations, le tableau indique le nombre total d'instances que toutes deux résolvent dans le temps imparti, puis, parmi celles-ci, le nombre d'instances résolues plus rapidement par l'implémentation en ordonnée et l'implémentation en abscisse respectivement. Lorsque la même implémentation est à la fois en abscisse et en ordonnée, nous indiquons uniquement le nombre d'instances qu'elle résout dans le temps imparti. Les implémentations comparées et les instances considérées sont les mêmes que dans le tableau 6.1.

	CDCL_{LB}	CDCL	Graphe	Parcours	SSA_{LB}	PropAltTard	PropAltAnt	Conservation
CDCL_{LB}	61	57 (36/20)	49 (36/12)	51 (41/9)	52 (42/9)	37 (31/5)	48 (40/7)	44 (35/8)
CDCL	57 (20/36)	57	45 (39/5)	48 (37/10)	49 (34/14)	36 (29/6)	44 (36/7)	42 (34/7)
Graphe	49 (12/36)	45 (5/39)	49	47 (13/33)	46 (11/34)	34 (24/9)	44 (29/14)	42 (26/15)
Parcours	51 (9/41)	48 (10/37)	47 (33/13)	51	50 (21/28)	37 (27/9)	45 (33/11)	43 (32/10)
SSA_{LB}	52 (9/42)	49 (14/34)	46 (34/11)	50 (28/21)	53	37 (31/5)	46 (34/11)	42 (33/8)
PropAltTard	37 (5/31)	36 (6/29)	34 (9/24)	37 (9/27)	37 (5/31)	37	37 (12/24)	35 (13/21)
PropAltAnt	48 (7/40)	44 (7/36)	44 (14/29)	45 (11/33)	46 (11/34)	37 (24/12)	48	41 (24/16)
Conservation	44 (8/35)	42 (7/34)	42 (15/26)	43 (10/32)	42 (8/33)	35 (21/13)	41 (16/24)	44

TABLEAU 6.3: Comparaison de différentes implémentations de CDCL et CDCL-SSA sur le temps total d'exécution pour les instances qu'elles résolvent en commun. Pour chaque paire d'implémentations, le tableau indique le temps total d'exécution nécessité respectivement par l'implémentation en ordonnée et l'implémentation en abscisse pour la résolution de l'ensemble des instances que les deux implémentations sont capables de résoudre dans le temps imparti. Lorsque la même implémentation est à la fois en abscisse et en ordonnée, nous indiquons uniquement le temps total d'exécution sur les instances qu'elle résout dans le temps imparti. Les implémentations comparées et les instances considérées sont les mêmes que dans le tableau 6.1.

	CDCL_{LB}	CDCL	Graphe	Parcours	SSA_{LB}	PropAltTard	PropAltAnt	Conservation
CDCL_{LB}	32 901	23 297/26 283	22 266/44 272	20 636/43 449	19 675/35 974	5 098/25 507	19 709/40 725	16 996/35 883
CDCL	26 283/23 297	26 283	13 142/39 241	15 008/38 970	15 932/33 548	6 458/23 858	14 432/30 540	11 642/32 474
Graphe	44 272/22 266	39 241/13 142	44 272	40 030/34 795	36 902/26 647	16 853/21 903	35 415/37 554	30 861/33 578
Parcours	43 449/20 636	38 970/15 008	34 795/40 030	43 449	40 130/32 591	15 676/25 507	27 133/36 507	24 380/35 579
SSA_{LB}	35 974/19 675	33 548/15 932	26 647/36 902	32 591/40 130	36 314	12 465/25 507	25 421/36 570	20 718/32 568
PropAltTard	25 507/5 098	23 858/6 458	21 903/16 853	25 507/15 676	25 507/12 465	25 507	25 507/17 415	24 611/18 545
PropAltAnt	40 725/19 709	30 540/14 432	37 554/35 415	36 507/27 133	36 570/25 421	17 415/25 507	40 725	27 854/29 379
Conservation	35 883/16 996	32 474/11 642	33 578/30 861	35 579/24 380	32 568/20 718	18 545/24 611	29 379/27 854	35 883

effectuées lors de l'exécution. En effet, si une implémentation a parvient à résoudre une instance i en moins d'étapes de propagation que l'implémentation b, cela signifie que l'implémentation a nécessite une moins grande quantité de recherche pour résoudre i que b. Toutefois, si ni a, ni b ne parviennent à résoudre i dans le temps imparti mais a a effectué dans ce temps moins d'étapes de propagation que b, cela n'indique pas que a peut résoudre i avec une plus petite quantité de recherche que b, ce qui serait un avantage, mais seulement que a a une fréquence de traitement inférieure à celle de b, ce qui est un inconvénient. Par conséquent, la quantité totale des étapes de propagation d'une implémentation donnée sur toutes les instances, résolues ou non, n'apporte pas vraiment d'information sur son efficacité.

Pour avoir des indications plus significatives, nous présentons donc des tableaux supplémentaires qui comparent les différentes implémentations deux par deux uniquement sur les instances qu'elles parviennent toutes deux à résoudre dans le temps imparti, que nous appellerons leurs instances résolues en commun. Par exemple, le tableau 6.4 indique, pour chaque paire d'implémentations distinctes, le nombre d'instances qu'elles résolvent toutes deux dans le temps imparti, et parmi celles-ci le nombre d'instances que chacune des implémentations résout en moins d'étapes de propagation que l'autre. Le tableau 6.5, quant à lui, compare deux implémentations en indiquant pour chacune le nombre total d'étapes de propagation sur l'ensemble des instances que les deux implémentations parviennent à résoudre. Les tableaux 6.2 et 6.3 sont respectivement similaires aux tableaux 6.4 et 6.5, à la différences qu'ils comparent les implémentations selon leurs temps d'exécution plutôt que leurs nombres d'étapes de propagation. Enfin, les tableaux 6.6 et 6.7 comparent les implémentations selon leur proportion moyenne de variables désinstanciées lors des conflits. Notons que le tableau 6.7 donne pour chaque paire d'implémentations la moyenne de cette proportion sur les instances qu'elles parviennent toutes deux à résoudre, tandis que les tableaux 6.3 et 6.5 donnent quant à eux des totaux (respectivement du temps d'exécution et du nombre d'étapes de propagation).

TABLEAU 6.4: Comparaison de différentes implémentations de CDCL et CDCL-SSA sur le nombre d'instances qu'elles résolvent en moins d'étapes de propagation. Pour chaque paire d'implémentations, le tableau indique le nombre total d'instances que toutes deux résolvent dans le temps imparti, puis, parmi celles-ci, le nombre d'instances résolues avec moins d'étapes de propagation par l'implémentation en ordonnée et l'implémentation en abscisse respectivement. Lorsque la même implémentation est à la fois en abscisse et en ordonnée, nous indiquons uniquement le nombre d'instances qu'elle résout dans le temps imparti. Les implémentations comparées et les instances considérées sont les mêmes que dans le tableau 6.1.

	$CDCL_{LB}$	CDCL	Graphe	Parcours	SSA_{LB}	PropAltTard	PropAltAnt	Conservation
$CDCL_{LB}$	61	57 (24/33)	49 (30/19)	51 (34/17)	52 (36/16)	37 (21/16)	48 (24/24)	44 (21/23)
CDCL	57 (33/24)	57	45 (32/13)	48 (37/11)	49 (36/13)	36 (23/13)	44 (27/17)	42 (23/19)
Graphe	49 (19/30)	45 (13/32)	49	47 (25/21)	46 (28/18)	34 (10/24)	44 (11/33)	42 (11/31)
Parcours	51 (17/34)	48 (11/37)	47 (21/25)	51	50 (33/17)	37 (8/29)	45 (7/38)	43 (7/36)
SSA_{LB}	52 (16/36)	49 (13/36)	46 (18/28)	50 (17/33)	53	37 (6/31)	46 (7/39)	42 (8/34)
PropAltTard	37 (16/21)	36 (13/23)	34 (24/10)	37 (29/8)	37 (31/6)	37	37 (15/22)	35 (16/19)
PropAltAnt	48 (24/24)	44 (17/27)	44 (33/11)	45 (38/7)	46 (39/7)	37 (22/15)	48	41 (22/19)
Conservation	44 (23/21)	42 (19/23)	42 (31/11)	43 (36/7)	42 (34/8)	35 (19/16)	41 (19/22)	44

TABLEAU 6.5: Comparaison de différentes implémentations de CDCL et CDCL-SSA sur le nombre total d'étapes de propagation pour les instances qu'elles résolvent en commun. Pour chaque paire d'implémentations, le tableau indique le nombre total d'étapes de propagation nécessitées respectivement par l'implémentation en ordonnée et l'implémentation en abscisse pour la résolution de l'ensemble des instances que les deux implémentations sont capables de résoudre dans le temps imparti. Lorsque la même implémentation est à la fois en abscisse et en ordonnée, nous indiquons uniquement le nombre total d'étapes de propagation sur les instances qu'elle résout dans le temps imparti. Les implémentations comparées et les instances considérées sont les mêmes que dans le tableau 6.1.

	$CDCL_{LB}$	CDCL	Graphe	Parcours	SSA_{LB}	PropAltTard	PropAltAnt	Conservation
$CDCL_{LB}$	985 261	608 438/451 900	750 461/518 483	704 702/642 567	687 425/646 549	161 713/130 802	622 684/253 543	533 876/194 724
CDCL	451 900/608 438	451 900	271 274/427 411	295 414/535 220	310 750/544 575	137 792/119 089	252 956/169 669	226 414/167 131
Graphe	518 483/750 461	427 411/271 274	518 483	479 565/563 990	466 601/556 201	188 036/119 666	386 396/241 393	333 724/186 363
Parcours	642 567/704 702	535 220/295 414	563 990/479 565	642 567	631 468/612 949	211 961/130 802	379 356/227 456	310 332/192 389
SSA_{LB}	646 549/687 425	544 575/310 750	556 201/466 601	612 949/631 468	659 092	211 784/130 802	419 745/227 913	345 300/186 297
PropAltTard	130 802/161 713	119 089/137 792	119 666/188 036	130 802/211 961	130 802/211 784	130 802	130 802/117 272	128 413/113 355
PropAltAnt	253 543/622 684	169 669/252 956	241 393/386 396	227 456/379 356	227 913/419 745	117 272/130 802	253 543	168 324/162 978
Conservation	194 724/533 876	167 131/226 414	186 363/333 724	192 389/310 332	186 297/345 300	113 355/128 413	162 978/168 324	194 724

6.6.2 Interprétation des résultats

En comparant les deux variantes du CDCL classique, on peut aisément remarquer l'intérêt pratique des littéraux bloqués, puisque leur désactivation provoque une baisse de la fréquence moyenne de traitement d'environ 27,5%. Il s'agit sans doute du facteur principal qui permet à CDCL_{LB} de résoudre 4 instances de plus que CDCL, tout en réduisant le temps d'exécution total. Notons toutefois que les littéraux bloqués semblent avoir un effet néfaste sur la taille des résolutions en nombre d'étapes de propagation, puisque parmi les 57 instances résolues à la fois par CDCL_{LB} et CDCL, ce dernier nécessite moins d'étapes de propagation sur 33 d'entre elles, et il réduit le nombre total d'étapes sur la résolution de ces instances d'environ 25% par rapport à CDCL_{LB}. Grâce à la différence de fréquence de traitement, CDCL_{LB} a toutefois un léger avantage sur CDCL en terme de temps d'exécution sur ces 57 instances, auquel il faut rajouter le gain conséquent des 4 instances résolues supplémentaires. Les différences entre les deux implémentations en termes de destructivité des conflits et de distance entre les conflits sont négligeables ; toutefois, la destructivité sensiblement moins élevée dans le cas du CDCL sans littéraux bloqués pourrait expliquer en partie la baisse du nombre d'étapes de propagation par rapport à CDCL_{LB}.

Si l'on compare les résultats expérimentaux de nos deux implémentations de CDCL sans saut arrière, on peut remarquer, même si l'on se restreint aux instances qu'elles résolvent toutes deux, qu'elles provoquent des exécutions significativement différentes sur une même instance. Cela peut sembler surprenant, puisque leur seule différence est la façon dont les instanciations à défaire lors des résolutions de conflits sont retrouvées ; toutes deux défont cependant exactement le même ensemble d'instanciations si elles se trouvent dans la même situation. Elles devraient donc a priori explorer l'espace de recherche de façon exactement identique.

En fait, cette différence d'exécution s'explique par l'implémentation de GLUCOSE, plus précisément par sa gestion de l'heuristique de décision. Celle-ci utilise une file de priorité, ordonnée par ordre décroissant d'activité, dont les variables sont retirées

TABLEAU 6.6: Comparaison de différentes implémentations de CDCL et CDCL-SSA sur le nombre d'instances qu'elles résolvent avec une plus petite destructivité des conflits. Pour chaque paire d'implémentations, le tableau indique le nombre total d'instances que toutes deux résolvent dans le temps imparti, puis, parmi celles-ci, le nombre d'instances résolues avec une plus petite destructivité des conflits par l'implémentation en ordonnée et l'implémentation en abscisse respectivement. Lorsque la même implémentation est à la fois en abscisse et en ordonnée, nous indiquons uniquement le nombre d'instances qu'elle résout dans le temps imparti. Les implémentations comparées et les instances considérées sont les mêmes que dans le tableau 6.1.

	$CDCL_{LB}$	CDCL	Graphe	Parcours	SSA_{LB}	PropAltTard	PropAltAnt	Conservation
$CDCL_{LB}$	61 (0/0)	57 (29/27)	49 (4/45)	51 (6/45)	52 (3/49)	37 (9/28)	48 (12/36)	44 (9/35)
CDCL	57 (27/29)	57 (0/0)	45 (3/42)	48 (4/44)	49 (3/46)	36 (6/30)	44 (9/35)	42 (7/35)
Graphe	49 (45/4)	45 (42/3)	49 (0/0)	47 (24/22)	46 (18/28)	34 (29/5)	44 (41/3)	42 (37/5)
Parcours	51 (45/6)	48 (44/4)	47 (22/24)	51 (0/0)	50 (23/27)	37 (32/5)	45 (38/7)	43 (38/5)
SSA_{LB}	52 (49/3)	49 (46/3)	46 (28/18)	50 (27/23)	53 (0/0)	37 (34/3)	46 (42/4)	42 (37/5)
PropAltTard	37 (28/9)	36 (30/6)	34 (5/29)	37 (5/32)	37 (3/34)	37 (0/0)	37 (17/20)	35 (12/23)
PropAltAnt	48 (36/12)	44 (35/9)	44 (3/41)	45 (7/38)	46 (4/42)	37 (20/17)	48 (0/0)	41 (18/23)
Conservation	44 (35/9)	42 (35/7)	42 (5/37)	43 (5/38)	42 (5/37)	35 (23/12)	41 (23/18)	44 (0/0)

TABLEAU 6.7: Comparaison de différentes implémentations de CDCL et CDCL-SSA sur la destructivité moyenne des conflits pour les instances qu'elles résolvent en commun. Pour chaque paire d'implémentations, le tableau indique la moyenne des destructivité des conflits respectivement dans l'implémentation en ordonnée et l'implémentation en abscisse sur l'ensemble des instances que les deux implémentations sont capables de résoudre dans le temps imparti. Lorsque la même implémentation est à la fois en abscisse et en ordonnée, nous indiquons uniquement la destructivité moyenne des conflits sur les instances qu'elle résout dans le temps imparti. Les implémentations comparées et les instances considérées sont les mêmes que dans le tableau 6.1.

	$CDCL_{LB}$	CDCL	Graphe	Parcours	SSA_{LB}	PropAltTard	PropAltAnt	Conservation
$CDCL_{LB}$	6,74%	6,52%/6,31%	6,41%/4,68%	6,48%/4,61%	7,38%/4,68%	6,09%/4,64%	6,53%/4,84%	5,87%/4,45%
CDCL	6,31%/6,52%	6,31%	5,93%/4,34%	5,95%/4,17%	6,78%/4,24%	5,75%/4,45%	5,89%/4,43%	5,54%/4,24%
Graphe	4,68%/6,41%	4,34%/5,93%	4,68%	4,84%/4,87%	4,94%/4,83%	4,59%/4,92%	4,32%/4,84%	4,25%/4,57%
Parcours	4,61%/6,48%	4,17%/5,95%	4,87%/4,84%	4,61%	4,69%/4,56%	4,34%/4,64%	4,31%/4,78%	4,21%/4,51%
SSA_{LB}	4,68%/7,38%	4,24%/6,78%	4,83%/4,94%	4,56%/4,69%	4,65%	4,13%/4,64%	4,34%/5%	4,1%/4,61%
PropAltTard	4,64%/6,09%	4,45%/5,75%	4,92%/4,59%	4,64%/4,34%	4,64%/4,13%	4,64%	4,64%/4,81%	4,83%/4,86%
PropAltAnt	4,84%/6,53%	4,43%/5,89%	4,84%/4,32%	4,78%/4,31%	5%/4,34%	4,81%/4,64%	4,84%	4,69%/4,5%
Conservation	4,45%/5,87%	4,24%/5,54%	4,57%/4,25%	4,51%/4,21%	4,61%/4,1%	4,86%/4,83%	4,5%/4,69%	4,45%

lorsqu'elles sont décidées ou détectées déjà instanciées par propagation, puis réinsérées lorsque leur instanciation est défaite. Si plusieurs variables dans la file ont une activité identique, leur ordre dans la file dépend de l'ordre dans lequel elles y ont été insérées. Or, CDCL-SSA-Graphe et CDCL-SSA-Parcours retrouvent les instanciations à défaire dans un ordre potentiellement différent, puisque CDCL-SSA-Graphe parcourt le graphe des dépendances en largeur à partir de la décision du niveau de conflit, tandis que CDCL-SSA-Parcours détecte les instanciations à défaire par ordre croissant de niveau de décision, puis, au sein d'un niveau de décision, par ordre chronologique d'instanciation. Puisque l'ordre de désinstanciation des variables est différent, l'ordre relatif des variables de même activité est également différent, et donc les deux implémentations peuvent prendre à un moment donné une décision différente. Les différences de traces d'exécutions entre les deux implémentations sont uniquement dues à ce type de divergence.

Les résultats expérimentaux indiquent que le but principal du CDCL sans saut arrière, qui est la réduction de la destructivité des résolutions de conflit, est bien atteint. En effet, CDCL-SSA-Graphe et CDCL-SSA-Parcours réduisent tous deux significativement la proportion moyenne de variables désinstanciées lors des conflits par rapport aux deux variantes du CDCL classique, que l'on considère toutes les instances ou seulement celles résolues par les deux implémentations comparées. Comme nous l'avons expliqué précédemment, il arrive parfois qu'une implémentation de CDCL classique résolve une instance avec une proportion moindre qu'une implémentation de CDCL sans saut arrière, mais le phénomène inverse est bien plus fréquent, comme le montre le tableau 6.6.

Toutefois, cette réduction de la destructivité des sauts arrière ne permet généralement pas de réduire la longueur des résolutions ; dans la plupart des cas, elles sont même rallongées (tableau 6.4), ce qui a pour conséquence pour les deux implémentations une légère baisse du nombre total d'étapes de propagation par rapport à CDCL_{LB} sur leurs instances résolues en commun, mais une augmentation importante par rapport à CDCL (tableau 6.5). La réduction de la destructivité des résolutions de conflits ne suffit donc souvent pas à réduire la longueur de la résolution.

Nous pouvons avancer différentes explications pour ce comportement à l'opposé de notre intuition. La première est que le CDCL sans saut arrière provoque certainement des interférences avec divers éléments de l'algorithme CDCL, notamment les heuristiques de décision. En effet, si, contrairement aux décompositions implicites par exemple, le CDCL sans saut arrière ne pose aucune contrainte sur ces heuristiques, la possibilité de défaire des décisions dans un ordre quelconque peut en quelque sorte les modifier a posteriori. Lorsqu'une décision l est défaite alors que plusieurs décisions ultérieures $l_1, \ldots, l_k$ sont conservées, la recherche se retrouve dans un état qui revient à avoir pris les décisions $l_1, \ldots, l_k$ avant la décision l, ce qui est donc un ordre différent de celui fourni par l'heuristique de décision ordinaire. Comme l'efficacité des heuristiques de décision basées sur l'activité des variables a été constatée empiriquement, il est possible qu'un tel impact puisse avoir un effet négatif sur la recherche dans son ensemble.

Une autre explication probable est la conséquence de la désactivation de la sauvegarde de phase. En effet, comme nous l'avons souligné dans la sous-section 2.5.6, celle-ci a un impact important sur les redémarrages puisqu'elle permet de retourner après un redémarrage dans un sous-espace de recherche très proche du sous-espace précédent. Sans sauvegarde de phase, la recherche après un redémarrage peut emprunter un chemin très différent et explorer un sous-espace de recherche très éloigné du précédent, ce qui peut par exemple nuire à l'efficacité des clauses récemment apprises, et expliquer en partie l'augmentation de la longueur des recherches.

Notons que près d'un tiers des conflits dans CDCL-SSA-Graphe et CDCL-SSA-Parcours est trivial, et donc redondant, selon la contraposée de la proposition 5.4. Or, chaque conflit redondant défait une partie de l'instanciation courante sans apporter aucune nouvelle connaissance. On peut donc considérer que les instanciations défaites ont été décidées ou propagées pour rien ; une partie des étapes de propagation antérieures n'a donc eu aucune utilité sur l'élagage de l'espace de recherche. Il est donc évident que l'occurrence de conflits redondants, qui ne peuvent survenir dans un CDCL classique, est un facteur additionnel dans l'augmentation du nombre d'étapes de propagation.

Enfin, on peut remarquer que pour les deux implémentations de CDCL sans saut arrière, la distance entre les conflits diminue en moyenne d'environ un tiers par rapport à CDCL_{LB}. Il est possible que cette diminution soit un effet positif de la réduction de la destructivité des conflits, qui permettrait au CDCL sans saut arrière d'accéder plus rapidement au conflit suivant car un plus grand nombre de variables est resté instancié. Cependant, cette diminution de la distance entre conflits pourrait également être due à la présence additionnelle de conflits redondants. Quelle que soit sa cause, cette diminution de la distance est environ équivalente à la proportion de conflits redondants ; par conséquent, la distance moyenne entre conflits utiles reste comparable à la distance entre conflits dans le CDCL_{LB}.

En plus de ce bilan mitigé sur le plan de la longueur de la recherche, CDCL-SSA-Graphe et CDCL-SSA-Parcours sont surtout handicapés par une baisse importante de la fréquence de traitement. En effet, la fréquence de CDCL-SSA-Parcours est à peine plus de la moitié de la fréquence de CDCL_{LB}, et est encore plus basse pour CDCL-SSA-Graphe. Si ce ralentissement de l'exécution s'explique en partie par l'absence de littéraux bloqués, elle est également due à d'autres facteurs supplémentaires, puisque la fréquence de traitement de CDCL (sans littéraux bloqués) est supérieure de plus de 30% à celle de CDCL-SSA-Parcours et d'environ 75% à celle de CDCL-SSA-Graphe. Le reste de cette baisse de fréquence peut logiquement être attribué aux manipulations supplémentaires nécessitées par le CDCL sans saut arrière : tout d'abord, pour chaque littéral propagé, il est nécessaire de calculer son niveau de décision à partir des niveaux de décision de ses antécédents. CDCL-SSA-Graphe nécessite en plus de manipuler le graphe de dépendances à chaque propagation et à chaque conflit. CDCL-SSA-Parcours, lui, requiert de parcourir un sous-ensemble des variables instanciées lors de chaque résolution de conflit. La différence importante de fréquence de traitement indique clairement que le surcoût causé par CDCL-SSA-Graphe est bien plus important que pour CDCL-SSA-Parcours, ce qui s'explique aisément par la manipulation du graphe à chaque nouvelle propagation unitaire, ce qui est très coûteux étant donnée la fréquence des propagations. Si CDCL-SSA-Parcours vérifie potentiellement plus de variables lors de la résolution des

conflits, le fait d'éviter ce traitement supplémentaire lors des propagations est donc clairement bénéfique à son efficacité.

Au final, comme les deux implémentations du CDCL sans saut arrière ne parviennent pas à réduire significativement la longueur de résolution tout en dégradant de façon importante la fréquence de traitement, le bilan en terme de temps de résolution est négatif par rapport aux deux implémentations de CDCL classique : sur les 62 instances testées, CDCL-SSA-Graphe et CDCL-SSA-Parcours parviennent à résoudre respectivement 12 et 10 instances de moins que l'implémentation originale de GLUCOSE dans le temps imparti, et le temps de résolution total est plus que doublé. Là encore, l'absence de littéraux bloqués ne suffit pas à expliquer cette baisse de performances, puisqu'elles parviennent également à résoudre respectivement 8 et 6 instances de moins que l'implémentation du CDCL classique sans littéraux bloqués, et le temps total de résolution reste significativement grand, même en ne considérant que les instanciations résolues à l'intérieur du temps imparti.

En résumé, CDCL-SSA-Graphe et CDCL-SSA-Parcours parviennent bien à réduire la destructivité moyenne des résolutions de conflits, mais elles baissent fortement la fréquence de traitement des étapes de propagation sans parvenir à réduire significativement la taille des résolutions (voire en l'augmentant par rapport au CDCL sans littéraux bloqués) ; par conséquent, elles parviennent à résoudre moins d'instances dans un temps donné et augmentent fortement le temps total de résolution. Toutefois, nos résultats démontrent aussi que CDCL-SSA-Parcours est significativement plus performante que CDCL-SSA-Graphe, puisque la fréquence des vérifications de clauses y est en moyenne supérieure d'environ un tiers, ce qui par conséquent augmente sensiblement le nombre d'instances résolues à l'intérieur du temps limite et réduit le temps total d'exécution. Nous utiliserons donc la résolution de conflit de CDCL-SSA-Parcours dans les autres variantes de CDCL sans saut arrière introduites dans la suite de ce chapitre, mais les stratégies décrites sont également implémentables avec la résolution de conflit de CDCL-SSA-Graphe.

6.7 CDCL sans saut arrière avec littéraux bloqués

Les deux implémentations de CDCL sans saut arrière testées dans la sous-section 6.6 n'utilisent pas le procédé de littéraux bloqués. Or, les résultats expérimentaux présentés démontrent l'importance des littéraux bloqués dans l'efficacité du CDCL classique. Cette section étudie donc l'intégration des littéraux bloqués dans l'algorithme de CDCL sans saut arrière. La sous-section 6.7.1 démontre que l'introduction des littéraux bloqués empêche la correction de l'algorithme et décrit une manière de rétablir la correction. La sous-section 6.7.2 présente quant à elle une évaluation expérimentale de cette variante de l'algorithme.

6.7.1 Propriétés et description

Les preuves de complétude et de terminaison du CDCL classique restent valables dans le CDCL sans saut arrière à littéraux bloqués, que nous noterons CDCL-SSA$_{\text{LB}}$. Toutefois, la correction n'est plus assurée car les littéraux bloqués brisent la surveillance partielle dans le CDCL sans saut arrière.

Proposition 6.5. *CDCL-SSA$_{LB}$ n'est pas à surveillance partielle, ni à conflits exhaustifs.*

Démonstration. Prouvons la proposition par contre-exemple. Considérons une formule propositionnelle à 5 variables a, b, c, d et e et à 5 clauses $c_1 = a \vee \neg b \vee \neg c$, $c_2 = \neg a \vee \neg d \vee \neg e$, $c_3 = \neg a \vee \neg d \vee e$, $c_4 = \neg a \vee d \vee \neg e$ et $c_5 = \neg a \vee d \vee e$. Supposons qu'initialement, dans chaque clause, les deux derniers littéraux de chaque clause sont surveillés et le premier littéral de la clause est le littéral bloqué pour les deux littéraux surveillés. Considérons l'exécution suivante :

- Le littéral a est instancié comme décision de niveau 1. Le littéral $\neg a$ n'est surveillé dans aucune clause, donc la propagation de a n'entraîne aucune vérification de clause.
- Le littéral b est instancié comme décision de niveau 2. Le littéral bloqué de c_1

pour $\neg b$, a, est vrai, donc la clause n'est pas vérifiée.

- Le littéral c est instancié comme décision de niveau 3. Le littéral bloqué de c_1 pour $\neg c$, a, est vrai, donc la clause n'est pas vérifiée. Elle est donc toujours surveillée par b et c, qui sont maintenant tous deux faux et propagés.
- Le littéral d est instancié comme décision de niveau 4. Le littéral opposé $\neg d$ est surveillé dans les clauses c_2 et c_3. Supposons que c_2 est vérifiée la première. Le littéral bloqué de c_2 pour $\neg d$, $\neg a$, est faux, et la clause est unitaire, donc $\neg e$ est instancié comme propagation au niveau 4 et devient le littéral bloqué de c_2 pour $\neg d$.
- La clause c_3 est maintenant vérifiée par propagation de d. Le littéral bloqué de c_3 pour $\neg d$, $\neg a$, est faux, comme tous les autres littéraux de la clause. La clause de conflit est $c_6 = \neg a \vee \neg d$, de niveau de conflit 4 et niveau d'assertion 1.
- Les instanciations d et $\neg e$ sont défaites et $\neg d$ est instancié comme propagation de c_6 au niveau 1. Le littéral opposé d est surveillé dans les clauses c_4 et c_5 ; supposons que c_4 est vérifiée la première. Le littéral bloqué de c_4 pour d, $\neg a$, est faux, et la clause est unitaire. Le littéral $\neg e$ est donc propagé au niveau 1.
- La clause c_5 est maintenant vérifiée par propagation de $\neg d$. Le littéral bloqué de c_5 pour d, $\neg a$, est faux, comme tous les autres littéraux de la clause. La clause de conflit est $c_7 = \neg a$, de niveau de conflit 1 et niveau d'assertion 0.
- La résolution du conflit défait les instanciations a et $\neg d$, puis instancie $\neg a$ comme propagation au niveau 0. Le littéral a n'est surveillé dans aucune clause, donc la propagation de $\neg a$ n'entraîne aucune vérification de clause. La clause c_1 est désormais surveillée par deux littéraux faux et déjà propagés, dont les littéraux bloqués associés sont faux ; elle n'est donc pas partiellement surveillée. De plus, cette clause est désormais fausse et n'a pas été détectée par la procédure de propagation unitaire, ce qui démontre que CDCL-SSA$_{\mathrm{LB}}$ n'est pas à conflits exhaustifs.

□

Le mécanisme de littéral bloqué n'est pas adapté à un algorithme comme le CDCL sans saut arrière qui permet de désinstancier les variables dans un ordre arbitraire. En effet, dans un CDCL classique, pour une clause c dont un littéral faux w_1 n'est pas remplacé en raison d'un littéral bloqué vrai $\beta(c, w_1)$, on a $\lambda(\beta(c, w_1)) \leq \lambda(w_1)$. Par conséquent, $\beta(c, w_1)$ restera instancié tant que w_1 le reste également. Cette relation entre les niveaux de décision de $\beta(c, w_1)$ et w_1 n'est pas forcément vraie dans le CDCL sans saut arrière ; de plus, l'algorithme ne désinstancie pas les variables dans l'ordre inverse de leur instanciation. $\beta(c, w_1)$ peut donc être désinstanciée avant w_1 ; si w_2 est également fausse et a de la même façon été couverte temporairement par un littéral bloqué vrai, possiblement le même, les deux littéraux surveillés de c peuvent être tous les deux faux et déjà propagés. La clause c n'est alors plus du tout surveillée tant que w_1 et w_2 ne sont pas défaits ; en particulier, si la clause devient entièrement fausse, cela ne sera pas détecté par la procédure de propagation unitaire.

L'exemple utilisé pour prouver la proposition 6.5 montre clairement que sans dispositif supplémentaire, cette non-exhaustivité des conflits rend l'algorithme incorrect : à la fin de cette preuve, on peut instancier les variables d et e par décision à n'importe quelle polarité et ainsi obtenir une instanciation complète qui falsifie une clause sans que cela soit détecté. Pour rétablir la correction, il est alors nécessaire au minimum de vérifier l'ensemble des clauses lorsque l'algorithme retourne une instanciation complète. Cette stratégie est illustrée par l'algorithme 6.8, similaire à la description globale du CDCL sans saut arrière de base (algorithme 6.1), à l'exception de l'utilisation de la procédure VérifierClauses, décrite par l'algorithme 6.9, lorsqu'une instanciation complète est atteinte. Cette procédure vérifie que toutes les clauses sont satisfaites par l'instanciation complète courante ; si ce n'est pas le cas, une des clauses fausses est renvoyée et déclenche un conflit classique, ce qui provoque une reprise de la recherche. Notons qu'il suffit de vérifier les clauses originales, puisque toutes les clauses apprises en sont des conséquences. CDCL-SSA$_{\text{LB}}$ utilise comme implémentation de Propager l'algorithme 2.16, qui décrit l'utilisation des littéraux bloqués dans un CDCL classique. RésoudreConflit est implémenté par la stratégie de parcours des niveaux de décision supérieurs, décrite par

Algorithme 6.8 CDCL-SSA$_{LB}$

```
1: λc ← 0 /*on commence au pseudo-niveau de décision 0*/
2: σ ← ∅ /*on commence avec l'affectation vide*/
3: c ← NUL /*on commence sans conflit*/
4: boucle
5:     si c = NUL alors
6:         /*c ≠ NUL si VérifierClauses a détecté un conflit*/
7:         /*à l'itération précédente*/
8:         c ← Propager() /*propager les clauses unitaires*/
9:     si c ≠ NUL alors /*un conflit a été découvert*/
10:        λγ ← λ(c) /*calcul du niveau de conflit*/
11:        si λγ = 0 alors /*conflit au niveau de décision 0*/
12:            retourner faux /*F est insatisfaisable*/
13:        sinon
14:            γ ← Analyser(c) /*déduire la clause de conflit γ*/
15:            a ← l ∈ γ | λ(a) = λγ
16:            /*a est unique ; c'est la future assertion*/
17:            RésoudreConflit(λγ)
18:            /*défaire λγ et les propagations qui en dépendent*/
19:            C ← C ∪ {γ} /*γ est apprise*/
20:            PropagerAssertion(a, γ)
21:            c ← NUL /*conflit résolu*/
22:    sinon /*aucun conflit pendant les propagations*/
23:        si D(σ) = V alors /*toutes les variables sont instanciées*/
24:            c ← VérifierClauses()
25:            /*vérification supplémentaire de toutes les clauses*/
26:            si c = NUL alors /*aucune clause insatisfaite*/
27:                retourner σ /*σ est un modèle de F*/
28:        sinon
29:            λc ← NouveauNiveau()
30:            Λ ← Λ ∪ {λc}
31:            choisir v ∈ V\D(σ) /*v est une variable non-instanciée*/
32:            choisir l ∈ {v, ¬v}
33:            Instancier(l, NUL)
34:            /*l est une décision, il n'a pas d'antécédent*/
```

Algorithme 6.9 VérifierClauses() *[CDCL-SSA$_{LB}$]*

```
1: pour c ∈ C faire
2:     si ∀l ∈ c, σ(l) = faux alors
3:         retourner c /*c est insatisfaite*/
4: retourner NUL /*toutes les clauses sont satisfaites*/
```

l'algorithme 6.6.

6.7.2 Évaluation expérimentale

Les résultats expérimentaux obtenus à l'aide de notre implémentation de CDCL-SSA_{LB} sont présentés dans les tableaux 6.1 à 6.7. Les résultats en termes de nombre d'instances résolues et de temps total d'exécution restent inférieurs à ceux des deux variantes du CDCL classique : CDCL-SSA_{LB} résout 8 instances de moins que $CDCL_{LB}$ et 4 de moins que CDCL, et son temps total d'exécution représente plus du double de celui de $CDCL_{LB}$ et presque le double de celui de CDCL. On note toutefois une sensible amélioration par rapport à l'implémentation CDCL-SSA-Parcours : CDCL-SSA_{LB} parvient en effet à résoudre 2 instances de plus et réduit le temps total d'exécution d'environ 17%, et de près de 19% si l'on ne considère que les instanciations résolues par les deux implémentations. CDCL-SSA_{LB} parvient en effet à résoudre plus rapidement 28 des 50 instances résolues à la fois par CDCL-SSA_{LB} et par CDCL-SSA-Parcours.

Cette amélioration du temps d'exécution par rapport à CDCL-SSA-Parcours est due essentiellement à une meilleure efficacité de traitement, puisque la fréquence des étapes de propagation est environ plus rapide de 30% et atteint presque celle du CDCL sans littéraux bloqués. Cette fréquence de traitement de CDCL-SSA_{LB} reste donc inférieure d'environ un tiers à la fréquence de $CDCL_{LB}$, ce qui peut s'expliquer de la même façon que la différence de fréquence entre CDCL-SSA-Parcours et le CDCL sans littéraux bloqués : par les surcoûts dus au calcul du niveau de propagation pour chaque clause unitaire détectée et au parcours de toutes les variables de niveau supérieur au niveau de conflit lors d'une résolution de conflit.

On peut remarquer que, comme dans le cas du CDCL classique, l'introduction des littéraux bloqués produit en général une augmentation de la longueur de résolution des instances : sur les 50 instances résolues par CDCL-SSA-Parcours et CDCL-SSA_{LB}, ce dernier produit une résolution plus longue dans 33 des cas. Cela ne provoque toutefois qu'une augmentation minime du nombre total d'étapes de propagation par rap-

port à CDCL-SSA-Parcours. Par conséquent, CDCL-SSA$_{LB}$ nécessite légèrement moins d'étapes de propagation que CDCL$_{LB}$ sur leurs instances résolues en commun, mais significativement plus que CDCL. On peut apporter à cette incapacité de réduire davantage la longueur des résolutions les mêmes explications que pour le CDCL sans saut arrière sans littéraux bloqués, notamment les interférences de la résolution de conflit sans saut arrière avec d'autres aspects de l'algorithme comme les heuristiques de décision, l'effet de la désactivation de la sauvegarde de phase sur les redémarrages, et le surcoût en étapes de propagation dû aux conflits triviaux et redondants.

La proportion de conflits triviaux est d'ailleurs quasiment la même que pour les deux implémentations de CDCL-SSA sans littéraux bloqués. Cela signifie donc que la perte de la surveillance partielle provoquée par les littéraux bloqués n'induit pas une augmentation significative du nombre de conflits triviaux. La destructivité des conflits et l'évolution de la distance entre conflits par rapport à CDCL$_{LB}$ restent elles aussi quasiment identiques.

Enfin, le nombre de vérifications finales nécessaires (c'est-à-dire d'appels à VérifierClauses) est très variable selon les instances ; 45 instances ne nécessitent aucune ou une seule vérification, 5 en nécessitent de 2 à 10, 7 de 11 à 99, 3 de 100 à 999, et une seule en nécessite plus de 1 000. Curieusement, le nombre de vérifications finales ne semble pas réellement corrélé avec l'efficacité de la recherche en nombre total de vérifications de clauses comparativement à l'algorithme CDCL classique : certaines des instances qui nécessitent beaucoup de vérifications finales avec CDCL-SSA$_{LB}$ sont toutefois résolues en moins de vérifications de clauses que par le CDCL ordinaire. Pourtant, pour chaque vérification finale sauf possiblement la dernière, un conflit a été découvert très tardivement, c'est-à-dire après la construction complète d'une instanciation. On aurait donc pu s'attendre à ce que de telles découvertes retardées de conflits dégradent fortement l'élagage de l'espace de recherche et induisent une augmentation significative du nombre total de vérifications de clauses nécessaire à la résolution du problème, surtout si la découverte tardive de conflits est fréquente. Cela ne semble donc pas réellement être le cas. Il est possible que, même dans les cas les plus extrêmes, le nombre de vérifications

finales reste trop peu élevé par rapport au nombre total de conflits pour avoir un impact notable sur les performances. Les conflits triviaux, en comparaison, sont par exemple bien plus fréquents.

En résumé, si l'introduction des littéraux bloqués dégrade les propriétés théoriques du CDCL sans saut arrière, leur efficacité pratique permet d'améliorer les performances de l'algorithme en augmentant sa rapidité de traitement, sans atteindre les performances du CDCL classique.

6.8 Propagations alternatives

Comme nous l'avons vu dans la section 6.4, la non-exhaustivité des propagations dans le CDCL sans saut arrière est due à la présence possible, après une résolution de conflit, de clauses unitaires susceptibles de propager à nouveau certaines instanciations défaites, mais qui ne sont pas détectables par la procédure de propagation unitaire. Nous appellerons de telles clauses des **propagations alternatives** pour un littéral donné. En fait, pour rétablir l'exhaustivité des propagations dans le CDCL sans saut arrière, il suffit de détecter toutes les propagations alternatives des instanciations défaites lors d'une résolution de conflit et de s'en servir pour repropager ces instanciations. Cette section étudie donc des variantes du CDCL sans saut arrière qui assurent l'exhaustivité des propagations à l'aide de la détection des propagations alternatives. Nous avons vu dans la sous-section 6.7.1 que le mécanisme de littéraux bloqués empêche la surveillance partielle dans le CDCL sans saut arrière ; a fortiori, la surveillance entière est également impossible dans un CDCL sans saut arrière qui utilise les littéraux bloqués. Les variantes du CDCL sans saut arrière à gestion des propagations alternatives doivent donc désactiver ce mécanisme, malgré son efficacité expérimentale constatée dans la sous-section 6.7.2.

Les sous-sections 6.8.1 et 6.8.2 présentent deux stratégies différentes de détection des propagations alternatives, respectivement les détections tardive et anticipée. La sous-section 6.8.3 montre l'impact des propagations alternatives sur les propriétés du CDCL sans saut arrière ; elle prouve notamment que si les propagations sont rendues ex-

haustives, cela ne suffit pas à éliminer totalement les conflits triviaux et redondants. La sous-section 6.8.4 présente une comparaison expérimentale de ces deux implémentations avec les variantes de CDCL et de CDCL sans saut arrière étudiées précédemment.

6.8.1 Détection tardive des propagations alternatives

La détection tardive des propagations alternatives, comme son nom l'indique, propose de repousser le plus possible la gestion des propagations alternatives, c'est-à-dire jusqu'à la désinstanciation des littéraux par une résolution de conflit. Lorsqu'un littéral l est défait, toute clause qui contient l peut potentiellement être une propagation alternative pour l. Cependant, nous pouvons nous restreindre à la recherche des propagations alternatives qui ne peuvent pas être détectées par la procédure de propagation unitaire usuelle. Or, toute propagation alternative pour l où l n'est pas surveillée sera détectée par la propagation unitaire. En effet, une telle clause c est unitaire et l est son seul littéral indéfini. Si l n'est pas surveillé dans c, les deux littéraux surveillés sont donc faux. Puisque le CDCL sans saut arrière est à surveillance partielle, au moins un de ces littéraux n'a pas encore été propagé. Lorsqu'il sera propagé, c sera détectée en tant que clause unitaire et propagera l. Par conséquent, la détection tardive des propagations alternatives consiste, pour tout littéral l désinstancié lors d'une résolution de conflit, à vérifier s'il existe une clause unitaire parmi les clauses où l est surveillé, et à la propager si elle existe.

Le pseudo-code du CDCL à saut arrière et détection tardive des propagations alternatives, nommé CDCL-SSA-PropAltTard, est décrit par l'algorithme 6.10. La procédure PROPAGER est implémentée de la même façon que dans le CDCL classique (algorithme 2.15). L'étape de résolution du conflit (algorithme 6.11) est légèrement modifiée afin de retourner la liste Θ des instanciations défaites. Cette liste est utilisée par la procédure REPROPAGER (ligne 21 de l'algorithme 6.10), qui est décrite en détail par l'algorithme 6.12 et consiste donc à parcourir les clauses où les instanciations de Θ sont surveillées à la recherche d'éventuelles propagations alternatives pour ces instanciations.

Algorithme 6.10 CDCL-SSA-PropAltTard

```
1: Λ ← 0
2: λ_c ← 0  /*on commence au pseudo-niveau de décision 0*/
3: σ ← ∅  /*on commence avec l'affectation vide*/
4: boucle
5:     c ← PROPAGER() /*propager les clauses unitaires*/
6:     si c ≠ NUL alors /*un conflit a été découvert*/
7:         λ_γ ← λ(c)  /*calcul du niveau de conflit*/
8:         si λ_γ = 0 alors /*conflit au niveau de décision 0*/
9:             retourner faux  /*F est insatisfaisable*/
10:        sinon
11:            répéter
12:                γ ← ANALYSER(c)  /*déduire la clause de conflit γ*/
13:                a ← l ∈ γ | λ(a) = λ_γ
14:                /*a est unique ; c'est la future assertion*/
15:                c ← γ  /*γ sera responsable du prochain conflit*/
16:                /*si ¬a n'est pas désinstancié*/
17:                Θ ← RÉSOUDRECONFLIT(λ_γ)
18:                /*défaire λ_γ et les propagations qui en dépendent*/
19:                /*Θ contient la liste des littéraux défaits*/
20:                C ← C ∪ {γ}  /*γ est apprise*/
21:                REPROPAGER(Θ)
22:                /*essayer de repropager les littéraux défaits*/
23:            jusqu'à σ(a) = indéfini
24:            PROPAGERASSERTION(a, γ)
25:    sinon /*aucun conflit pendant les propagations*/
26:        si D(σ) = V alors /*toutes les variables sont instanciées*/
27:            retourner σ  /*σ est un modèle de F*/
28:        sinon
29:            λ_c ← NOUVEAUNIVEAU()
30:            Λ ← Λ ∪ {λ_c}
31:            choisir v ∈ V\D(σ)  /*v est une variable non-instanciée*/
32:            choisir l ∈ {v, ¬v}
33:            INSTANCIER(l, NUL)
34:            /*l est une décision, il n'a pas d'antécédent*/
```

Algorithme 6.11 RÉSOUDRECONFLIT(λ_γ) *[CDCL-SSA-PropAltTard]*

```
1: Θ ← ∅  /*variables défaites pendant le traitement*/
2: pour  λ_γ ≤ i ≤ λ_c  faire
3:     /*on parcourt tous les niveaux à partir du niveau de conflit*/
4:     pour  l ∈ σ | λ(l) = i  faire
5:         /*parcours des littéraux du niveau dans l'ordre d'instanciation*/
6:         si  i = λ_γ  alors
7:             /*défaire toutes les instanciations du niveau de conflit*/
8:             DÉFAIRE(l)
9:             Θ ← Θ ∪ {l}
10:        sinon si  ∃l' ∈ α(l) | σ(l') = indéfini  alors
11:            /*défaire l si un de ses antécédents a été défait*/
12:            DÉFAIRE(l)
13:            Θ ← Θ ∪ {l}
14: Λ ← Λ \ {λ_γ}  /*λ_γ est le seul niveau entièrement défait*/
15: retourner Θ
```

Notons que lorsqu'un littéral défait l n'est pas repropagé, cette désinstanciation compromet la surveillance entière des clauses où l est surveillé et le second littéral propagé w est faux et déjà propagé : en effet, la clause est alors surveillée par un littéral faux déjà propagé et un littéral indéfini, ce qui constitue une surveillance partielle, mais pas entière. Pour résoudre ce problème, lors de la recherche d'une propagation alternative pour l, l'algorithme effectue le remplacement d'un éventuel littéral surveillé faux par un autre littéral vrai ou indéfini dans toute clause parcourue (lignes 9 à 11 de l'algorithme 6.12). Ce remplacement est toujours possible, sinon cela signifie que la clause est unitaire et repropage l, ce qui résout le problème.

Les propagations alternatives peuvent en outre avoir pour conséquence d'empêcher la résolution du conflit. En effet, il est possible que l'algorithme détecte une propagation alternative pour le littéral opposé à l'assertion, c'est-à-dire à la conséquence qui doit être propagée par la clause de conflit. Dans ce cas, la clause de conflit reste fausse malgré la résolution du conflit et ne peut donc être propagée. Il est alors nécessaire de procéder immédiatement à une nouvelle résolution de conflit, où la clause de conflit nouvellement apprise tient le rôle de clause fausse déclencheuse de conflit. Ce cas est couvert dans l'algorithme 6.10 par la boucle **répéter** (lignes 11 à 23) qui exécute à nouveau les étapes

Algorithme 6.12 REPROPAGER(Θ) *[CDCL-SSA-PropAltTard]*

Requis : $\forall l \in \Theta, \sigma(l) =$ indéfini

pour tout $l \in \Theta$ **faire**
  $\Omega \leftarrow \{c \in \mathcal{C} \mid l \in \omega(c)\}$
  /*Ω est l'ensemble des clauses où l est surveillé*/
  tant que $\sigma(l) =$ indéfini **et** $\Omega \neq \emptyset$ **faire**
    choisir $c \in \Omega$
    $\Omega \leftarrow \Omega \backslash \{c\}$
    si $\forall l' \in c \setminus \{l\}, \sigma(l') =$ faux **alors**
      INSTANCIER(l, c) /*l est repropagé par c*/
    sinon /*rétablir la surveillance entière de c si besoin*/
      choisir $l' \in c \setminus \{l\} \mid \sigma(l') \neq$ faux
      $\omega(c) \leftarrow \{l, l'\}$

d'analyse, de saut arrière et de repropagation jusqu'à ce que l'opposé de l'assertion ne soit plus repropagé et que donc l'assertion puisse être propagée.

6.8.2 Détection anticipée des propagations alternatives

La méthode de détection tardive des propagations alternatives requiert de parcourir pour chaque littéral l défait l'ensemble des clauses où il est surveillé. Ce comportement est très proche de celui de la procédure de propagation unitaire, qui pour chaque littéral instancié l vérifie l'ensemble des clauses où son opposé $\neg l$ est surveillé. Or, la propagation unitaire est de loin l'opération la plus coûteuse lors d'un CDCL. Une telle tactique de découverte des propagations alternatives revient donc à ajouter une nouvelle étape avec un coût similaire aux propagations.

Le nombre de propagations alternatives potentielles à vérifier lors de la désinstanciation d'un littéral l peut être réduit en effectuant en cours de recherche une présélection des clauses qui, parmi les clauses où l est surveillé, peuvent être des propagations alternatives pour l.

Il est possible de réduire le coût de cette procédure en essayant de ne pas parcourir toutes les clauses où l est surveillée, mais seulement celles qui ne pourront pas être ultérieurement détectées par le mécanisme de propagation classique. Il s'agit donc des

clauses où le second littéral surveillé w est faux et a déjà été propagé. Dans ce cas, on peut mémoriser la clause lorsqu'elle est parcourue lors de la propagation de w. Lorsque l sera désinstancié, il suffira de chercher une éventuelle propagation alternative dans l'ensemble des clauses précédemment mémorisées, que nous noterons $\Xi(l)$.

Toutefois, l'ensemble des clauses où l est surveillé et le second littéral surveillé est faux reste possiblement très grand ; en particulier, les littéraux non-surveillés de telles clauses peuvent être indéfinis ou vrais, et donc ne pourront pas repropager l si celui-ci est désinstancié. Pour réduire cet ensemble, il est possible de modifier la procédure de propagation unitaire afin de forcer le remplacement des littéraux surveillés faux tant que possible, sans tenir compte du statut du second littéral surveillé. Une clause c sera alors ajoutée à $\Xi(l)$ seulement si l est vrai et surveillé dans c et si w, le second littéral surveillé dans c, est faux et ne peut pas être remplacé. Cette modification de la procédure de propagation unitaire permet donc de réduire la taille des ensembles de propagations alternatives possibles en contrepartie d'un surcoût lors des propagations, puisque l'on remplace les littéraux surveillés faux dans un plus grand nombre de cas.

La détection anticipée des propagations alternatives consiste donc, pour chaque littéral l, à mémoriser toutes les clauses rencontrées lors de propagations unitaires où l est le seul littéral vrai et tous les autres sont faux. Il est néanmoins possible qu'entre le moment où une clause c est ajoutée à $\Xi(l)$ et celui où l est défait, d'autres littéraux de c aient été défaits. Dans ce cas, c n'est bien sûr plus une propagation alternative valide pour l. Les propagations alternatives qui ne sont plus valides ne sont pas détectées avant la désinstanciation de l, car une stratégie de mise à jour systématique serait bien plus complexe et coûteuse.

La stratégie de détection anticipée des propagations alternatives est décrite par l'algorithme 6.13, noté CDCL-SSA-PropAltAnt. La seule différence avec la stratégie de détection tardive au niveau de la description globale de l'algorithme est l'initialisation des ensembles $\Xi(l)$ pour tous les littéraux $l \in \mathcal{L}$. Les différences principales se situent au niveau de la procédure PROPAGER (algorithme 6.14) : pour tout littéral faux l surveillé

Algorithme 6.13 CDCL-SSA-PropAltAnt

```
1: Λ ← 0
2: λc ← 0 /*on commence au pseudo-niveau de décision 0*/
3: σ ← ∅ /*on commence avec l'affectation vide*/
4: ∀l ∈ L, Ξ(l) = ∅ /*pas de propagations alternatives initialement*/
5: boucle
6:     c ← PROPAGER() /*propager les clauses unitaires*/
7:     si c ≠ NUL alors /*un conflit a été découvert*/
8:         λγ ← λ(c) /*calcul du niveau de conflit*/
9:         si λγ = 0 alors /*conflit au niveau de décision 0*/
10:            retourner faux /*F est insatisfaisable*/
11:        sinon
12:            répéter
13:                γ ← ANALYSER(c) /*déduire la clause de conflit γ*/
14:                a ← l ∈ γ | λ(a) = λγ
15:                /*a est unique; c'est la future assertion*/
16:                c ← γ /*γ sera responsable du prochain conflit*/
17:                /*si a n'est pas désinstancié*/
18:                Θ ← RÉSOUDRECONFLIT(λγ)
19:                /*défaire λγ et les propagations qui en dépendent*/
20:                /*Θ contient la liste des littéraux défaits*/
21:                C ← C ∪ {γ} /*γ est apprise*/
22:                REPROPAGER(Θ)
23:                /*essayer de repropager les littéraux défaits*/
24:            jusqu'à σ(a) = indéfini
25:            PROPAGERASSERTION(a, γ)
26:    sinon /*aucun conflit pendant les propagations*/
27:        si D(σ) = V alors /*toutes les variables sont instanciées*/
28:            retourner σ /*σ est un modèle de F*/
29:        sinon
30:            λc ← NOUVEAUNIVEAU()
31:            Λ ← Λ ∪ {λc}
32:            choisir v ∈ V\D(σ) /*v est une variable non-instanciée*/
33:            choisir l ∈ {v, ¬v}
34:            INSTANCIER(l, NUL)
35:            /*l est une décision, il n'a pas d'antécédent*/
```

Algorithme 6.14 PROPAGER() *[CDCL-SSA-PropAltAnt]*

```
1: Π ← {l ∈ σ | propagé(l) = faux}
2: tant que Π ≠ ∅ faire
3:     choisir l ∈ Π
4:     pour c ∈ C | ¬l ∈ ω(c) faire /*¬l est surveillé dans c*/
5:         w ← ω(c) \ {¬l} /*w est le second littéral surveillé dans c*/
6:         Ω ← {l' ∈ c \ {w} | σ(l') ≠ faux}
7:         /*Ω est l'ensemble des littéraux pouvant remplacer ¬l*/
8:         si Ω ≠ ∅ alors
9:             choisir w'∈Ω
10:            ω(c) ← {w, w'} /*w' est surveillé à la place de ¬l*/
11:        sinon /*tous les littéraux de c \ {¬l} sont faux*/
12:            si σ(w) = vrai alors /*w est le seul littéral non-faux dans c*/
13:                Ξ(w) ← Ξ(w) ∪ {c}
14:                /*c est une propagation alternative pour w*/
15:            sinon si σ(w) = indéfini alors /*c est unitaire*/
16:                INSTANCIER(w, c) /*w est propagé par c*/
17:                Π ← Π ∪ {w} /*w doit être lui-même propagé*/
18:            sinon
19:                retourner c /*c est un conflit*/
20:    Π ← Π \ {l}
21:    propagé(l) ← vrai
22: retourner NUL /*aucun conflit rencontré*/
```

dans une clause c, un remplaçant est cherché quelque soit la valeur de w, le second littéral surveillé. Si aucun remplaçant n'est trouvé et que w est vrai, c est ajouté à $\Xi(w)$. La procédure REPROPAGER est également modifiée (algorithme 6.15) : les propagations alternatives potentielles d'un littéral désinstancié l sont cherchées dans $\Xi(l)$ et non pas dans l'ensemble des clauses où l est surveillé. Lorsqu'une clause vérifiée n'est pas une propagation alternative valide pour l, elle est retirée de $\Xi(l)$; toutefois, si une propagation alternative valide est trouvée, les clauses non-vérifiées sont conservées dans $\Xi(l)$, car elles pourraient être valides lors d'une nouvelle désinstanciation ultérieure de l. Comme dans le cas de la détection tardive des propagations alternatives, la surveillance entière est assurée en remplaçant tout littéral surveillé faux dans chaque clause parcourue qui n'est pas une propagation alternative valide.

Algorithme 6.15 REPROPAGER(Θ) *[CDCL-SSA-PropAltAnt]*

Requis : $\forall l \in \Theta, \sigma(l) =$ indéfini

pour tout $l \in \Theta$ **faire**
  tant que $\sigma(l) =$ indéfini **et** $\Xi(l) \neq \emptyset$ **faire**
    choisir $c \in \Xi(l)$
    $\Xi(l) \leftarrow \Xi(l) \setminus \{c\}$
    si $\forall l' \in c \setminus \{l\}, \sigma(l') =$ faux **alors**
      INSTANCIER(l, c) /* l est repropagé par c */
    sinon /* rétablir la surveillance entière de c si besoin */
      choisir $l' \in c \setminus \{l\} \,|\, \sigma(l') \neq$ faux
      $\omega(c) \leftarrow \{l, l'\}$

6.8.3 Propriétés du CDCL sans saut arrière à propagations alternatives

La caractéristique principale des algorithmes de CDCL sans saut arrière à détection de propagations alternatives est l'assurance d'une surveillance entière des clauses en tout temps.

Proposition 6.6. *CDCL-SSA-PropAltTard et CDCL-SSA-PropAltAnt sont à surveillance entière.*

Démonstration. Nous allons montrer que si toutes les clauses sont entièrement surveillées, elles le seront toujours après instanciation d'un littéral ou résolution d'un conflit.

Supposons qu'un littéral l est instancié. Cette instanciation n'a aucune incidence sur la surveillance des clauses où ni l, ni $\neg l$ ne sont surveillées. Si c est une clause entièrement surveillée où l est surveillé, elle restera entièrement surveillée après l'instanciation. Soit c une clause entièrement surveillée où $\neg l$ est surveillé. Soit w le second littéral surveillé, qui est soit vrai, soit indéfini, soit faux mais pas encore propagé. Si w est vrai, c est toujours entièrement surveillée. Sinon, s'il existe un littéral vrai ou indéfini dans $c \setminus \{l, w\}$, il est surveillé à la place de l est c reste entièrement surveillée. Sinon, si w est indéfini, alors c est unitaire et w est propagée ; c reste donc entièrement surveillée. Enfin, si w est faux, alors c est fausse et un conflit est déclenché. Comme c était entièrement surveillée avant l'instanciation de l, w n'est pas encore propagée. Comme la propagation

de l a été interrompue par un conflit, il reste également considéré comme non-propagé. c reste donc entièrement surveillée même si ni l ni w ne sont défaites par la résolution du conflit.

Supposons maintenant qu'une résolution de conflit a lieu. Toutes les clauses dont les deux littéraux surveillés sont soit vrais, soit indéfinis, soit faux mais pas encore propagés restent entièrement surveillées, quelque soient les littéraux de la clause qui sont désinstanciés. Soit c une clause dont le premier littéral surveillé w_1 est vrai et le second w_2 est faux et déjà propagé. Si w_2 est désinstancié, c reste entièrement surveillée. Si w_1 est désinstancié mais pas w_2, c n'est plus correctement surveillée ; cependant, c est alors détectée comme une propagation alternative pour w_1 quel que soit le mode de détection utilisé. Si c est unitaire, w_1 est repropagé, ce qui rétablit la surveillance entière de c. Sinon, les littéraux surveillés de c sont ajustés afin de rétablir la surveillance entière. □

Corollaire 6.2. *CDCL-SSA-PropAltTard et CDCL-SSA-PropAltAnt sont à propagations exhaustives.*

Démonstration. Par les propositions 6.6 et 5.3. □

La détection des propagations alternatives permet donc d'assurer l'exhaustivité des propagations dans le CDCL sans saut arrière. Toutefois, cette propriété ne suffit pas à rendre l'algorithme non-trivial et non-redondant. En effet, si la présence de conflits triviaux et redondants dans le CDCL sans saut arrière de base est en partie due à la non-exhaustivité des propagations, elle a également sa source dans une caractéristique fondamentale de l'algorithme : la non-unicité des niveaux de propagation. En effet, comme nous l'avons mentionné dans la section 5.4, cette non-unicité rend quasiment impossible d'assurer qu'aucun conflit ne peut être trivial. Nous allons prouver que nos deux variantes de CDCL-SSA à propagations alternatives ne sont effectivement ni non-triviales, ni non-redondantes.

Proposition 6.7. *CDCL-SSA-PropAltTard n'est pas non-trivial ni non-redondant.*

Démonstration. Considérons la formule propositionnelle formée par les 4 variables a, b, c et d et les 5 clauses $c_1 = a \vee \neg b$, $c_2 = \neg a \vee \neg c \vee \neg d$, $c_3 = \neg a \vee \neg c \vee d$, $c_4 = \neg a \vee c \vee \neg d$ et $c_5 = \neg a \vee c \vee d$. Supposons que chaque clause est initialement surveillée par ses deux derniers littéraux et que la suite d'opérations suivantes est effectuée :

- Le littéral a est instancié comme décision de niveau 1.
- Le littéral b est instancié comme décision de niveau 2.
- Le littéral c est instancié comme décision de niveau 3. Les clauses c_2 et c_3 deviennent unitaires. Supposons que c_2 est détectée la première ; alors $\neg d$ est instancié comme propagation au niveau 3, ce qui rend la clause c_3 fausse. La clause de conflit est $c_6 = \{\neg a \vee \neg c\}$ de niveau de conflit 3 et de niveau d'assertion 1.
- Les instanciations c et $\neg d$ sont défaites par la résolution de conflit et ne sont pas réinstanciées par propagation unitaire car aucune des clauses où ces littéraux sont respectivement surveillés n'est unitaire. Le littéral $\neg c$ est instancié comme propagation de la clause de conflit c_6 au niveau 1. Les clauses c_4 et c_5 deviennent unitaires. Supposons que c_4 est détectée la première ; alors $\neg d$ est instanciée comme propagation au niveau 1, ce qui rend c_5 fausse. La clause de conflit est $c_7 = \neg a$, de niveau de conflit 1 et de niveau d'assertion 0.
- Les instanciations a et $\neg c$ sont défaites par la résolution de conflit. La clause c_1 est une propagation alternative pour a, qui est donc réinstancié comme propagation au niveau 2. Le littéral $\neg c$ n'est pas réinstancié car aucune clause où il est surveillé n'est unitaire.
- La réinstanciation de a a rendue fausse la dernière clause de conflit c_7, ce qui déclenche donc un nouveau conflit en cascade. Comme c_7 contient un seul littéral, elle contient un seul littéral de niveau maximal ; le conflit est donc trivial. La nouvelle clause de conflit est identique à c_7 puisque CDCL-SSA-PropAltTard utilise la stratégie d'apprentissage du premier point d'implication unique ; le conflit est également redondant.

□

Proposition 6.8. *CDCL-SSA-PropAltAnt n'est pas non-trivial ni non-redondant.*

Démonstration. On peut utiliser le même contre-exemple que dans la preuve de la proposition 6.7 ; elle reste entièrement valide dans le cas du CDCL sans saut arrière à détection anticipée des propagations alternatives. La seule différence est dans la gestion anticipée des détections. En particulier, lorsque b est instancié en tant que décision du niveau 2, la clause c_1 est ajoutée à $\Xi(a)$. Ultérieurement, lorsque a est désinstancié, c_1 est le seul élément de $\Xi(a)$ et est toujours une propagation alternative valide pour a, donc la repropagation a lieu de la même façon. □

Une modification de CDCL-SSA rendant la surveillance des clauses entière en tout temps ne suffit donc pas à rendre l'algorithme non-trivial ni non-redondant. Toutefois, la non-unicité du niveau de propagation et la partialité des surveillances de clauses sont deux causes distinctes, quoique parfois concomitantes, de conflits triviaux. Si la surveillance entière ne permet donc pas d'empêcher totalement les conflits triviaux, elle peut toutefois en réduire la fréquence, comme le démontrent les résultats expérimentaux de la sous-section suivante.

6.8.4 Comparaison expérimentale

Les résultats expérimentaux obtenus par nos implémentations du CDCL-SSA à détection respectivement tardive et anticipée des propagations alternatives sont présentés par les tableaux 6.1 à 6.7. Le premier constat est que les performances en temps et en nombre d'instances résolues sont dégradées par rapport aux performances de CDCL-SSA-Parcours. Cette dégradation est très marquée dans le cas de la détection tardive des propagations alternatives, mais plus mesurée dans celui de la détection anticipée, dont les performances sont similaires à celles de CDCL-SSA-Graphe.

Ces mauvaises performances du point de vue du temps de résolution s'expliquent facilement par le surcoût de travail causé par la gestion des propagations alternatives. En effet, pour les deux implémentations, la fréquence moyenne des vérifications de clauses

(au cours des propagations ordinaires) est inférieure à 40% de la fréquence de ces vérifications pour CDCL-SSA-Parcours. Cela confirme notre intuition de comparer la charge de travail de la gestion des propagations alternatives à la charge causée par les propagations ordinaires. La fréquence des vérifications de clause est toutefois environ 20% plus élevée lorsque la détection des propagations est anticipée ; il est donc en pratique utile de réduire le nombre de clauses vérifiées lors de la désinstanciation d'une variable, malgré le coût de gestion des ensembles de propagations alternatives candidates au cours de la recherche et de la surveillance accrue des clauses déjà satisfaites.

La baisse des performances en temps par rapport à CDCL-SSA-Parcours est relativement modérée en comparaison de la baisse de fréquence de traitement des étapes de propagation. Cela peut s'expliquer par deux phénomènes complémentaires. D'une part, les propagations alternatives permettent de réduire significativement le nombre de vérifications de clauses nécessaires à compléter la résolution dans la plupart des cas, particulièrement lorsque les détections sont anticipées. Ainsi, CDCL-SSA-PropAltTard et CDCL-SSA-PropAltAnt réduisent le nombre d'étapes de propagation par rapport à CDCL-SSA-Graphe, CDCL-SSA-Parcours et CDCL-SSA_{LB} dans au moins 70% des instances résolues en commun, et elles réduisent fortement le nombre total d'étapes de propagation sur l'ensemble de ces instances. Si la comparaison est moins favorable par rapport à CDCL_{LB} et CDCL, CDCL-SSA-PropAltTard et CDCL-SSA-PropAltAnt parviennent toutefois à réduire le nombre total d'étapes de propagation sur les instances résolues en commun pour ces deux variantes de CDCL classique, et CDCL-SSA-PropAltAnt diminue le nombre de vérifications de clauses par rapport à CDCL_{LB} dans exactement la moitié des instances qu'ils parviennent tous deux à résoudre, soit 24 sur 48.

La seconde explication est que l'exhaustivité des propagations, si elle ne suffit pas à totalement éliminer les conflits triviaux et redondants, parvient toutefois à les réduire d'un peu plus de moitié par rapport aux trois implémentations de CDCL-SSA sans gestion des propagations alternatives. Les vérifications de clauses, en plus d'être un peu moins nombreuses, sont donc aussi plus souvent utilisées dans des parties de la re-

cherche qui sont utiles et contribuent à l'élagage de l'espace de recherche. De plus, dans le cas des propagations alternatives anticipées, la distance moyenne entre les conflits reste comparable à celle du CDCL-SSA sans propagations alternatives (pour une raison indéterminée, cette distance augmente significativement dans le cas de la détection tardive des propagations alternatives). Cela signifie donc que la distance moyenne entre les conflits utiles diminue, à la fois par rapport au CDCL-SSA sans propagations alternatives et au CDCL classique, ce qui explique aussi en partie les bonnes performances de CDCL-SSA-PropAltAnt en termes de taille des exécutions.

On peut remarquer que la destructivité moyenne des conflits est sensiblement plus élevée pour les deux variantes à propagations alternatives que pour les trois implémentations sans propagations alternatives ; elle reste toutefois considérablement plus basse que pour le CDCL classique.

En résumé, la gestion des propagations alternatives permet de réduire le nombre d'étapes de propagation nécessaires à la résolution des instances par un CDCL sans saut arrière, notamment en réduisant de moitié le taux de conflits triviaux. Le nombre d'étapes de propagation reste cependant souvent comparable ou supérieur à celui nécessité par un CDCL ordinaire. Enfin, la lourdeur de la gestion des propagations alternatives handicape fortement les performances en temps de l'algorithme et le rend moins efficace que le CDCL sans saut arrière de base.

6.9 Conservation additionnelle d'instanciations

Les stratégies de gestion des propagations alternatives que nous avons proposées ont l'inconvénient de causer une désinstanciation puis une réinstanciation du même littéral, ainsi que, potentiellement, d'une suite de propagations identiques. Pour éviter la perte de temps causée par ce travail redondant, il est possible dans certains cas d'être sûr que l'instanciation pourra être refaite avant même de la défaire. On peut alors simplement conserver l'instanciation et modifier son antécédent. En procédant ainsi, on peut également éviter de défaire les propagations qui dépendent de ce littéral et

Algorithme 6.16 RÉSOUDRECONFLIT(λ_γ) *[CDCL-SSA-Conservation]*

```
1: Θ ← ∅  /*variables défaites pendant le traitement*/
2: pour  λ_γ ≤ i ≤ λ_c  faire
3:     /*on parcourt tous les niveaux à partir du niveau de conflit*/
4:     pour  l ∈ σ | λ(l) = i  faire
5:         défaire_l ← faux
6:         /*parcours des littéraux du niveau dans l'ordre d'instanciation*/
7:         si  i = λ_γ  alors
8:             /*défaire toutes les instanciations du niveau de conflit*/
9:             défaire_l ← vrai
10:        sinon si  ∃l' ∈ α(l) | σ(l') = indéfini  alors
11:            /*défaire l si un de ses antécédents a été défait*/
12:            défaire_l ← vrai
13:        si  défaire_l = vrai et CONSERVER(l) = faux  alors
14:            /*la tentative de conserver l a échoué*/
15:            DÉFAIRE(l)
16:            Θ ← Θ ∪ {l}
17:        si  l ∈ D(σ) et λ(l) ≠ λ(α(l) \ {l})  alors
18:            /*si l n'est pas défait, ajuster son niveau si besoin*/
19:            DÉPLACER(ν(l))
20: Λ ← Λ \ {λ_γ}  /*λ_γ est le seul niveau entièrement défait*/
21: retourner Θ
```

dont aucun autre antécédent n'est défait. Nous appellerons CDCL-SSA-Conservation la variante de CDCL-SSA-PropAltAnt implémentant cette tactique de conservation des instanciations à défaire. La description globale de l'algorithme, ainsi que la procédure de propagation unitaire, restent identiques aux algorithmes 6.13 et 6.14 respectivement. La procédure RÉSOUDRECONFLIT est modifiée de la façon indiquée par l'algorithme 6.16. La différence principale avec la variante implémentant uniquement les repropagations (algorithme 6.11) est qu'à la ligne 13, on peut éviter de défaire le littéral l si la procédure CONSERVER(l), décrite par l'algorithme 6.17, retourne vrai.

Toutes les clauses pouvant empêcher la désinstanciation de l font partie de l'ensemble de clauses de $\Xi(l)$, il suffit donc de le parcourir avant de commencer les désinstanciations de variables. Une telle clause restera une propagation valide pour l après la résolution du conflit si et seulement si aucun autre littéral de la clause n'est désinstancié. Une condition suffisante (mais non nécessaire) pour cela est que tous les littéraux autres

que l soient d'un niveau de décision strictement inférieur au niveau de conflit. Pour des raisons d'efficacité, nous nous restreindrons à éviter de défaire des instanciations dans ces cas de figures uniquement.

Algorithme 6.17 CONSERVER(l) *[CDCL-SSA-Conservation]*

Requis : $\sigma(l) = \text{vrai}$
1: $\Omega \leftarrow \emptyset$ /*Ω est l'ensemble des clauses rejetées pour le déplacement de l*/
2: /*mais qui restent des propagations alternatives valides*/
3: $\alpha' \leftarrow \text{NUL}$ /*α' est un nouvel antécédent pour l*/
4: **tant que** $\alpha' = \text{NUL}$ **et** $\Xi(l) \neq \emptyset$ **faire**
5: **choisir** $c \in \Xi(l)$
6: $\Xi(l) \leftarrow \Xi(l) \setminus \{c\}$
7: $c' \leftarrow c \setminus \{l\}$
8: **si** $\forall l' \in c', \sigma(l') = \text{faux}$ **alors**
9: **si** $\lambda(c') < \lambda_\gamma$ **alors**
10: $\alpha(l) \leftarrow \alpha' \leftarrow c$
11: DÉPLACER($\nu(l)$)
12: /*$\lambda(l) \neq \lambda(c')$ car $\lambda(c') < \lambda_\gamma \leq \lambda(l)$*/
13: **sinon**
14: $\Omega \leftarrow \Omega \cup \{c\}$
15: $\Xi(l) \leftarrow \Xi(l) \cup \Omega$
16: **retourner** $\alpha' \neq \text{NUL}$
17: /*retourne vrai si on a trouvé un autre antécédent pour l*/

Cette stratégie d'évitement des désinstanciations, décrite par l'algorithme 6.17, peut donc rejeter des propagations alternatives valides (des clauses contenant des littéraux de niveau supérieur au niveau de conflit mais qui ne seront pas défaits). Cette étape ne détecte pas toutes les propagations alternatives possibles. Pour conserver une exhaustivité des propagations, il convient donc d'effectuer une recherche classique des propagations alternatives après la résolution du conflit. Le nombre de clauses parcourues lors de cette seconde recherche sera cependant bien plus restreint, puisque la première recherche élimine de l'ensemble des clauses candidates toutes les clauses qui n'étaient pas unitaires avant les désinstanciations de littéraux.

Notons qu'en changeant d'antécédent, le littéral conservé change obligatoirement de niveau de décision. En effet, son niveau était précédemment supérieur ou égal au niveau de conflit λ_γ, puisque le saut arrière ne peut en aucun cas défaire une instanciation

Algorithme 6.18 DÉPLACER(v) *[CDCL-SSA-Conservation]*

Requis : $\sigma(v) \neq$ indéfini, $\lambda(v) \neq \lambda(\alpha(v) \setminus \{v\})$

1: $\lambda(v) \leftarrow \lambda(\alpha(v) \setminus \{v\})$

2: /*v est déplacée vers le plus grand niveau de décision parmi ses antécédents*/

d'un niveau inférieur au niveau de conflit. Cependant, si le littéral a bien été conservé, cela signifie que tous les autres littéraux de son nouvel antécédent sont de niveaux strictement inférieurs à λ_γ. Le littéral conservé va donc être déplacé au plus grand de ces niveaux par la procédure DÉPLACER à la ligne 11 de l'algorithme 6.17. La procédure DÉPLACER est détaillée par l'algorithme 6.18.

Le déplacement d'un littéral conservé peut à son tour modifier le niveau de décision des propagations qui en dépendent, et ainsi de suite récursivement. La procédure RÉSOUDRECONFLIT parcourt obligatoirement un littéral après tous ses antécédents. La cohérence des niveaux de décision peut donc être simplement assurée en vérifiant pour tout littéral parcouru et non désinstancié si son niveau de décision correspond toujours au niveau le plus élevé parmi ses antécédents, et en le déplaçant si nécessaire. Cette vérification est assurée par les lignes 17 à 19 de l'algorithme 6.16.

Les propriétés de CDCL-SSA-Conservation restent identiques à celles de CDCL-SSA-PropAltTard et CDCL-SSA-PropAltAnt ; les résultats expérimentaux de son implémentation sont présentés dans les tableaux 6.1 à 6.7. Les performances en temps d'exécution et en nombre d'instances résolues sont clairement dégradées par rapport à CDCL-SSA-PropAltAnt, tout en restant légèrement supérieures à celles de CDCL-SSA-PropAltTard. CDCL-SSA-Conservation parvient à résoudre 4 instances de moins que CDCL-SSA-PropAltAnt dans le temps imparti, ce qui s'explique par le surcoût de détection et d'exécution des déplacements, qui provoque une baisse sensible de la fréquence des vérifications de clauses (environ 5% par rapport à CDCL-SSA-PropAltAnt). La proportion de désinstanciations évitées (qui n'est pas présentée dans les tableaux) est trop faible pour compenser ce surcoût : en moyenne, seules 0,21% des désinstanciations sont évitées au cours d'une exécution de CDCL-SSA-Conservation, pour un maximum de 0,91%. Ce faible taux de conservation n'a donc pas d'impact significatif sur le nombre

d'étapes de propagation nécessaires à la résolution des instances, qui est par exemple à peine inférieur à celui de CDCL-SSA-PropAltAnt sur le total des instances résolues par les deux implémentations. La destructivité des conflits, la distance entre les conflits et la proportion de conflits redondants restent aussi quasiment identiques.

Au final, si CDCL-SSA-Conservation permet en principe de réduire la taille des exécutions en évitant des désinstanciations temporaires dans le CDCL sans saut arrière à propagations alternatives, la proportion des désinstanciations concernées est en pratique bien trop faible pour compenser le surcoût de traitement occasionné.

6.10 Conclusion

Ce chapitre a présenté diverses variantes de l'algorithme CDCL sans saut arrière. Cette variation du CDCL part du constat que le saut arrière jusqu'au niveau d'assertion n'est pas essentiel à la correction de l'algorithme et qu'il suffit de défaire le niveau de conflit, ainsi que les propagations qui en dépendent, pour résoudre le conflit et pouvoir propager l'assertion.

Nous avons introduit trois variantes principales de l'algorithme qui diffèrent par leurs propriétés. La version basique de l'algorithme assure une détection exhaustive des conflits, mais pas des clauses unitaires ; elle nécessite toutefois de désactiver le mécanisme des littéraux bloqués. L'implémentation la plus efficace de cette version utilise un parcours des niveaux de décision supérieurs pour la résolution de conflits. Une seconde variante est obtenue en réintroduisant les littéraux bloqués, ce qui améliore les performances expérimentales de l'algorithme, en contrepartie de ne plus permettre la détection exhaustive des conflits. Enfin, une troisième variante, au contraire, renforce les propriétés de la version basique et permet une détection exhaustive à la fois des conflits et des clauses unitaires, à l'aide de la détection des propagations alternatives. L'implémentation la plus efficace de cette stratégie utilise une détection anticipée. L'exhaustivité des propagations ne suffit pas à empêcher l'occurrence de conflits triviaux et redondants mais en réduit fortement l'occurrence.

Malheureusement, les variantes aux propriétés les plus complètes sont aussi les moins efficaces en pratiques, en raison du surcoût nécessaire à la maintenance de ces propriétés. Ainsi, le CDCL-SSA à détection de propagations alternatives est moins efficace que le CDCL-SSA de base, lui-même surpassé par le CDCL-SSA à littéraux bloqués. Au final, mêmes les variantes les plus efficaces du CDCL sans saut arrière restent en pratique moins efficaces que le CDCL ordinaire dans la majorité des cas. Cette perte de performance est sans doute principalement due au coût accru de certaines opérations très fréquentes au cours de l'algorithme, comme les propagations unitaires, qui nécessitent la détermination du niveau de décision où le littéral doit être propagé, ou encore la résolution du conflit, qui requiert une analyse plus complexe que dans le CDCL classique pour déterminer les variables à défaire. Une partie de la perte de performance peut aussi s'expliquer par des interactions complexes avec d'autres composantes du CDCL, comme les heuristiques de décision ou l'impact de l'absence de sauvegarde de phase sur les redémarrages. Les gains éventuels découlant de la réduction de destructivité des résolutions de conflit ne sont probablement pas assez importants pour compenser ces surcoûts.

En résumé, si les différentes variantes du CDCL sans saut arrière sont intéressantes du point de vue théorique, les implémentations de l'état de l'art actuel du CDCL classique sont si optimisées que le CDCL sans saut arrière ne semble pas compétitif d'un point de vue pratique, à moins de trouver des optimisations conséquentes pour son implémentation.

CHAPITRE VII

CDCL À ORDRE PARTIEL

Ce chapitre présente une nouvelle variation de l'algorithme CDCL qui introduit, comme son nom l'indique, un ordre partiel sur les niveaux de décision au cours de l'exécution. Son but, tout comme celui du CDCL sans saut arrière, présenté au chapitre précédent, est de réduire la perte de l'état courant de la recherche causée par l'étape de saut arrière du CDCL. La stratégie utilisée est toutefois différente.

Le CDCL sans saut arrière constate que pour résoudre un conflit, il suffit de désinstancier le niveau de décision où il a eu lieu, et éventuellement les instanciations d'autres niveaux propagées à l'aide du niveau de conflit. Cette technique a l'avantage de défaire considérablement moins d'instanciations que le CDCL classique lors d'un conflit. Elle réduit cependant significativement l'efficacité en pratique de l'algorithme.

Le CDCL à ordre partiel, quant à lui, conserve la notion de saut arrière. Cependant, contrairement au CDCL classique, le CDCL à ordre partiel tient compte des interactions entre les différents niveaux de décision et son saut arrière ne défait pas systématiquement tous les niveaux de décision créés après le niveau d'assertion, mais uniquement ceux qui dépendent de ce niveau d'assertion.

Cette stratégie permet de sauver moins d'instanciations lors de la résolution d'un conflit que le CDCL à saut arrière, mais est bien plus efficace en pratique. En fait, contrairement au CDCL sans saut arrière, le CDCL à ordre partiel peut aussi bien réduire qu'augmenter la destructivité des conflits ; or, nous verrons qu'en pratique les

deux cas peuvent permettre d'améliorer l'efficacité de résolution selon les instances. Par conséquent, le CDCL à ordre partiel permet de concurrencer, voire dans certains cas de significativement améliorer la résolution d'instances applicatives par rapport au CDCL classique.

En outre, l'ordre partiel sur les niveaux de décision a pour conséquence que, lors d'un conflit, le niveau d'assertion n'est plus défini uniquement, ce qui permet dans certains cas de choisir ce niveau d'assertion parmi plusieurs candidats possibles. Nos résultats expérimentaux montrent que ce choix a un impact significatif sur les performances et permet de considérablement accélérer l'algorithme sur certaines instances.

Les sections 7.1 et 7.2 proposent respectivement une description informelle et algorithmique du CDCL à ordre partiel. La section 7.3 compare le concept de CDCL à ordre partiel avec les autres méthodes de réduction de la destructivité des sauts arrière. La section 7.4 établit ses propriétés théoriques. La section 7.5 détaille l'implémentation de la gestion des dépendances entre niveaux de décision. La section 7.6 évalue expérimentalement l'implémentation du CDCL à ordre partiel . La section 7.7 introduit des variantes du CDCL à ordre partiel qui assurent l'exhaustivité des propagations, démontre leurs propriétés et évalue expérimentalement leur efficacité. La section 7.8 explique l'importance de la densité des dépendances entre niveaux de décision sur les performances de l'algorithme et l'illustre par des résultats expérimentaux sur des familles d'instances à faible densité de dépendances. La section 7.9 propose différentes heuristiques pour le choix du niveau d'assertion et vérifie leur efficacité en pratique. Enfin, la section 7.10 conclut le chapitre en résumant ses principaux résultats.

7.1 Description informelle

Cette première section introduit le CDCL à ordre partiel par une description informelle de l'algorithme et de ses principales caractéristiques. La sous-section 7.1.1 motive la conception de l'algorithme par rapport au CDCL ordinaire et au CDCL sans saut arrière et donne une description globale de son fonctionnement. Les sous-sections 7.1.2 à 7.1.4

abordent chacune un aspect plus précis de l'algorithme, respectivement son impact sur la destructivité des sauts arrière, l'occurrence de choix multiples pour les niveaux d'assertion et le cas particulier du pseudo-niveau de décision 0. Enfin, la sous-section 7.1.5 illustre l'algorithme par un exemple d'exécution.

7.1.1 Motivation et principe

Le chapitre 6 a introduit le CDCL sans saut arrière, qui, contrairement au CDCL classique, permet de résoudre un conflit en défaisant uniquement le niveau de conflit et les propagations d'autres niveaux qui en dépendent. Le CDCL sans saut arrière présente cependant deux inconvénients principaux qui handicapent ses performances. Le premier est de devoir tracer ou retrouver les dépendances exactes entre les différentes variables du problème, qui alourdit considérablement la gestion des propagations unitaires ou des résolutions de conflits, selon la stratégie utilisée. Le second est de permettre des propagations à des niveaux de décision quelconques, ce qui complexifie la gestion des propagations par rapport au CDCL ordinaire, où toute propagation ne peut avoir lieu qu'au niveau de décision maximal courant; de plus, cette caractéristique cause l'occurrence de conflits triviaux et redondants. Si le CDCL sans saut arrière réduit de façon très importante la destructivité des sauts arrière, c'est donc au prix d'une plus grande complexité qui l'empêche de rivaliser avec le CDCL ordinaire en matière de temps d'exécution.

L'algorithme que nous décrirons dans ce chapitre, le CDCL à ordre partiel (ou CDCL-OP), est en quelque sorte un compromis entre le CDCL ordinaire et le CDCL sans saut arrière. En effet, le CDCL à ordre partiel ne supprime pas totalement la notion de saut arrière et empêche donc moins de destructivité que le CDCL sans saut arrière, tout en pouvant la réduire par rapport au CDCL ordinaire. En contrepartie, le CDCL à ordre partiel nécessite uniquement de garder la trace des dépendances entre niveaux de décision, et non pas entre variables comme le CDCL sans saut arrière. Comme un niveau de décision peut contenir un très grand nombre d'instanciations, la gestion des dépendances en est grandement allégée. Enfin, le CDCL à ordre partiel conserve la

caractéristique d'unicité du niveau de propagation du CDCL ordinaire. Nous verrons que cela nous permet de concevoir des variantes non-triviales et non-redondantes du CDCL à ordre partiel.

Dans le CDCL sans saut arrière, la source de la non-unicité du niveau de propagation est le fait qu'après un conflit, la clause de conflit cause une propagation au niveau d'assertion λ_a alors qu'il reste possiblement des niveaux supérieurs à ce niveau d'assertion qui n'ont pas été défaits. Pour contourner ce problème, il est possible de réordonner dynamiquement les niveaux de décision avant la propagation, en supposant que le niveau d'assertion devient le nouveau niveau maximal. Un tel réordonnancement est possible s'il produit un état de recherche en profondeur cohérent. En fait, le nouvel état est cohérent si, avant le réordonnancement des niveaux, aucune propagation à un niveau de décision supérieur à λ_a n'a un littéral de niveau λ_a parmi ses antécédents. Dans le cas contraire, le réordonnancement placerait une propagation à un niveau de décision inférieur à celui d'un de ses antécédents, ce qui est évidemment absurde.

Pour obtenir un état cohérent après le réordonnancement, il serait possible de défaire toutes les instanciations dont un antécédent appartient au niveau λ_a, puis, récursivement, toutes les instanciations dont un antécédent a été défait. Toutefois, cela nous contraindrait à effectuer une gestion des dépendances entre variables, dont nous avons vu qu'elle handicape les performances du CDCL sans saut arrière. Nous allons donc opter pour une gestion des dépendances moins fine, au niveau des niveaux de décision. Ainsi, si un niveau de décision contient une propagation dont un antécédent appartient au niveau λ_a, ce niveau sera entièrement défait ; puis, récursivement, nous déferons tout niveau de décision contenant une propagation dont un antécédent a été défait. Cette gestion moins détaillée aura évidemment l'inconvénient de défaire un plus grand nombre d'instanciations pour permettre le réordonnancement dynamique. Une fois ce réordonnancement effectué, la clause de conflit peut être propagée et λ_a est le nouveau niveau maximal et niveau de propagation unique. Il s'agit du principe de base du CDCL à ordre partiel.

Afin de ne pas avoir à découvrir les relations de dépendances entre les niveaux à chaque conflit, celles-ci seront mises à jour au cours de la recherche sous la forme d'une relation $\Delta \subseteq \Lambda \times \Lambda$ (à ne pas confondre avec la relation $\Delta : \mathcal{V} \times \mathcal{V}$ du CDCL sans saut arrière qui formalise les dépendances entre instanciations). $\forall \{i, j\} \subseteq \Lambda$, $i \Delta j$ signifie que le niveau de décision j dépend directement du niveau de décision i, c'est-à-dire qu'une propagation du niveau j a un littéral du niveau i comme antécédent. Δ ne tient pas compte des antécédents appartenant au même niveau de décision que leur propagation, qui n'ont pas d'incidence sur la procédure de réordonnancement ; Δ est donc une relation irréflexive et asymétrique. Par conséquent, sa fermeture transitive Δ^+ est une relation d'ordre stricte sur Λ, et nous utiliserons la notation $i \prec_\Delta j$ comme équivalente à $i \Delta^+ j$. Lorsqu'un niveau de décision est créé au cours de la recherche, nous considérons qu'il ne dépend initialement d'aucun des niveaux de décision existants. Les dépendances entre niveaux de décision sont construites lors de propagations unitaires : si des littéraux $l_1, l_2, \ldots, l_{n-1}$ propagent un littéral l_n au niveau courant λ_c, alors λ_c dépend de $\lambda(l_1)$, $\lambda(l_2)$, ...,et $\lambda(l_{n-1})$ (pour les niveaux différents de λ_c).

Cette relation d'ordre permet alors d'effectuer un saut arrière partiel, qui correspond au réordonnancement dynamique avec désinstanciation des niveaux de décision non-compatibles que nous avons décrit précédemment : il n'y a pas besoin de défaire tous les niveaux créés chronologiquement après λ_a, mais uniquement les niveaux qui dépendent de λ_a, directement ou transitivement (ainsi que le niveau de conflit, c'est-à-dire le niveau courant λ_c, qui peut ne pas dépendre de λ_a).

7.1.2 Impact sur la destructivité des sauts arrière

Le réordonnancement dynamique des niveaux de décision a pour conséquence qu'un niveau de décision peut dépendre d'un autre niveau chronologiquement plus récent. En effet, soit $\tau : \Lambda \to \mathbb{N}$ une fonction injective qui associe à tout niveau de décision son instant de création exprimé dans une unité quelconque (par exemple, le nombre de décisions prises depuis le début de l'exécution). Supposons qu'il existe deux niveaux de décision i et j indépendants l'un de l'autre tels que $\tau(i) < \tau(j)$. Supposons maintenant

qu'un conflit survient à un niveau différent de j et que son niveau d'assertion est i. j n'est pas défait par le saut arrière partiel puisque $i \not\prec_\Delta j$. Par la suite, si une propagation unitaire implique le niveau de j pendant que i est le niveau de décision courant, alors la dépendance $j \prec_\Delta i$ est ajoutée, qui va donc à l'encontre de l'ordre chronologique de création des deux niveaux.

Si, par la suite, un conflit intervient à un niveau différent de i et j et que j est le niveau d'assertion, le saut arrière partiel provoque la destruction du niveau i, alors que celui-ci est chronologiquement antérieur au niveau j. Cet exemple peut donner l'impression que le saut arrière partiel est potentiellement plus destructif qu'un saut arrière conventionnel, puisque celui-ci défait uniquement des niveaux de décision chronologiquement ultérieurs au niveau d'assertion. Toutefois, il convient de remarquer que si i dépend de j alors que j est plus récent, cela signifie que j a préalablement été conservé lors d'un conflit précédent dont le niveau d'assertion était i, et donc que le même conflit géré par un saut arrière conventionnel aurait défait j. Par conséquent, tout niveau de décision défait par un saut arrière partiel alors qu'il est plus ancien que le niveau d'assertion implique que ce niveau d'assertion a préalablement été «sauvé» par un autre saut arrière partiel.

Par conséquent, si le saut arrière partiel est toujours moins destructif qu'un saut arrière ordinaire lors du premier conflit de la recherche, car à ce moment les dépendances respectent l'ordre chronologique des niveaux, la destructivité des deux types de saut arrière lors des conflits suivants est plus difficile à comparer, puisque chacun est susceptible de désinstancier certains niveaux qui seront conservés par l'autre. La différence principale reste que le CDCL à ordre partiel défait uniquement en plus du niveau de conflit des niveaux qui dépendent du niveau d'assertion, tandis que le CDCL classique défait tous les niveaux chronologiquement supérieurs au niveau d'assertion sans tenir compte de leur relation ou absence de relation avec ce niveau. Si la destructivité du CDCL à ordre partiel n'est donc pas forcément plus faible que celle du CDCL classique en quantité, le CDCL à ordre partiel a toutefois l'avantage de ne défaire aucun niveau sans lien avec le niveau d'assertion.

De plus, nous verrons dans la sous-section 7.9.2 que l'absence de garantie de réduction de la destructivité des conflits est un avantage dans certains cas, puisque le CDCL à ordre partiel peut améliorer les performances de résolution par rapport au CDCL classique sur certaines instances en augmentant cette destructivité. Le CDCL à ordre partiel permet donc finalement soit de réduire, soit d'augmenter cette destructivité selon les besoins, alors que le CDCL sans saut arrière ne permet que la réduction.

7.1.3 Choix multiples des niveaux d'assertion

L'ordre partiel sur les niveaux de décision a pour conséquence un assouplissement de la définition du niveau d'assertion. En effet, dans un CDCL classique, celui-ci est défini comme le plus grand niveau de décision représenté dans la clause de conflit, le niveau de conflit excepté. Avec un ordre total sur les niveaux de décision, le niveau d'assertion est défini de façon unique dans un CDCL classique. Ce n'est plus le cas avec un ordre partiel sur les niveaux de décision : l'ensemble des niveaux de la clause de conflit moins le niveau de conflit admet un ou plusieurs éléments maximaux, selon les cas. L'algorithme de CDCL à ordre partiel permet donc de choisir arbitrairement le niveau d'assertion parmi l'ensemble Γ de ces éléments maximaux, ce qui rajoute un degré de liberté par rapport au CDCL conventionnel.

Il serait théoriquement possible d'agrandir encore plus cette liberté et de permettre de choisir n'importe quel niveau de décision impliqué dans la clause de conflit, hormis le niveau de conflit, comme niveau d'assertion. Toutefois, si le niveau d'assertion λ_a n'appartient pas à Γ, cela signifie qu'au moins un niveau de la clause de conflit différent du niveau de conflit dépend de λ_a. Le saut arrière résolvant le conflit défait donc au moins deux niveaux dans la clause de conflit, qui n'est donc pas unitaire. Une telle liberté de choix du niveau d'assertion peut donc causer des sauts arrière sans nouvelle propagation immédiate. L'algorithme résultant reste correct mais élague évidemment moins efficacement l'espace de recherche ; de plus, la preuve de terminaison de l'algorithme repose sur la propagation systématique de la clause de conflit (voir section 7.4). Ces raisons nous ont incités à conserver la restriction du choix du niveau d'assertion à Γ.

7.1.4 Cas du pseudo-niveau de décision 0

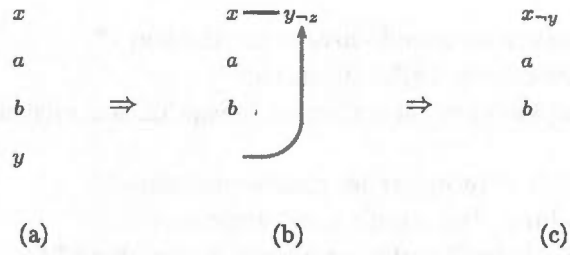
Le pseudo-niveau de décision 0, qui contient les propagations n'impliquant aucune décision, doit être géré de façon particulière. En effet, considérons un conflit où le niveau d'assertion est ce pseudo-niveau 0. Si nous appliquons notre algorithme général de saut arrière partiel, le conflit est résolu en défaisant uniquement les niveaux de décision dont une propagation dépend directement ou récursivement d'une instanciation du niveau 0, plus le niveau de conflit. Supposons qu'au moins un niveau de décision $i \neq 0$ est conservé. Alors, pendant que le niveau 0 est le niveau de décision courant, on peut obtenir dans ce pseudo-niveau de décision des propagations qui dépendent d'une décision, par exemple de $\delta(i)$. Or, par définition, le niveau 0 contient l'ensemble des propagations ne dépendant d'aucune décision. Un tel comportement de l'algorithme est donc absurde.

Par conséquent, tous les niveaux de décision sont considérés comme dépendants du pseudo-niveau 0, même si cette dépendance n'est motivée par aucune propagation. Ceci implique que, comme dans un CDCL conventionnel, toutes les décisions et leurs conséquences sont détruites lors d'un saut arrière vers le niveau 0 (ainsi que lors d'un redémarrage, qui est implémenté comme un saut arrière vers le niveau 0).

7.1.5 Exemple d'exécution

Nous allons illustrer le fonctionnement du CDCL à ordre partiel sans saut arrière sur la même instance utilisée dans les sous-sections 2.4.4, 2.5.9 et 6.1. Les principales étapes de cette exécution sont représentées par la figure 7.1. Nous supposerons à nouveau que la recherche commence en instanciant successivement les littéraux x, a, b et y. Les niveaux correspondant à ces décisions ne sont initialement pas dépendants entre eux, comme l'indique la figure 7.1a. Si les trois premières décisions ne provoquent aucune propagation unitaire, l'instanciation de y rend les clauses c_1 et c_2 unitaires. Supposons comme précédemment que c_1 est détectée la première ; le littéral $\neg z$ est alors instancié par propagation au niveau de décision courant, soit le niveau de y. Comme l'instanciation x a été impliquée dans cette propagation, le niveau de y devient dépendant du niveau de x,

$$c_1 = \neg x \vee \neg y \vee \neg z \qquad c_2 = \neg x \vee \neg y \vee z$$



(a) (b) (c)

FIGURE 7.1: Exemple d'exécution du CDCL à ordre partiel. La figure 7.1a décrit l'arbre de recherche produit par CDCL-OP sur la formule $\mathcal{F}(\{a, b, x, y, z\}, \{c_1, c_2\})$ après les décisions consécutives x, a, b et y. La figure 7.1b montre la modification produite par la propagation de $\neg z$ par la clause unitaire c_1. La figure 7.1c représente l'arbre de recherche après la résolution du conflit provoqué par la clause fausse c_2.

ce qui est illustré par la figure 7.1b. La clause c_2 est désormais fausse. Le niveau de conflit est le niveau de décision courant, donc le niveau de y. La clause de conflit est $\gamma = \neg x \vee \neg y$. Le niveau de x est l'unique niveau de décision présent dans γ hormis le niveau de conflit, c'est donc l'unique choix de niveau d'assertion. Le saut arrière partiel défait uniquement le niveau de conflit, puisqu'il s'agit du seul niveau dépendant du niveau d'assertion, puis propage l'assertion $\neg y$ à ce niveau d'assertion ; l'état de la recherche après résolution du conflit est représenté par la figure 7.1c. Si nous supposons que la formule contient une clause additionnelle $c_4 = y \vee \neg a \vee d$, le littéral d sera propagé au niveau de décision de x, qui deviendra dépendant du niveau de a, alors que $\tau(x) < \tau(a)$.

7.2 Description algorithmique

Le pseudo-code global du CDCL à ordre partiel, dont nous abrégerons le nom par CDCL-OP, est décrit par l'algorithme 7.1. Ce pseudo-code est strictement identique à celui du CDCL classique (algorithme 2.8), à l'exception de l'initialisation de la relation Δ (ligne 4) et l'ajout d'une dépendance au pseudo-niveau 0 pour chaque nouveau niveau (ligne 25). Les procédures PROPAGER, DÉFAIRE et PROPAGERASSERTION du CDCL à ordre partiel restent identiques au CDCL classique (algorithmes 2.15, 2.14 et 2.12 respectivement). INSTANCIER (algorithme 7.2) effectue les même tâches que dans un

Algorithme 7.1 CDCL-OP

1: $\Lambda \leftarrow \{0\}$
2: $\lambda_c \leftarrow 0$ /*on commence au pseudo-niveau de décision 0*/
3: $\sigma \leftarrow \emptyset$ /*on commence avec l'affectation vide*/
4: $\Delta \leftarrow \emptyset$ /*aucune dépendance entre niveaux puisqu'un seul niveau initial*/
5: **boucle**
6: $c \leftarrow$ PROPAGER() /*propager les clauses unitaires*/
7: **si** $c \neq$ NUL **alors** /*un conflit a été découvert*/
8: **si** $\lambda_c = 0$ **alors** /*conflit au niveau de décision 0*/
9: **retourner** faux /*$\mathcal{F}$ est insatisfaisable*/
10: **sinon**
11: $\gamma \leftarrow$ ANALYSER(c) /*déduire la clause de conflit γ*/
12: $a \leftarrow l \in \gamma \,|\, \lambda(a) = \lambda_c$
13: /*a est unique ; c'est la future assertion*/
14: $\lambda_a \leftarrow$ NIVEAUASSERTION(γ)
15: /*choix du niveau d'assertion*/
16: SAUTARRIÈRE(λ_a) /*retour au niveau d'assertion*/
17: $\lambda_c \leftarrow \lambda_a$ /*λ_a devient le niveau courant*/
18: $\mathcal{C} \leftarrow \mathcal{C} \cup \{\gamma\}$ /*γ est apprise*/
19: PROPAGERASSERTION(a, γ)
20: **sinon** /*aucun conflit pendant les propagations*/
21: **si** $\mathcal{D}(\sigma) = \mathcal{V}$ **alors** /*toutes les variables sont instanciées*/
22: **retourner** σ /*σ est un modèle de $\mathcal{F}$*/
23: **sinon**
24: $\lambda_c \leftarrow$ NOUVEAUNIVEAU()
25: $\Delta \leftarrow \Delta \cup \{(0, \lambda_c)\}$
26: $\Lambda \leftarrow \Lambda \cup \{\lambda_c\}$
27: **choisir** $v \in \mathcal{V} \backslash \mathcal{D}(\sigma)$ /*v est une variable non-instanciée*/
28: **choisir** $l \in \{v, \neg v\}$
29: INSTANCIER(l, NUL)
30: /*l est une décision, il n'a pas d'antécédent*/

Algorithme 7.2 INSTANCIER(l, c) *[CDCL-OP]*

Requis : $\nu(l) \notin \mathcal{D}(\sigma)$
1: $v \leftarrow \nu(l)$
2: $\sigma(v) \leftarrow \rho(l)$
3: $\lambda(v) \leftarrow \lambda_c$
4: $\alpha(v) \leftarrow c$
5: **si** $c \neq$ NUL **alors** /*l est une propagation*/
6: /*λ_c dépend de tous les autres niveaux dans c*/
7: **pour** $l' \in c \setminus \{l\} \,|\, \lambda(l') \neq \lambda_c$ **faire**
8: $\Delta \leftarrow \Delta \cup \{(\lambda(l'), \lambda_c)\}$

Algorithme 7.3 NIVEAUASSERTION(γ) *[CDCL-OP]*

1: $\Theta \leftarrow \{\lambda(l) \,|\, l \in \gamma\} \setminus \{\lambda_c\}$
2: est l'ensemble des niveaux impliqués dans le conflit, hormis λ_c*/
3: **si** $\Theta = \emptyset$ **alors** /*$|\gamma| = 1$*/
4: **retourner** 0
5: **sinon**
6: $\Gamma \leftarrow \{i \in \Theta, \nexists j \in \Theta \,|\, i \prec_\Delta j\}$
7: /* Γ est l'ensemble des éléments maximaux selon Δ^+ dans Θ*/
8: **choisir** $\lambda_a \in \Gamma$
9: **retourner** λ_a

Algorithme 7.4 SAUTARRIÈRE(λ_a) *[CDCL-OP]*

1: /*défaire le niveau courant et tous les niveaux qui dépendent de λ_a*/
2: **pour** $i \in \Lambda \,|\, a \prec_\Delta i$ **ou** $i = \lambda_c$ **faire**
3: **pour** $v \in \mathcal{V} \,|\, \lambda(v) = i$ **faire**
4: DÉFAIRE(v)
5:

CDCL classique (algorithme 2.13), auxquelles sont ajoutées la gestion des dépendances dans le cas d'une propagation (lignes 5 à 8). NIVEAUASSERTION (algorithme 7.3) permet de choisir le niveau d'assertion parmi un ou plusieurs niveaux candidats, ainsi qu'expliqué dans la sous-section 7.1.3. Le SAUTARRIÈRE (algorithme 7.4) défait le niveau courant λ_c, où a eu lieu le conflit, ainsi que tous les niveaux qui dépendent du niveau d'assertion.

On peut bien entendu utiliser le mécanisme des littéraux bloqués dans le CDCL à ordre partiel. Nous appellerons cette variante CDCL-OP$_{\text{LB}}$. Les seules parties de l'algorithme à modifier sont la description globale (algorithme 7.5) ainsi que la procédure PROPAGER, dont la description est alors identique à celle du CDCL ordinaire à littéraux bloqués (algorithme 2.16). La seule modification du pseudo-code global par rapport à celui de CDCL-OP consiste à vérifier toutes les clauses lorsqu'une instanciation complète est obtenue, car comme pour le CDCL sans saut arrière, les littéraux bloqués brisent la surveillance partielle dans le CDCL à ordre partiel et rendent donc l'algorithme incorrect sans vérification supplémentaire (voir sous-section 7.4.3). La vérification des clauses est implémentée de la même façon que dans CDCL-SSA$_{\text{LB}}$ (voir algorithme 6.9).

D'un point de vue plus implémentatif, la trace des instanciations est gérée comme

Algorithme 7.5 CDCL-OP$_{\text{LB}}$

```
1: Λ ← {0}
2: λc ← 0 /*on commence au pseudo-niveau de décision 0*/
3: σ ← ∅ /*on commence avec l'affectation vide*/
4: Δ ← ∅ /*aucune dépendance entre niveaux puisqu'un seul niveau initial*/
5: boucle
6:     si c = NUL alors
7:         /*c ≠ NUL si VérifierClauses a détecté un conflit*/
8:         /*à l'itération précédente*/
9:         c ← Propager() /*propager les clauses unitaires*/
10:    si c ≠ NUL alors /*un conflit a été découvert*/
11:        si λc = 0 alors /*conflit au niveau de décision 0*/
12:            retourner faux /*F est insatisfaisable*/
13:        sinon
14:            γ ← Analyser(c) /*déduire la clause de conflit γ*/
15:            a ← l ∈ γ | λ(a) = λc
16:            /*a est unique ; c'est la future assertion*/
17:            λa ← NiveauAssertion(γ)
18:            /*choix du niveau d'assertion*/
19:            SautArrière(λa) /*retour au niveau d'assertion*/
20:            λc ← λa /*λa devient le niveau courant*/
21:            C ← C ∪ {γ} /*γ est apprise*/
22:            PropagerAssertion(a, γ)
23:            c ← NUL /*conflit résolu*/
24:    sinon /*aucun conflit pendant les propagations*/
25:        si D(σ) = V alors /*toutes les variables sont instanciées*/
26:            c ← VérifierClauses()
27:            /*vérification supplémentaire de toutes les clauses*/
28:            si c = NUL alors /*aucune clause insatisfaite*/
29:                retourner σ /*σ est un modèle de F*/
30:        sinon
31:            λc ← NouveauNiveau()
32:            Δ ← Δ ∪ {(0, λc)}
33:            Λ ← Λ ∪ {λc}
34:            choisir v ∈ V\D(σ) /*v est une variable non-instanciée*/
35:            choisir l ∈ {v, ¬v}
36:            Instancier(l, NUL)
37:            /*l est une décision, il n'a pas d'antécédent*/
```

dans le CDCL sans saut arrière par un vecteur à deux dimensions, puisque le CDCL à ordre partiel permet d'une part de rajouter des instanciations à un niveau de décision qui n'est pas le dernier niveau chronologiquement créé, et d'autre part de détruire un niveau de décision tout en conservant un autre niveau chronologiquement plus récent. La gestion de la trace par un vecteur unidimensionnel serait donc inefficace.

Cette gestion de la trace est toutefois simplifiée par rapport au CDCL sans saut arrière, puisqu'aucun niveau de décision ne peut être défait seulement partiellement. Chaque élément d'un vecteur-niveau de décision représente donc toujours une instanciation valide, sans avoir à compacter le niveau lors d'une désinstanciation partielle. De même, lorsqu'un niveau de décision est désinstancié alors qu'un autre niveau plus récent est conservé, son emplacement dans le vecteur de niveaux de décision peut être réutilisé, puisque les relations entre niveaux de décision sont maintenues par la relation Δ. Dans le cas du CDCL sans saut arrière, l'ordre total entre les niveaux de décision correspond à leur ordre dans ce vecteur, et la position d'un niveau défait ne peut donc être réutilisée à moins de compacter le vecteur.

Enfin, dans le CDCL à ordre partiel, la gestion de la file des instanciations pas encore propagée est aussi simple que dans le CDCL ordinaire, grâce à l'unicité du niveau de propagation. En effet, toutes ces instanciations se situent toujours au niveau courant, elles sont donc toutes défaites en cas de saut arrière partiel. Il suffit donc de maintenir un pointeur sur le premier littéral non-propagé dans le niveau courant. En comparaison, le CDCL sans saut arrière nécessite de maintenir une file différente pour chaque niveau de décision.

7.3 Étude comparée du CDCL à ordre partiel

Cette section compare conceptuellement le CDCL à ordre partiel avec les autres méthodes évoquées dans cette thèse qui permettent de réduire la destructivité des sauts arrière. La sous-section 7.3.1 est consacrée à une comparaison avec notre contribution précédente, le CDCL sans saut arrière, tandis que la sous-section 7.3.2 compare le CDCL

à ordre partiel avec les différentes stratégies de l'état de l'art mentionnées dans le chapitre 3.

7.3.1 Comparaison avec le CDCL sans saut arrière

Nous pouvons noter deux différences fondamentales entre le CDCL à ordre partiel et le CDCL sans saut arrière. La première est d'ordre conceptuel : le CDCL à ordre partiel reste plus proche du CDCL classique que le CDCL sans saut arrière, puisqu'il ne supprime pas totalement l'étape de saut arrière mais cherche uniquement à réduire la quantité de désinstanciations lors de cette étape. À l'échelle d'un conflit, il est donc évident que le CDCL à ordre partiel évitera moins de désinstanciations que le CDCL sans saut arrière. Ce compromis a toutefois son intérêt en pratique, puisqu'il évite de nombreuses complexités supplémentaires apportées par la suppression totale du saut arrière : propagations et conflits à des niveaux de décision quelconques, occurrence de conflits triviaux et redondants qui en découle, détection des propagations dépendant du niveau de conflit pour pouvoir résoudre le conflit, gestion coûteuse des propagations alternatives pour rétablir l'exhaustivité des propagations unitaires. [1]

La seconde différence tient à la granularité des dépendances considérées. Le CDCL à ordre partiel maintient un ordre entre les différents niveaux de décision au cours d'une recherche, alors que le CDCL sans saut arrière considère des dépendances plus détaillées entre les instanciations elles-mêmes.

La réduction de la finesse des dépendances implique elle aussi une réduction de la quantité d'instanciations sauvables par le CDCL à ordre partiel. En effet, il suffit qu'une seule instanciation d'un niveau de décision ait été propagée à partir du niveau d'assertion pour que le niveau entier soit défait. Il aurait été possible de concevoir cet algorithme avec un ordre partiel construit entre les instanciations elles-mêmes, comme pour le CDCL

---

1. Le CDCL à ordre partiel nécessite également un mécanisme supplémentaire pour assurer l'exhaustivité des propagations unitaires, mais nous constaterons dans la section 7.7 qu'il peut être significativement plus léger que les propagations alternatives.

sans saut arrière. Cependant, cela aurait induit la même lourdeur de gestion, et même une lourdeur plus importante, puisque le CDCL à ordre partiel nécessite de trouver toutes les dépendances du niveau d'assertion, tandis qu'il s'agit des dépendances du niveau de conflit dans le cas du CDCL sans saut arrière. Comme le niveau d'assertion est (sous un ordre total des niveaux de décision) strictement inférieur au niveau de conflit, la découverte de ses dépendances peut être considérablement plus longue, que les dépendances soient représentées explicitement par un arbre ou retrouvées par un parcours de la trace des instanciations.

Par ces deux aspects, le CDCL à ordre partiel permet de sauver moins d'instanciations que le CDCL sans saut arrière, mais il permet en contrepartie un traitement plus rapide.

7.3.2 Comparaison avec les autres méthodes de réduction de destructivité des sauts arrière

Si le CDCL sans saut arrière montre des similarités importantes avec l'algorithme DBT pour le problème CSP, le CDCL à ordre partiel est lui plus proche de POB. En effet, les deux algorithmes ont en commun l'assouplissement de l'ordre total respectivement sur les niveaux de décision et les variables qui leur permet une liberté de choix sur un élément qui est défini de façon unique dans une recherche en profondeur à saut arrière conventionnelle (respectivement le niveau d'assertion et la variable coupable).

Ils diffèrent cependant sur les contraintes qu'ils imposent aux différents choix de l'algorithme pour en garantir la terminaison. Le CDCL à ordre partiel restreint en tout temps le choix du niveau d'assertion à un niveau maximal parmi les niveaux impliqués dans la clause de conflit, hormis le niveau de conflit lui-même ; cependant, il n'impose absolument aucune restriction à l'heuristique de choix des décisions. Au contraire, POB permet initialement une liberté totale dans le choix de la variable coupable parmi les variables impliquées dans le *nogood* résultant du conflit. Cependant, il pose progressivement des contraintes globales de précédence entre variables qui restreignent à la fois le

choix de la variable coupable et celui de la prochaine variable à instancier.

Les algorithmes diffèrent également dans la granularité des contraintes qu'ils considèrent : CDCL-OP raisonne par niveaux de décision entiers et peut donc défaire plusieurs décisions en un seul saut arrière partiel. La granularité de POB est plutôt similaire à celle de CDCL-SSA : chaque résolution de conflit défait uniquement la variable coupable et ses conséquences sur les éliminations de variables.

CDCL-OP a également une similarité conceptuelle avec DBT, car leurs deux stratégies reviennent plus ou moins à déplacer certaines décisions (respectivement le niveau d'assertion et la variable coupable) par rapport à l'ordre chronologique. Cependant, DBT ne permet pas de liberté dans le choix de la variable coupable. La différence de granularité dans la gestion des dépendances est identique à celle constatée précédemment par rapport à POB.

Comme CDCL-SSA, CDCL-OP permet une résolution indépendante des composantes connexes d'un sous-problème résiduel sans avoir à les détecter : en effet, une composante ne peut pas provoquer de propagation unitaire dans une autre composante, donc aucune dépendance ne peut être créée entre elles et un saut arrière partiel dans une composante ne peut défaire aucun instanciation dans une autre. CDCL-OP tient compte des dépendances effectivement utilisées entre niveaux de décision lors de la résolution d'un conflit, alors que les méthodes de détection de la connectivité des sous-problèmes résiduels tient compte des dépendances possibles entre les variables non-instanciées avant une décision.

Les avantages et inconvénients de CDCL-OP par rapport aux décompositions implicites sont également similaires à la comparaison de ces derniers avec CDCL-SSA : le CDCL à ordre partiel évite le coût de construction d'une décomposition de l'instance et tire parti de la connectivité exacte des sous-problèmes résiduels, mais ne peut provoquer la séparation de l'instance et donc ne modifie pas la complexité théorique du CDCL.

Notons enfin que le CDCL à ordre partiel a aussi une certaine convergence avec les

méthodes à sauts arrière illimités. En effet, si celles-ci peuvent augmenter la destructivité des sauts arrière du CDCL classique, ce qui est à l'opposé du but du CDCL à ordre partiel, leur objectif principal est de pouvoir défaire une instanciation antérieure au niveau d'assertion conventionnel, afin d'accroître la mobilité de la recherche dans l'espace de recherche. Le CDCL à saut arrière partiel permet lui aussi de choisir le niveau d'assertion sans respecter l'ordre chronologique de la création des niveaux, et nous verrons dans la sous-section 7.9.2 que dans certains cas il semble aussi améliorer les performances de résolution en permettant un déplacement plus rapide dans l'espace de recherche. Le CDCL à ordre partiel a en outre l'avantage de conserver la complétude de l'algorithme dans tous les cas ; les méthodes à saut arrière illimité, quant à elles, sont obligées de limiter de façon importante le nombre de sauts arrière illimités pour conserver la complétude de la recherche.

7.4 Propriétés du CDCL à ordre partiel

Cette section s'intéresse aux propriétés théoriques du CDCL à ordre partiel, avec ou sans gestion des littéraux bloqués. Nous allons montrer que ces propriétés sont identiques à celles du CDCL sans saut arrière respectivement avec ou sans littéraux bloqués :

- le CDCL à ordre partiel est complet, termine et a une complexité temporelle exponentielle au pire des cas, avec ou sans littéraux bloqués ;
- sans littéraux bloqués, il est à conflits exhaustifs, ce qui garantit la correction de l'algorithme, mais pas à propagations exhaustives, ce qui permet l'occurrence de conflits triviaux et redondants ;
- avec littéraux bloqués, ni les conflits ni les propagations ne sont exhaustifs, ce qui nécessite donc des vérifications supplémentaires des clauses pour maintenir la correction de l'algorithme, sans éviter les conflits triviaux et redondants.

La sous-section 7.4.1 démontre les propriétés de complétude, terminaison et complexité temporelle communes aux deux variantes, tandis que les sous-sections 7.4.2 et 7.4.3 couvrent les propriétés propres au CDCL à ordre partiel respectivement sans ou avec littéraux bloqués.

7.4.1 Complétude, terminaison et complexité temporelle

Proposition 7.1. *CDCL-OP et CDCL-OP$_{LB}$ sont complets.*

Démonstration. La preuve de complétude du CDCL classique (proposition 2.6) reste valide dans le cas du CDCL à ordre partiel. □

Proposition 7.2. *CDCL-OP et CDCL-OP$_{LB}$ terminent.*

Démonstration. La terminaison du CDCL à ordre partiel peut être prouvée en adaptant la preuve de terminaison du CDCL ordinaire (proposition 2.7).

Pour tout $i \in \mathbb{N}$, soient Λ_i et Δ_i respectivement l'ensemble des niveaux de décision non-vides et leurs relations après la $i^{\text{ième}}$ instanciation dans la recherche CDCL à ordre partiel (« au temps i »). Si l'exécution se termine après $n \in \mathbb{N}$ instanciations, nous supposerons que $\forall i > n$, Λ_i et Δ_i représentent l'état des niveaux de décision et des relations à la fin de l'exécution. Soient $\Lambda_\infty = \bigcup_{i\in\mathbb{N}} \Lambda_i$ et $\Delta_\infty = \bigcup_{i\in\mathbb{N}} \Delta_i$ les ensembles, possiblement infinis si l'exécution ne termine pas, des niveaux de décision et des relations au cours de l'exécution. Δ_∞ est toujours une relation irréflexive. De plus, supposons que, lorsqu'un niveau de décision est désinstancié, son « symbole » n'est jamais réutilisé par un niveau de décision ultérieur. Alors, Δ_∞ est également asymétrique. Par conséquent, Δ_∞^+ est un ordre partiel sur Λ_∞ et $\forall i \in \mathbb{N}$, $\Delta_\infty \cap (\Lambda_i \times \Lambda_i)$ est un ordre partiel sur Λ_i ; de plus, pour tout ordre total Ψ sur Λ_∞ qui étend Δ_∞^+, la restriction de Ψ à $\Lambda_i \times \Lambda_i$ est également un ordre total sur Λ_i qui étend Δ_i^+ pour tout $i \in \mathbb{N}$. Ψ est donc un ordre total sur l'ensemble des niveaux de décision utilisés au cours de la recherche qui est compatible localement avec l'ordre partiel à chaque étape.

$\forall i \in \mathbb{N}$, $\forall j \in \Lambda_\infty$, notons $\kappa(i)$ et $\upsilon_i(j)$ respectivement le nombre de niveaux de décision non-vide au temps i et le nombre de variables instanciées au niveau j au temps i (ou à la fin de la recherche si celle-ci termine après moins de i instanciations). La fonction ρ_i définie ci-dessous numérote de 1 à $\kappa(i)$ les $\kappa(i)$ niveaux de décision non-vides

au temps i selon l'ordre total Ψ et renvoie une valeur nulle pour tous les autres niveaux de décision :

$$\rho_i(j) = \begin{cases} 0 & \text{si } j = 0 \text{ ou } v_i(j) = 0 \\ |\{k \in \Lambda_\infty \setminus \{0\} \mid k <_\Psi j \text{ et } v_i(j) \neq 0\}| + 1 & \text{sinon} \end{cases}.$$

Enfin, la fonction $f : \mathbb{N} \to \mathbb{R}$ suivante associe un poids à chaque instant de l'exécution en fonction de la répartition par niveaux des instanciations :

$$f(i) = \sum_{j \in \Lambda_\infty} \frac{v_i(j)}{|\mathcal{V}|^{\rho_i(j)+1}} .$$

Comme la fonction définie dans la preuve de la proposition 2.7, f est construite de telle façon que toute instanciation au $j^{\text{ième}}$ niveau de décision non-vide a plus de poids que la totalité des variables situées à des niveaux de décision plus élevés. On peut donc de la même façon prouver que f est une fonction strictement croissante jusqu'à la fin de la recherche. En effet, lors d'un saut arrière partiel, le poids des variables désinstanciées est compensé par celui de l'assertion, car tous les niveaux désinstanciés dépendent du niveau d'assertion et y sont donc strictement supérieurs dans l'ordre total Ψ.[2] Notons que dans le cas du CDCL à ordre partiel, le poids d'un niveau de décision non-vide j_1 peut décroître si une décision est prise dans un nouveau niveau j_2 qui lui est inférieur selon Ψ ; cependant, le poids de la décision prise dans j_2 compense également la perte de poids du niveau j_1.

La fonction f croît donc strictement tant que l'exécution continue et ne peut prendre qu'un ensemble fini de valeurs ; cela prouve la terminaison du CDCL à ordre partiel. □

Proposition 7.3. *CDCL-OP et CDCL-OP$_{LB}$ ont une complexité temporelle exponen-*

2. Nous supposons ici que lors de tout conflit le niveau de conflit est rendu dépendant du niveau d'assertion.

tielle au pire des cas.

Démonstration. La preuve de complexité temporelle du CDCL classique (proposition 2.8) reste valide dans le cas du CDCL à ordre partiel. □

7.4.2 Propriétés du CDCL à ordre partiel sans littéraux bloqués

Nous allons montrer ici que CDCL-OP est à surveillance partielle, ce qui implique l'exhaustivité des conflits et la correction de l'algorithme, mais qu'il n'est pas à propagations exhaustives, ce qui a pour conséquence l'occurrence de conflits triviaux et redondants.

Proposition 7.4. *CDCL-OP est à surveillance partielle.*

Démonstration. Au début de l'exécution, aucune variable n'est instanciée, donc toutes les clauses sont partiellement surveillées. Seules les instanciations de littéraux peuvent menacer la surveillance partielle des clauses, car les sauts arrière partiels ne font que désinstancier des littéraux. Lorsqu'un littéral l est instancié, il ne peut modifier la surveillance partielle que pour les clauses $c \in \mathcal{C}$ où $\neg l$ est un littéral surveillé. Lorsque la clause c est vérifiée, $\neg l$ n'est pas remplacé si le second littéral surveillé w est vrai, ce qui suffit à maintenir la surveillance partielle de c. Si w est indéfini ou faux, la procédure de propagation unitaire cherche à remplacer $\neg l$ par un littéral vrai ou indéfini dans $c \setminus \{\neg l, w\}$. Si un tel littéral existe, il est surveillé à la place de $\neg l$ et rétablit la surveillance partielle de c. Si tous les littéraux de $c \setminus \{\neg l, w\}$ sont faux et w est indéfini, celui-ci est instancié par propagation unitaire et c est donc partiellement surveillée. Si w est faux, c est entièrement fausse, ce qui déclenche un conflit. L'instanciation l est défaite par le saut arrière partiel car elle appartient au niveau de conflit. $\neg l$ devient donc indéfini et c est partiellement surveillée. □

Corollaire 7.1. *CDCL-OP est à conflits exhaustifs et totalement correct.*

Démonstration. CDCL-OP est à conflits exhaustifs d'après les propositions 7.4 et 5.2, ce qui implique sa correction d'après la proposition 5.1. Comme il est également complet et termine, d'après les propositions 7.1 et 7.2 respectivement, il est donc totalement correct. □

Proposition 7.5. *CDCL-OP n'est pas à surveillance entière ni à propagations exhaustives.*

Démonstration. Considérons une formule propositionnelle à 6 variables a, b, c, d, e et f et à 4 clauses $c_1 = \neg a \vee \neg b \vee \neg c$, $c_2 = \neg a \vee b \vee \neg d$, $c_3 = \neg e \vee f$ et $c_4 = \neg a \vee \neg e \vee \neg f$. Supposons que chaque clause est initialement surveillée par ses deux premiers littéraux. Nous allons montrer qu'il existe une exécution de CDCL-OP sur cette formule où une clause n'est pas entièrement surveillée en tout temps et où une procédure de propagation unitaire termine en ayant omis une clause unitaire.

- Le littéral a est instancié comme décision de niveau 1. Les clauses c_1, c_2 et c_4 sont maintenant surveillées par $\{\neg b, \neg c\}$, $\{b, \neg d\}$ et $\{\neg e, \neg f\}$ respectivement.
- Le littéral b est instancié comme décision de niveau 2. La clause c_1 est détectée comme unitaire. Le littéral $\neg c$ est donc instancié comme propagation de c_1 au niveau 2. Comme $\neg a$ est un littéral antécédent de $\neg c$ de niveau 1, la dépendance $1\Delta 2$ est ajoutée à Δ. Aucune clause n'est surveillée par c.
- Le littéral d est instancié comme décision de niveau 3. La seule clause surveillée par $\neg d$ est c_2, dont le second littéral surveillé b est vrai ; la propagation de d n'a donc aucun effet.
- Le littéral e est instancié comme décision de niveau 4, ce qui rend les clauses c_3 et c_4 unitaires. Supposons que c_3 est détectée la première. f est alors instancié comme propagation unitaire de c_3 au niveau 4, ce qui rend c_4 fausse. La clause de conflit est $c_5 = \neg a \vee \neg e$, de niveau de conflit 4. L'unique niveau d'assertion possible est le niveau 1.
- Pour effectuer le saut arrière partiel, l'algorithme défait le niveau de conflit 4, ainsi que le niveau 2 qui dépend du niveau d'assertion. L'assertion $\neg e$ est

ensuite instanciée au niveau d'assertion 1 par propagation de c_5. La procédure de propagation unitaire consécutive termine sans vérifier aucune clause, puisque e n'est surveillé dans aucune clause.

La clause c_2 est maintenant surveillée par un littéral indéfini b et un littéral faux et déjà propagé $\neg d$. CDCL-OP n'est donc pas à surveillance entière. De plus, cette clause est unitaire et n'a pas été détectée par la procédure de propagation qui a suivi l'instanciation de e ; CDCL-OP n'est donc pas non plus à propagations exhaustives. □

Corollaire 7.2. *CDCL-OP n'est ni non-trivial, ni non-redondant.*

Démonstration. CDCL-OP n'est pas à propagations exhaustives d'après la proposition 7.5. Comme il est à niveau de propagation unique et utilise la stratégie d'apprentissage du premier point d'implication unique, la non-exhaustivité des propagations implique que l'algorithme n'est ni non-trivial, ni non-redondant, d'après la contraposée du corollaire 5.5. □

7.4.3 Propriétés du CDCL à ordre partiel avec littéraux bloqués

Nous allons montrer que CDCL-OP$_{\text{LB}}$ n'est ni à surveillance partielle, ni a conflits exhaustifs, ce qui justifie la présence de vérifications supplémentaires pour assurer la correction de l'algorithme, et qui, comme pour le CDCL à ordre partiel sans littéraux bloqués, empêche d'éviter les conflits triviaux et redondants.

Proposition 7.6. *CDCL-OP$_{LB}$ n'est pas à surveillance partielle ni à conflits exhaustifs.*

Démonstration. Cette propriété peut être prouvée en utilisant le contre-exemple de la preuve de la proposition 6.5, qui montre les mêmes propriétés dans le cas de l'algorithme CDCL-SSA$_{\text{LB}}$. En effet, si l'on conserve la même séquence de décisions et le même état initial des littéraux surveillés et bloqués, CDCL-OP$_{\text{LB}}$ s'exécute exactement de la même façon que CDCL-SSA$_{\text{LB}}$ sur cette formule. Cela est dû au fait que, lors de chaque conflit rencontré, les deux implémentations défont uniquement toutes les instanciations

du niveau de conflit car, d'une part, aucune propagation à un autre niveau ne dépend du niveau de conflit dans CDCL-SSA$_{LB}$ et, d'autre part, aucun niveau de décision autre que le niveau de conflit ne dépend du niveau d'assertion. □

Corollaire 7.3. *CDCL-OP$_{LB}$ n'est pas à propagations exhaustives, ni non-trivial, ni non-redondant.*

Démonstration. Comme l'algorithme n'est pas à conflits exhaustifs (proposition 7.6), il n'est pas non plus à propagations exhaustives (contraposée du corollaire 5.1). Comme il est à niveau de propagation unique et utilise la stratégie d'apprentissage du premier point d'implication unique, la non-exhaustivité des propagations implique que l'algorithme n'est ni non-trivial, ni non-redondant, d'après la contraposée du corollaire 5.5. □

Proposition 7.7. *CDCL-OP$_{LB}$ est totalement correct.*

Démonstration. Malgré la non-exhaustivité des conflits, l'algorithme est correct grâce à la vérification supplémentaire des clauses à la ligne 26 de l'algorithme 7.5. Comme il est également complet (proposition 7.1) et il termine (proposition 7.2), il est donc totalement correct. □

7.5 Implémentation de la relation Δ

L'algorithme CDCL-OP effectue fréquemment des opérations sur la relation Δ ; il est donc important de représenter cette relation d'une façon qui permette d'implémenter efficacement ces opérations, qui sont de trois types différents :

- lors d'une propagation unitaire, CDCL-OP ajoute une ou plusieurs dépendances précises entre deux niveaux donnés ;
- lors d'un saut arrière partiel, pour déterminer les niveaux à défaire, CDCL-OP doit pouvoir retrouver à partir de Δ tous les niveaux qui dépendent, directement ou non, d'un niveau donné ;

- lorsqu'un niveau est défait, il est nécessaire de retirer de Δ toutes les informations sur ses dépendances dans les deux directions.

Cette section discute donc différents aspects de l'implémentation de la relation Δ et différents types de structures de données utilisables. La sous-section7.5.1 justifie tout d'abord le stockage de la relation non-transitive Δ plutôt que de sa fermeture transitive Δ^+. Les sous-sections 7.5.2 à 7.5.4 décrivent différentes structures de données pour représenter Δ, respectivement des vecteurs non-ordonnés, des arbres rouge et noir et une matrice à deux dimensions, ainsi que leur impact sur l'efficacité théorique et pratique des différentes opérations de l'algorithme. Enfin, la sous-section 7.5.5 présente la solution d'implémentation que nous avons retenue, qui consiste en une utilisation combinée de vecteurs et d'une matrice.

7.5.1 Stockage non-transitif de Δ

Nous avons choisi de stocker uniquement Δ, c'est-à-dire les relations directes entre niveaux décision, plutôt que sa fermeture transitive Δ^+. La raison en est simplement que la maintenance de la fermeture transitive totale de Δ est très coûteuse en espace, mais surtout en temps, pour une utilité très limitée.

En effet, lorsqu'une nouvelle dépendance directe entre deux niveaux de décision $i\Delta j$ est ajoutée, la mise à jour de la fermeture transitive requiert de rajouter une dépendance entre tout élément plus petit ou égal à i et tout élément plus grand ou égal à j. Plus formellement, $(\Delta \cup \{(i,j)\})^+ = \Delta^+ \cup \{k \prec l \,|\, \forall k \preceq i, \forall l \succeq j, k \neq l\}$. Une telle mise à jour est potentiellement nécessaire pour chaque littéral antécédent de chaque propagation unitaire.

La connaissance intégrale de cette fermeture transitive a toutefois très peu d'utilité, puisque l'algorithme nécessite uniquement de connaître lors de chaque conflit la fermeture transitive de la relation restreinte au niveau d'assertion et aux éléments qui lui sont supérieurs dans Δ^+.

Or, les propagations unitaires sont bien plus fréquentes que les conflits, et la

partie de Δ^+ nécessaire lors des conflits est souvent très petite par rapport à la relation entière. Il est donc beaucoup plus efficace de mémoriser uniquement Δ et de construire à chaque conflit la partie de Δ^+ nécessaire par un parcours récursif de Δ à partir du niveau d'assertion. La différence d'efficacité a été constatée expérimentalement en implémentant les deux variantes et est clairement en faveur du stockage non-transitif de Δ. C'est donc cette méthode de stockage que nous considérerons lors des études expérimentales de ce chapitre.

7.5.2 Stockage par vecteurs

Le stockage de la relation Δ peut être implémenté simplement à l'aide de vecteurs non-ordonnés, ou plus exactement de vecteurs de vecteurs : pour tout niveau de décision i, $dep[i]$ est un vecteur contenant l'ensemble des niveaux j qui dépendent de i, c'est-à-dire tels que $i\Delta j$. Cette représentation utilise un espace en $O(|\Lambda|^2)$, puisqu'elle utilise $|\Lambda|$ listes contenant chacune au plus $|\Lambda| - 1$ entrées.

Lors d'une propagation unitaire au niveau j qui implique un littéral antécédent de niveau $i \neq j$, il faut tout d'abord vérifier si j appartient déjà à $dep[i]$, puis le rajouter si ce n'est pas le cas. L'insertion non-ordonnée se fait en temps constant, mais la recherche prend un temps linéaire selon la taille de $dep[i]$, bornée par $|\Lambda| - 1$. L'ajout d'une dépendance entre deux niveaux précis se fait donc en temps $O(|\Lambda|)$.

Retrouver la liste de tous les niveaux qui dépendent directement ou indirectement d'un niveau $i \in \Lambda$ s'effectue récursivement en parcourant d'abord $dep[i]$, puis la liste des dépendances des niveaux dans $dep[i]$, et ainsi de suite. On parcourt au pire des cas les listes de dépendances des $|\Lambda|$ niveaux, ce qui compte tenu de la taille maximale des listes implique une complexité de $O(|\Lambda|^2)$ pour trouver la liste des dépendances directes ou indirectes d'un niveau donné.

Pour un niveau de décision i, il est facile d'effacer toutes les dépendances de type $i\Delta j$: il suffit de vider la liste $dep[i]$. Au contraire, retrouver et effacer les dépendances de type $j\Delta i$ nécessite de parcourir les listes de tous les autres niveaux de décision, ce

qui donne une complexité de $O(|\Lambda|^2)$. Il est possible de réduire cette complexité en maintenant un second ensemble de vecteurs qui permet de parcourir les dépendances dans le sens inverse : $dep^{-1}[i]$ contient l'ensemble des niveaux de décision dont dépend i.

En pratique, chaque élément de $dep[i]$ est un couple $(j, pos) \in \Lambda \times \mathbb{N}$ où j est un niveau de décision qui dépend de i et pos est la position de i dans la liste $dep^{-1}[j]$; inversement, chaque élément de $dep^{-1}[i]$ est un couple $(j, pos) \in \Lambda \times \mathbb{N}$ où j est un niveau de décision dont dépend i et pos est la position de i dans la liste $dep[j]$. dep et dep^{-1} peuvent donc être considérés comme des fonctions de type $\Lambda \to \Lambda \times \mathbb{N}$. De cette façon, lors de la désinstanciation d'un niveau de décision i, il suffit d'effacer le $pos^{\text{ième}}$ élément de $dep^{-1}[j]$ pour tout $(j, pos) \in dep[i]$, d'effacer le $pos^{\text{ième}}$ élément de $dep[j]$ pour tout $(j, pos) \in dep^{-1}[i]$, puis d'effacer $dep[i]$ et $dep^{-1}[i]$. Mettre la représentation de Δ à jour lorsqu'un niveau est défait s'effectue donc en temps $|\Lambda|$.

Notons que la gestion des vecteurs inverses ne modifie pas la complexité des deux opérations précédentes : l'ajout si besoin d'une nouvelle dépendance précise se fait de la même façon que pour le vecteur de base, et la détection des dépendances récursives d'un niveau n'utilise pas du tout les vecteurs inverses.

7.5.3 Stockage par arbres rouge et noir

On peut facilement améliorer les bornes de complexité obtenues par la solution précédente en implémentant les listes à l'aide de structures de données plus complexes, par exemple des arbres rouge et noir. En effet, l'insertion, la recherche ou la suppression d'un élément dans un arbre rouge et noir à n éléments se fait en temps $O(\log(n))$. Si pour tout niveau $i \in \Lambda$ les ensembles $dep[i]$ et $dep^{-1}[i]$ sont implémentés par des arbres rouge et noir, on obtient alors une complexité de $O(\log(|\Lambda|))$ pour l'ajout d'une dépendance précise ou pour la suppression d'un niveau de décision et de $O(|\Lambda| \times \log(|\Lambda|))$ pour le calcul des dépendances récursives d'un niveau. Malgré ces meilleures bornes de complexité, notre implémentation de cette stratégie à l'aide des ensembles ordonnés de

la librairie standard C++, qui eux-mêmes implémentent les arbres rouge et noir, est en pratique bien moins efficace qu'une implémentation utilisant des vecteurs non-ordonnés.

7.5.4 Stockage par matrice

Il est également possible de représenter la relation Δ par une matrice booléenne à deux dimensions, $\mathcal{M} : \Lambda \times \Lambda \to \mathbb{B}$, telle que $\mathcal{M}(i,j) = \text{vrai} \Leftrightarrow i\Delta j$. Cette implémentation utilise un espace en $\Theta(|\Lambda|^2)$. Son avantage principal est que la vérification ou l'ajout d'une dépendance précise $i\Delta j$ s'effectue en temps constant, puisqu'il suffit de consulter ou de modifier $\mathcal{M}(i,j)$.

Connaître la liste des dépendances directes d'un niveau i requiert de parcourir entièrement la $i^{\text{ième}}$ ligne de $\mathcal{M}$, ce qui s'effectue en temps $\Theta(|\Lambda|)$. Pour trouver la liste des dépendances directes ou indirectes de i, il faut d'abord trouver les dépendances directes de i, puis récursivement les dépendances directes de ces dépendances directes, et ainsi de suite. Il faut donc parcourir $O(|\Lambda|)$ lignes, ce qui donne un temps d'exécution global borné à la fois par $O(|\Lambda|^2)$ et par $\Omega(|\Lambda|)$. En comparaison, la borne inférieure de cette opération avec un stockage par vecteurs est constante, soit $\Omega(1)$: cette borne correspond au cas où aucun niveau ne dépend de i.

Enfin, pour effacer toutes les dépendances qui impliquent un niveau de décision i, il faut parcourir entièrement la $i^{\text{ième}}$ ligne et la $i^{\text{ième}}$ colonne de la matrice, ce qui prend un temps $\Theta(|\Lambda|)$. Là encore, la borne supérieure est identique à celle obtenue par l'implémentation par vecteurs, mais la borne inférieure de celle-ci est constante.

Globalement, par rapport à l'implémentation par vecteurs, l'utilisation d'une matrice améliore la complexité pour la gestion d'une dépendance particulière, mais impose des bornes temporelles inférieures pour les deux autres types d'opérations nécessaires, tout en augmentant le besoin d'espace mémoire.

7.5.5 Stockage par vecteurs et matrice

La solution que nous avons retenue pour l'implémentation de CDCL-OP combine l'utilisation de vecteurs non-ordonnés et de matrices pour exploiter leurs avantages respectifs pour les différentes opérations nécessaires.

Notre implémentation utilise donc deux ensembles de vecteurs $dep : \Lambda \to \Lambda \times \mathbb{N}$ et $dep^{-1} : \Lambda \to \Lambda \times \mathbb{N}$ définis de la même façon que dans la sous-section 7.5.2 et une matrice booléenne à deux dimensions $\mathcal{M} : \Lambda \times \Lambda \to \mathbb{B}$.

L'existence d'une dépendance particulière (i, j) peut être vérifiée en temps constant en consultant $\mathcal{M}(i, j)$. Si elle n'existe pas et doit être ajoutée, $\mathcal{M}(i, j)$ est assigné à vrai, j est ajouté à $dep[i]$ et i est ajouté à $dep^{-1}[j]$.

Le calcul des dépendances récursives d'un niveau de décision est effectué uniquement à l'aide des vecteurs et prend donc un temps $O(|\Lambda|^2)$ et $\Omega(1)$.

La suppression des dépendances d'un niveau est également effectué de la même façon que pour les vecteurs seuls, à la différence que les dépendances parcourues sont au passage retirées de la matrice. La suppression prend donc un temps $O(|\Lambda|)$ et $\Omega(1)$.

Cette structure de donnée combinée cumule donc le temps constant de vérification ou modification d'une dépendance par la matrice à la borne inférieure constante pour les deux autres opérations lorsqu'elles sont effectuées par les vecteurs. L'espace mémoire requis, dû principalement à la matrice, est en pratique le plus souvent raisonnable ; cependant, il a parfois dépassé les capacités de la mémoire vive de notre machine de test, sur certaines instances où le nombre maximal de niveaux de décision simultanés est particulièrement élevé. Dans ces cas, l'exécution n'a pas été interrompue mais fortement ralentie par l'utilisation de la mémoire cache. Pour éviter cet inconvénient, notre implémentation de CDCL-OP utilise au départ la structure combinée pour gérer la relation Δ, mais la matrice est détruite en cours d'exécution si le nombre de niveaux de décision simultanés dépasse un certain seuil prédéfini. Dans ce cas, l'exécution est terminée en utilisant seulement les vecteurs.

7.6 Évaluation expérimentale du CDCL à ordre partiel

Pour évaluer expérimentalement l'efficacité du CDCL à ordre partiel, nous l'avons implémenté, comme le CDCL sans saut arrière, en modifiant le solveur GLUCOSE 1.0 (Audemard et Simon, 2009). Dans toutes nos variantes de CDCL à ordre partiel, nous avons désactivé la sauvegarde de phase, pour les mêmes raisons que dans le cas du CDCL sans saut arrière (le CDCL à ordre partiel est conçu comme une alternative à la sauvegarde de phase, et des tests préliminaires confirment que conserver la sauvegarde de phase détériore les performances du CDCL à ordre partiel).

Nous avons utilisé par défaut une heuristique de choix du niveau d'assertion qui se rapproche le plus possible du comportement du CDCL classique, dans le but d'évaluer l'impact des sauts arrière partiels uniquement, sans réelle utilisation de cette liberté de choix supplémentaire. Lorsque l'ensemble Θ des niveaux d'assertion candidats comporte plus d'une possibilité, nous choisissons le niveau le plus récemment créé parmi ces candidats. Nous appellerons cette heuristique de choix de niveau d'assertion l'heuristique chronologique. D'autres heuristiques seront présentées et évaluées dans la section 7.9.

Les tableaux 7.1 à 7.7 présentent une comparaison de l'exécution des implémentations de CDCL-OP et CDCL-OP$_{\text{LB}}$ avec l'implémentation originale de GLUCOSE, respectivement avec (CDCL$_{\text{LB}}$) et sans (CDCL) littéraux bloqués, sur la même sélection d'instances utilisée dans le chapitre 6. Les tableaux présentent le même type d'informations que les tableaux 6.1 à 6.7 respectivement, et nous renvoyons à la sous-section 6.6.1 pour leur description détaillée. La seule différence est que, dans les tableaux 7.1, 7.6 et 7.7, la destructivité moyenne des sauts arrière est calculée sur l'ensemble des conflits et des redémarrages de chaque exécution, tandis que dans les tableaux 6.1, 6.6 et 6.7 seuls les conflits ordinaires sont pris en compte. Toutefois, cette différence a peu d'impact sur les données, étant donné que les conflits sont bien plus fréquents que les redémarrages. On peut ainsi constater que la différence entre les destructivités moyennes des conflits des implémentations CDCL$_{\text{LB}}$ et CDCL présentées par les tableaux 7.1 et 7.7 d'une part et 6.1 et 6.7 d'autre part ne varient que de quelques centièmes de pourcentage.

TABLEAU 7.1: Comparaison expérimentale de différentes implémentations (*impl*) du CDCL classique et du CDCL à ordre partiel dans le solveur GLUCOSE. Les implémentations du CDCL classique sont l'implémentation originale de GLUCOSE (CDCL_{LB}) et une variante sans littéraux bloqués (CDCL). Les implémentations du CDCL à ordre partiel sont CDCL-OP (OP), CDCL-OP_{LB} (OP_{LB}), CDCL-OP-Surveillance (Surv), CDCL-OP-Dépendances (Dép) et CDCL-OP-$\text{Dépendances}_{\text{LB}}$ (Dép_{LB}). Chaque implémentation a été exécutée sur les 62 instances énumérées dans le tableau A.2 avec une limite d'une heure. Pour chaque implémentation, nous indiquons le nombre d'instances résolues à l'intérieur de ce délai (*rés*), le temps total d'exécution en secondes (t), le nombre total d'étapes de propagation en millions (*ep*), le nombre moyen d'étapes de propagation par secondes (*#ep/sec*), la destructivité moyenne des conflits (d/c), la proportion moyenne de conflits triviaux (*triv*) et l'évolution moyenne de la distance entre les conflits par rapport à l'implémentation originale de GLUCOSE (*dist*).

impl	*rés*	t	*ep*	*#ep/sec*	d/c	*triv*	*dist*
CDCL_{LB}	61	36 500	1 143 301	29 240 778	6,71%	0,00%	100,00%
CDCL	57	44 282	838 267	21 203 876	6,49%	0,00%	96,90%
OP	47	93 825	1 272 982	15 823 169	6,69%	7,29%	112,54%
OP_{LB}	51	81 903	1 432 879	21 504 421	6,57%	7,07%	111,80%
Surv	46	90 894	770 487	9 434 446	6,68%	0,00%	96,71%
Dép	51	82 559	755 573	11 111 976	6,53%	0,00%	105,35%
Dép_{LB}	60	66 141	1 151 240	18 133 495	6,72%	0,00%	109,52%

Les résultats en nombre d'instances résolues et en temps total d'exécution du CDCL à ordre partiel sans et avec littéraux bloqués sont légèrement moins bons que ceux du CDCL sans saut arrière sans et avec littéraux bloqués, respectivement (voir tableau 6.1). On peut envisager deux explications principales à ces résultats. Premièrement, on peut remarquer que, malgré un principe supposément plus simple, la vitesse d'exécution du CDCL à ordre partiel, c'est-à-dire la fréquence des étapes de propagation, reste quasiment identique à celle du CDCL sans saut arrière (en moyenne légèrement inférieure sans littéraux bloqués et légèrement supérieure avec littéraux bloqués). Par rapport à la version originale de GLUCOSE, la fréquence de vérification des clauses dans CDCL-OP_{LB} baisse donc d'environ 30%, et dans CDCL-OP elle est presque divisée par deux.

Le surcoût causé par le CDCL à ordre partiel est donc comparable à celui du CDCL sans saut arrière. En effet, lorsque le CDCL sans saut arrière doit à chaque propagation

TABLEAU 7.2: Comparaison entre différentes implémentations de CDCL et CDCL-OP sur le nombre d'instances qu'elles résolvent plus rapidement. Pour chaque paire d'implémentations, le tableau indique le nombre total d'instances que toutes deux résolvent dans le temps imparti, puis, parmi celles-ci, le nombre d'instances résolues plus rapidement par l'implémentation en ordonnée et l'implémentation en abscisse respectivement. Lorsque la même implémentation est à la fois en abscisse et en ordonnée, nous indiquons uniquement le nombre d'instances qu'elle résout dans le temps imparti. Les implémentations comparées et les instances considérées sont les mêmes que dans le tableau 7.1.

	$CDCL_{LB}$	CDCL	OP	OP_{LB}	Surv	Dép	$Dép_{LB}$
$CDCL_{LB}$	61	57 (36/20)	47 (34/12)	50 (38/11)	46 (38/7)	50 (40/9)	59 (45/13)
CDCL	57 (20/36)	57	43 (33/9)	47 (33/13)	44 (36/7)	47 (39/7)	55 (41/13)
OP	47 (12/34)	43 (9/33)	47	45 (14/30)	41 (23/17)	45 (20/24)	47 (13/33)
OP_{LB}	50 (11/38)	47 (13/33)	45 (30/14)	51	44 (29/14)	49 (26/22)	51 (17/33)
Surv	46 (7/38)	44 (7/36)	41 (17/23)	44 (14/29)	46	46 (18/27)	46 (9/36)
Dép	50 (9/40)	47 (7/39)	45 (24/20)	49 (22/26)	46 (27/18)	51	51 (13/37)
$Dép_{LB}$	59 (13/45)	55 (13/41)	47 (33/13)	51 (33/17)	46 (36/9)	51 (37/13)	60

TABLEAU 7.3: Comparaison de différentes implémentations de CDCL et CDCL-OP sur le temps total d'exécution pour les instances qu'elles résolvent en commun. Pour chaque paire d'implémentations, le tableau indique le temps total d'exécution nécessité respectivement par l'implémentation en ordonnée et l'implémentation en abscisse pour la résolution de l'ensemble des instances que les deux implémentations sont capables de résoudre dans le temps imparti. Lorsque la même implémentation est à la fois en abscisse et en ordonnée, nous indiquons uniquement le temps total d'exécution sur les instances qu'elle résout dans le temps imparti. Les implémentations comparées et les instances considérées sont les mêmes que dans le tableau 7.1.

	$CDCL_{LB}$	CDCL	OP	OP_{LB}	Surv	Dép	$Dép_{LB}$
$CDCL_{LB}$	32 901	23 297/26 283	21 898/39 826	20 947/39 308	16 263/33 294	20 296/42 270	32 238/58 700
CDCL	26 283/23 297	26 283	15 530/34 136	18 322/35 195	12 103/31 057	14 718/38 466	25 388/52 495
OP	39 826/21 898	34 136/15 530	39 826	35 404/29 207	26 562/26 064	35 404/34 583	39 826/28 445
OP_{LB}	39 308/20 947	35 195/18 322	29 207/35 404	42 303	26 664/30 648	37 714/38 861	42 303/36 990
Surv	33 294/16 263	31 057/12 103	26 064/26 562	30 648/26 664	33 294	33 294/31 608	33 294/24 365
Dép	42 270/20 296	38 466/14 718	34 583/35 404	38 861/37 714	31 608/33 294	42 960	42 960/32 281
$Dép_{LB}$	58 700/32 238	52 495/25 388	28 445/39 826	36 990/42 303	24 365/33 294	32 281/42 960	58 941

unitaire calculer le niveau de décision de cette propagation, ce niveau est connu sans calcul dans le CDCL à ordre partiel (comme dans le CDCL classique), mais il faut ajouter si besoin des dépendances entre le niveau courant et le niveau de décision de chaque littéral antécédent. La gestion des conflits est elle légèrement simplifiée, puisqu'elle se fait par niveaux de décision entiers et non pas littéral par littéral ; toutefois, les propagations unitaires étant une des opérations les plus fréquentes dans un CDCL, le surcoût causé par la gestion de Δ à chaque propagation est significatif et suffit à expliquer la baisse substantielle de fréquence de traitement du CDCL à ordre partiel.

Une seconde explication est que, comme le CDCL sans saut arrière, le CDCL à ordre partiel ne parvient généralement pas à raccourcir la taille des résolutions en nombre d'étapes de propagation : si l'on compare une des implémentations de CDCL-OP avec une des implémentations de CDCL, cette dernière fournit la résolution la plus courte sur 62% à 75% des instances résolues dans le temps imparti par les deux implémentations. Si l'on compare la taille totale des résolutions sur cet ensemble d'instances résolues en commun, l'avantage est là encore aux implémentations de CDCL, excepté pour la comparaison de CDCL_{LB} et CDCL-OP, où ce dernier parvient à légèrement réduire le nombre d'étapes de propagation total.

Des facteurs similaires à ceux du CDCL sans saut arrière peuvent entrer en jeu dans cette augmentation moyenne de la taille des résolutions : l'interférence avec les heuristiques de décision est d'autant plus nette que le CDCL à ordre partiel abolit la notion d'ordre total entre les niveaux de décision, et permet non seulement de défaire les niveaux dans un ordre non-chronologique mais aussi de défaire un niveau plus ancien que le niveau d'assertion du conflit. L'impact supposé de la sauvegarde de phase sur l'efficacité des redémarrages reste également pertinent. Enfin, le CDCL à ordre partiel permet comme le CDCL sans saut arrière la présence de conflits triviaux et redondants. Toutefois, on peut remarquer qu'ils sont bien moins fréquents dans le cas du CDCL à ordre partiel, puisqu'ils représentent en moyenne environ 7% des conflits, contre environ 32% en moyenne pour le CDCL sans saut arrière (et environ 15% en moyenne pour les variantes à propagations exhaustives du CDCL sans saut arrière). Cela signifie donc

TABLEAU 7.4: Comparaison de différentes implémentations de CDCL et CDCL-OP sur le nombre d'instances qu'elles résolvent en moins d'étapes de propagation. Pour chaque paire d'implémentations, le tableau indique le nombre total d'instances que toutes deux résolvent dans le temps imparti, puis, parmi celles-ci, le nombre d'instances résolues avec moins d'étapes de propagation par l'implémentation en ordonnée et l'implémentation en abscisse respectivement. Lorsque la même implémentation est à la fois en abscisse et en ordonnée, nous indiquons uniquement le nombre d'instances qu'elle résout dans le temps imparti. Les implémentations comparées et les instances considérées sont les mêmes que dans le tableau 7.1.

	CDCL_{LB}	CDCL	OP	OP_{LB}	Surv	Dép	Dép_{LB}
CDCL_{LB}	61	57 (24/33)	47 (29/18)	50 (35/15)	46 (29/17)	50 (30/20)	59 (38/21)
CDCL	57 (33/24)	57	43 (32/11)	47 (35/12)	44 (28/16)	47 (32/15)	55 (41/14)
OP	47 (18/29)	43 (11/32)	47	45 (25/20)	41 (7/34)	45 (10/35)	47 (17/30)
OP_{LB}	50 (15/35)	47 (12/35)	45 (20/25)	51	44 (11/33)	49 (8/41)	51 (14/37)
Surv	46 (17/29)	44 (16/28)	41 (34/7)	44 (33/11)	46	46 (22/24)	46 (24/22)
Dép	50 (20/30)	47 (15/32)	45 (35/10)	49 (41/8)	46 (24/22)	51	51 (27/24)
Dép_{LB}	59 (21/38)	55 (14/41)	47 (30/17)	51 (37/14)	46 (22/24)	51 (24/27)	60

TABLEAU 7.5: Comparaison de différentes implémentations de CDCL et CDCL-OP sur le nombre total d'étapes de propagation pour les instances qu'elles résolvent en commun. Pour chaque paire d'implémentations, le tableau indique le nombre total d'étapes de propagation nécessitées respectivement par l'implémentation en ordonnée et l'implémentation en abscisse pour la résolution de l'ensemble des instances que les deux implémentations sont capables de résoudre dans le temps imparti. Lorsque la même implémentation est à la fois en abscisse et en ordonnée, nous indiquons uniquement le nombre total d'étapes de propagation sur les instances qu'elle résout dans le temps imparti. Les implémentations comparées et les instances considérées sont les mêmes que dans le tableau 7.1.

	CDCL_{LB}	CDCL	OP	OP_{LB}	Surv	Dép	Dép_{LB}
CDCL_{LB}	985 261	608 438/451 900	686 664/627 446	650 533/783 283	521 712/310 549	689 543/421 317	976 349/945 041
CDCL	451 900/608 438	451 900	292 845/504 636	327 658/651 717	244 701/284 468	285 880/362 682	441 568/751 447
OP	627 446/686 664	504 636/292 845	627 446	565 946/609 332	428 023/261 978	565 946/367 333	627 446/543 223
OP_{LB}	783 283/650 533	651 717/327 658	609 332/565 946	864 248	575 097/307 158	806 606/412 666	864 248/674 058
Surv	310 549/521 712	284 468/244 701	261 978/428 023	307 158/575 097	310 549	310 549/308 894	310 549/420 907
Dép	421 317/689 543	362 682/285 880	367 333/565 946	412 666/806 606	308 894/310 549	427 890	427 890/573 273
Dép_{LB}	945 041/976 349	751 447/441 568	543 223/627 446	674 058/864 248	420 907/310 549	573 273/427 890	951 223

qu'une plus petite proportion de la recherche est passée à découvrir des conflits qui pourraient être évités par propagations de clauses déjà connues. Les conflits redondants doivent donc moins handicaper les performances du CDCL à ordre partiel que celles du CDCL sans saut arrière. Nous verrons tout de même dans la sous-section 7.7.4 que malgré leur faible proportion, l'élimination totale des conflits triviaux et redondants a un impact important sur les performances de l'algorithme. Notons également que la non-exhaustivité des conflits dans CDCL-OP_{LB} semble avoir un impact encore plus négligeable que dans CDCL-SSA_{LB}, puisque seules 2 instances sur 62 nécessitent plus d'une vérification finale et que ce nombre de vérifications finales est au plus de 5.

Cependant, le CDCL à ordre partiel a un inconvénient supplémentaire par rapport au CDCL sans saut arrière : contrairement à ce dernier, il ne permet pas de réduire en moyenne la destructivité des sauts arrière. En effet, comme nous l'avons indiqué dans la sous-section 7.1.2, le CDCL à ordre partiel ne garantit pas de réduire la destructivité des conflits par rapport au CDCL classique, contrairement au CDCL sans saut arrière, dont la résolution de conflit est assurée de défaire moins d'instanciations qu'un saut arrière conventionnel dans la même situation. Cette propriété est confirmée en pratique, car nos résultats expérimentaux indiquent que la destructivité moyenne de CDCL-OP et CDCL-OP_{LB} reste très proche de celle du CDCL classique (à peine inférieure à celle de CDCL_{LB}, un peu supérieure à celle de CDCL), que l'on compare les implémentations sur l'ensemble des instances ou seulement sur leurs instances résolues en commun.

Or c'est cette réduction de la destructivité des conflits qui semble permettre au CDCL sans saut arrière de réduire la distance moyenne entre les conflits. De fait, les deux implémentations de CDCL à ordre partiel, quant à elles, ne réduisent pas cette distance entre conflits par rapport à l'implémentation originale de GLUCOSE ; au contraire, elles augmentent cette distance d'environ 12% en moyenne, sans compter la présence de conflits redondants qui augmente donc davantage la distance entre conflits utiles à l'élagage de l'espace de recherche. Si le CDCL à ordre partiel détruit à chaque conflit une quantité d'instanciation en moyenne équivalente à celle du CDCL ordinaire, il semble donc conserver des instances moins susceptibles de conduire rapidement à de nouveaux

TABLEAU 7.6: Comparaison de différentes implémentations de CDCL et CDCL-OP sur le nombre d'instances qu'elles résolvent avec une plus petite destructivité des conflits. Pour chaque paire d'implémentations, le tableau indique le nombre total d'instances que toutes deux résolvent dans le temps imparti, puis, parmi celles-ci, le nombre d'instances résolues avec une plus petite destructivité des conflits par l'implémentation en ordonnée et l'implémentation en abscisse respectivement. Lorsque la même implémentation est à la fois en abscisse et en ordonnée, nous indiquons uniquement le nombre d'instances qu'elle résout dans le temps imparti. Les implémentations comparées et les instances considérées sont les mêmes que dans le tableau 7.1.

	CDCL_{LB}	CDCL	OP	OP_{LB}	Surv	Dép	Dép_{LB}
CDCL_{LB}	61	57 (28/28)	47 (29/18)	50 (28/22)	46 (29/17)	50 (26/24)	59 (32/27)
CDCL	57 (28/28)	57	43 (28/15)	47 (28/19)	44 (29/15)	47 (29/18)	55 (32/23)
OP	47 (18/29)	43 (15/28)	47	45 (23/21)	41 (27/14)	45 (15/30)	47 (21/26)
OP_{LB}	50 (22/28)	47 (19/28)	45 (21/23)	51	44 (28/16)	49 (16/33)	51 (21/30)
Surv	46 (17/29)	44 (15/29)	41 (14/27)	44 (16/28)	46	46 (14/32)	46 (16/30)
Dép	50 (24/26)	47 (18/29)	45 (30/15)	49 (33/16)	46 (32/14)	51	51 (29/21)
Dép_{LB}	59 (27/32)	55 (23/32)	47 (26/21)	51 (30/21)	46 (30/16)	51 (21/29)	60

TABLEAU 7.7: Comparaison de différentes implémentations de CDCL et CDCL-OP sur la destructivité moyenne des conflits pour les instances qu'elles résolvent en commun. Pour chaque paire d'implémentations, le tableau indique la moyenne des destructivité des conflits respectivement dans l'implémentation en ordonnée et l'implémentation en abscisse sur l'ensemble des instances que les deux implémentations sont capables de résoudre dans le temps imparti. Lorsque la même implémentation est à la fois en abscisse et en ordonnée, nous indiquons uniquement la destructivité moyenne des conflits sur les instances qu'elle résout dans le temps imparti. Les implémentations comparées et les instances considérées sont les mêmes que dans le tableau 7.1.

	CDCL_{LB}	CDCL	OP	OP_{LB}	Surv	Dép	Dép_{LB}
CDCL_{LB}	6,77%	6,55%/6,34%	6,43%/6,38%	7,13%/6,86%	5,94%/5,82%	6,18%/5,94%	6,94%/6,93%
CDCL	6,34%/6,55%	6,34%	6,01%/5,96%	6,47%/6,42%	5,64%/5,62%	5,62%/5,54%	6,51%/6,73%
OP	6,38%/6,43%	5,96%/6,01%	6,38%	6,58%/6,46%	6,29%/6,26%	6,58%/6,37%	6,38%/6,45%
OP_{LB}	6,86%/7,13%	6,42%/6,47%	6,46%/6,58%	6,79%	5,98%/6,04%	6,19%/6,08%	6,79%/7,03%
Surv	5,82%/5,94%	5,62%/5,64%	6,26%/6,29%	6,04%/5,98%	5,82%	5,82%/5,71%	5,82%/6%
Dép	5,94%/6,18%	5,54%/5,62%	6,37%/6,58%	6,08%/6,19%	5,71%/5,82%	5,87%	5,87%/6,15%
Dép_{LB}	6,93%/6,94%	6,73%/6,51%	6,45%/6,38%	7,03%/6,79%	6%/5,82%	6,15%/5,87%	6,86%

conflits.

En résumé, le CDCL à ordre partiel ne parvient pas à surpasser les performances du CDCL sans saut arrière car il souffre de problèmes similaires : un surcoût de travail comparable et de probables interactions négatives avec d'autres aspects de l'algorithme. S'il déclenche en moyenne beaucoup moins de conflits redondants, il a toutefois l'inconvénient de ne pas réduire la destructivité moyenne des conflits, ce qui l'empêche de réduire la distance entre les conflits ainsi que la taille globale des résolutions. Cette incapacité à réduire en pratique la destructivité moyenne des conflits est a priori un inconvénient important, puisqu'il s'agit du but original du CDCL à ordre partiel. Cependant, nous verrons dans la sous-section 7.9.2 que, sur certaines instances, la destructivité moyenne peut être au choix significativement réduite ou augmentée en utilisant des heuristiques particulières pour le choix des niveaux d'assertion ; de plus, nous montrerons aussi que, selon la satisfaisabilité de l'instance, la réduction ou l'augmentation de la destructivité des conflits peuvent toutes deux améliorer l'efficacité de résolution.

7.7 Propagations exhaustives dans un CDCL à ordre partiel

Contrairement au CDCL sans saut arrière, le CDCL à ordre partiel est à niveau de propagation unique, ce qui permet d'obtenir un algorithme non-trivial et non-redondant si l'exhaustivité des propagations est assurée, cette condition étant elle-même remplie si toutes les clauses sont entièrement surveillées en tout temps.

L'exemple utilisé dans la preuve de la proposition 7.5 montre que le manque de surveillance des clauses est lié aux littéraux surveillés vrais. En effet, si une clause c est surveillée par un littéral w_1 vrai, le second littéral surveillé w_2 ne sera pas remplacé s'il devient faux. Or, si $\lambda(w_1) \not\asymp_\Delta \lambda(w_2)$, un saut arrière partiel peut défaire w_1 sans défaire w_2 ; dans ce cas, c n'est plus entièrement surveillée si w_2 a été entièrement propagé.

Par conséquent, pour maintenir en tout temps une surveillance entière sur les clauses, il est nécessaire que pour toute clause c surveillée par un littéral vrai w_1 et un littéral faux déjà propagé w_2, le niveau $\lambda(w_2)$ dépende du niveau $\lambda(w_1)$ si les deux ni-

veaux sont différents. Ainsi, si w_1 est défait, w_2 le sera aussi et donc c restera entièrement surveillé par deux littéraux indéfinis.

Cette stratégie de maintenance de la surveillance entière a toutefois comme inconvénient de rajouter des dépendances supplémentaires entre niveaux de décision, ce qui peut évidemment accroître la destructivité des sauts arrière partiels ultérieurs. Nous en avons donc conçu deux variantes qui cherchent ou non à minimiser le nombre de dépendances ajoutées.

Dans le premier cas, pour éviter si possible d'ajouter une dépendance entre $\lambda(w_1)$ et $\lambda(w_2)$, le littéral surveillé faux w_2 est remplacé par un littéral indéfini ou vrai de $c\backslash\{w_1, w_2\}$ s'il en existe, malgré la présence du littéral surveillé vrai w_1. Une dépendance entre $\lambda(w_1)$ et $\lambda(w_2)$ est alors ajoutée uniquement si w_2 ne peut pas être remplacé. Nous dirons que cette tactique assure l'exhaustivité des propagations par surveillance accrue.

La seconde tactique, à l'opposé, ne cherche pas à minimiser le nombre de dépendances ajoutées et ajoute une dépendance $i\Delta\lambda_c$ dès qu'un littéral de niveau i évite de remplacer un littéral surveillé faux de niveau courant dans une clause. Nous parlerons d'exhaustivité des propagations par dépendances supplémentaires.

Les sous-sections 7.7.1 et 7.7.2 présentent respectivement les stratégies de propagations exhaustives par surveillance accrue et dépendances supplémentaires. La sous-section 7.7.3 démontre l'exhaustivité des propagations dans ces deux algorithmes et ses conséquences ; nous verrons notamment que les littéraux bloqués peuvent être intégrés au CDCL à ordre partiel et dépendances supplémentaires sans dégrader ses propriétés. Enfin, la sous-section 7.7.4 vérifie expérimentalement l'efficacité de l'implémentation de ces algorithmes.

7.7.1 Propagations exhaustives par surveillance accrue

Nous noterons CDCL-OP-Surveillance la variante du CDCL à ordre partiel qui assure l'exhaustivité des propagations par surveillance accrue des clauses. Cet algorithme

est strictement identique au CDCL à ordre partiel de base (algorithme 7.1), à l'exception de la procédure PROPAGER qui est décrite par l'algorithme 7.6. Son principe est de remplacer tant que possible tous les littéraux surveillés faux sans exception, et d'ajouter une dépendance entre niveaux en dernier recours. La surveillance accrue est similaire à la stratégie utilisée dans le CDCL sans saut arrière à détection anticipée des propagations alternatives (voir sous-section 6.8.2), mais a cependant un but différent : dans CDCL-OP-Surveillance, la surveillance accrue sert à éviter de rajouter des dépendances supplémentaires entre niveaux de décision, alors que dans CDCL-SSA-PropAltAnt, elle sert à réduire le nombre de propagations alternatives candidates pour chaque littéral instancié.

Par conséquent, lorsqu'une clause c est vérifiée parce qu'un littéral l a été instancié et que c est surveillée par $\neg l$, l'algorithme cherche un remplaçant pour $\neg l$ dans tous les cas, y compris lorsque le second littéral surveillé w est vrai. S'il n'existe pas un tel remplaçant, la clause est conflictuelle, unitaire ou satisfaite, selon que w est respectivement faux, indéfini ou vrai. La gestion des deux premiers cas est identique au CDCL conventionnel. Dans le cas où w est vrai, il est nécessaire, pour garantir que la clause reste entièrement surveillée, d'interdire de désinstancier w tout en conservant $\neg l$. Cette contrainte est assurée si le niveau de $\neg l$ (c'est-à-dire le niveau courant λ_c) dépend de celui de w ou y est identique. Sinon, s'il existe un autre littéral faux dans la clause dont le niveau dépend de $\lambda(w)$ ou y est identique, il est possible de le surveiller à la place de $\neg l$ (lignes 18 à 25). Si aucun littéral ne satisfait ces conditions, il est alors nécessaire de rajouter la dépendance $\lambda(w)\Delta\lambda_c$ (ligne 27). Notons qu'en pratique nous nous restreignons à chercher un autre littéral faux qui dépend directement de $\lambda(w)$ afin de ne pas avoir à calculer une partie de la fermeture transitive de Δ. Cela ne modifie pas les propriétés de l'algorithme (en particulier la surveillance entière des clauses), mais cela risque de rajouter une dépendance entre $\lambda(w)$ et λ_c alors qu'un autre littéral faux de la clause dépend indirectement de λ_c et aurait donc pu être surveillé à la place de w sans nécessiter d'ajout de dépendance.

Le CDCL à ordre partiel et surveillance accrue est décrit sans utilisation des

Algorithme 7.6 PROPAGER() *[CDCL-OP-Surveillance]*

```
1: Π ← {l ∈ σ | propagé(l) = faux}
2: tant que Π ≠ ∅ faire
3:     choisir l ∈ Π
4:     pour c ∈ C | ¬l ∈ ω(c) faire /*¬l est surveillé dans c*/
5:         w ← ω(c) \ {¬l} /*w est le second littéral surveillé dans c*/
6:         Ω ← {l' ∈ c \ {w} | σ(l') ≠ faux}
7:         /*Ω est l'ensemble des littéraux pouvant remplacer ¬l*/
8:         si Ω ≠ ∅ alors
9:             choisir w'∈Ω
10:            ω(c) ← {w, w'} /*w' est surveillé à la place de ¬l*/
11:        sinon /*tous les autres littéraux de c sont faux*/
12:            si σ(w) = indéfini alors /*c est unitaire*/
13:                INSTANCIER(w, c) /*w est propagé par c*/
14:                Π ← Π ∪ {w} /*w doit être lui-même propagé*/
15:            sinon si σ(w) = faux alors
16:                retourner c /*c est un conflit*/
17:            sinon /*σ(w) = vrai*/
18:                si λ(w) ⋠Δ λ(l) alors
19:                    Υ ← {l' ∈ c \ {¬l, w} | λ(w) ⪯Δ λ(l')}
20:                    /*Υ contient les littéraux de c dont le niveau*/
21:                    /*dépend du niveau de w ou y est identique*/
22:                    si Υ ≠ ∅ alors
23:                        choisir l' ∈ Υ
24:                        ω(c) ← {l', w}
25:                        /*si w est désinstancié, l' le sera aussi*/
26:                    sinon
27:                        Δ ← Δ ∪ {λ(w), λc}
28:                        /*si w est désinstancié, ¬l le sera aussi*/
29:    Π ← Π \ {l}
30:    propagé(l) ← vrai
31: retourner NUL /* aucun conflit rencontré */
```

littéraux bloqués. En effet, leur intégration serait contradictoire avec le principe même de surveillance accrue, puisque le but des littéraux bloqués est au contraire de réduire le travail de surveillance des clauses. Nous ne considérerons donc pas de variante de CDCL-OP-Surveillance gérant les littéraux bloqués, bien qu'il soit possible de les intégrer tout en conservant l'exhaustivité des propagations de la même façon que pour le CDCL à ordre partiel et dépendances supplémentaires.

7.7.2 Propagations exhaustives par dépendances supplémentaires

Le CDCL à ordre partiel et surveillance accrue garantit l'exhaustivité des propagations tout en cherchant à minimiser le nombre d'ajouts de dépendances entre niveaux, dans le but de réduire la destructivité des sauts arrière partiels à venir ; pour cela, il continue entre autres de remplacer tous les littéraux surveillés faux, même lorsque le second littéral surveillé de la clause est vrai.

Le CDCL à ordre partiel et dépendances supplémentaires, noté CDCL-OP-Dépendances, choisit une tactique opposée et cherche avant tout à assurer l'exhaustivité des propagations en minimisant le travail supplémentaire ; en particulier, il évite la surveillance accrue des clauses. Ainsi, lorsque dans une clause c un littéral surveillé w_1 est vrai, le second littéral surveillé w_2 ne sera pas remplacé même s'il est faux. Ce comportement est identique à la stratégie d'un CDCL classique. En contrepartie, pour éviter que l'algorithme brise la surveillance entière de la clause en désinstanciant w_1 mais pas w_2, lorsque le littéral surveillé faux w_2 est propagé, la dépendance $\lambda(w_1)\Delta\lambda_c$ est ajoutée (car $\lambda_c = \lambda(w_2)$). Intuitivement, cette dépendance signifie que les propagations unitaires au niveau courant peuvent être interrompues grâce au littéral w_1 de niveau $\lambda(w_1)$. CDCL-OP-Dépendances rajoute donc systématiquement des dépendances entre littéraux surveillés vrais et faux afin de garantir la surveillance entière des clauses sans compliquer la gestion des littéraux surveillés, au prix d'une augmentation des dépendances entre niveaux de décision.

Comme pour CDCL-OP-Surveillance, le pseudo-code de CDCL-OP-Dépendances

Algorithme 7.7 PROPAGER() *[CDCL-OP-Dépendances]*

```
1: Π ← {l ∈ σ | propagé(l) = faux}
2: tant que Π ≠ ∅ faire
3:     choisir l ∈ Π
4:     pour c ∈ C | ¬l ∈ ω(c) faire /*¬l est surveillé dans c*/
5:         w ← ω(c) \ {¬l}  /*w est le second littéral surveillé dans c*/
6:         si σ(w) = vrai alors
7:             si λ(w) ≠ λc alors /*λ(¬l) = λc*/
8:                 Δ ← Δ ∪ {λ(w)Δλc}
9:                 /*w évite de continuer les propagations au niveau λc*/
10:        sinon
11:            Ω ← {l' ∈ c \ {w} | σ(l') ≠ faux}
12:            /*Ω est l'ensemble des littéraux pouvant remplacer ¬l*/
13:            si Ω ≠ ∅ alors
14:                choisir w'∈Ω
15:                ω(c) ← {w, w'}  /*w' est surveillé à la place de ¬l*/
16:            sinon /*tous les autres littéraux de c sont faux*/
17:                si σ(w) = indéfini alors /*c est unitaire*/
18:                    INSTANCIER(w, c)  /*w est propagé par c*/
19:                    Π ← Π∪{w}  /*w doit être lui-même propagé*/
20:                sinon
21:                    retourner c  /*c est un conflit*/
22:    Π ← Π \ {l}
23:    propagé(l) ← vrai
24: retourner NUL  /*aucun conflit rencontré*/
```

est identique à celui du CDCL à ordre partiel de base (algorithme 7.1) à l'exception de la procédure PROPAGER, dont la modification est décrite par l'algorithme 7.7. La seule modification par rapport à la version originale de la procédure (algorithme 2.15) consiste à ajouter la dépendance $\lambda(w)\Delta\lambda_c$ lorsque le littéral surveillé faux $\neg l$ de niveau courant n'est pas remplacé parce que le second littéral surveillé w est vrai (lignes 6 à 9 de l'algorithme 7.7).

Un grand avantage en pratique de CDCL-OP-Dépendances sur CDCL-OP-Surveillance est que son principe général, qui tend à minimiser le plus possible le travail de gestion des littéraux surveillés, est similaire à celui des littéraux bloqués. En fait, il est possible d'intégrer la gestion des littéraux surveillés à CDCL-OP-Dépendances tout en conservant l'exhaustivité des propagations unitaires. En effet, les littéraux bloqués ne

Algorithme 7.8 PROPAGER() *[CDCL-OP-Dépendances$_{LB}$]*

```
1: Π ← {l ∈ σ | propagé(l) = faux}
2: tant que Π ≠ ∅ faire
3:     choisir l ∈ Π
4:     pour c ∈ C | ¬l ∈ ω(c) faire /*¬l est surveillé dans c*/
5:         si σ(β(c, ¬l)) = vrai alors
6:             si λ(β(c, ¬l)) ≠ λc alors /*λ(¬l) = λc*/
7:                 Δ ← Δ ∪ {λ(β(c, ¬l))Δλc}
8:                 /*le littéral bloqué évite de remplacer ¬l*/
9:         sinon
10:            w ← ω(c) \ {¬l} /*w est le second littéral surveillé dans c*/
11:            si σ(w) = vrai alors
12:                β(c, ¬l) ← w /*w est le nouveau littéral bloqué de ¬l*/
13:                si λ(w) ≠ λc alors /*λ(¬l) = λc*/
14:                    Δ ← Δ ∪ {λ(w)Δλc} /*w évite de continuer*/
15:                    /*les propagations au niveau λc*/
16:            sinon
17:                Ω ← {l' ∈ c \ {w} | σ(l') ≠ faux}
18:                /*Ω est l'ensemble des littéraux pouvant remplacer ¬l*/
19:                si Ω ≠ ∅ alors
20:                    choisir w'∈Ω
21:                    ω(c) ← {w, w'}
22:                    /*w' est surveillé à la place de ¬l*/
23:                    β(c, w') ← β(c, ¬l)
24:                    /*transmission du littéral bloqué*/
25:                sinon /*tous les autres littéraux de c sont faux*/
26:                    si σ(w) = indéfini alors /*c est unitaire*/
27:                        INSTANCIER(w, c) /*w est propagé par c*/
28:                        Π ← Π∪{w} /*w doit être lui-même propagé*/
29:                    sinon
30:                        retourner c /*c est un conflit*/
31:    Π ← Π \ {l}
32:    propagé(l) ← vrai
33: retourner NUL /* aucun conflit rencontré */
```

font que rajouter un cas de non-remplacement de littéraux surveillés faux. Si w_1 est un littéral surveillé rendu faux dans une clause c, il peut ne pas être remplacé si le second littéral surveillé w_2 est vrai, mais aussi si son littéral bloqué associé $\beta(c, w_1)$ est vrai. Dans ce deuxième cas, la surveillance entière de c peut être brisée ultérieurement si un saut arrière partiel défait $\beta(c, w_1)$ sans défaire w_1. Pour éviter cela, il suffit alors lors du non-remplacement de w_1 d'ajouter la dépendance $\lambda(\beta(c, w_1))\Delta\lambda_c$ (le littéral w_1 étant de niveau courant). Cette gestion est strictement identique au cas où le non-remplacement de w_1 est permis par le second littéral surveillé w_2.

Les modifications apportées à la procédure PROPAGER pour la gestion des littéraux bloqués sont indiquées par l'algorithme 7.8. La principale différence avec l'implémentation sans littéraux bloqués (algorithme 7.7) est la possibilité de ne pas remplacer un littéral surveillé faux lorsque son littéral bloqué associé est vrai, ainsi que l'ajout de dépendance correspondant (lignes 5 à 8). Les seules autres différences sont le remplacement d'un littéral bloqué (ligne 12) et la transmission du littéral bloqué lors du remplacement d'un littéral surveillé (ligne 23).

7.7.3 Propriétés des CDCL à ordre partiel et propagations exhaustives

Cette sous-section démontre les propriétés des trois algorithmes décrits dans les sous-sections 7.7.1 et 7.7.2. Elle prouve notamment qu'ils sont bien à surveillance entière, et donc à propagations exhaustives. Nous verrons également qu'ils conservent la correction totale du CDCL à ordre partiel de base. Si leur correction découle des propriétés de surveillance, la complétude, la terminaison et la complexité temporelle se démontrent de la même façon que pour le CDCL à ordre partiel de base :

Proposition 7.8. *CDCL-OP-Surveillance, CDCL-OP-Dépendances et CDCL-OP-Dépendances$_{LB}$ sont complets, terminent et ont une complexité temporelle exponentielle au pire des cas.*

Démonstration. Les preuves de complétude, de terminaison et de complexité temporelle

du CDCL à ordre partiel de base (respectivement propositions 7.1, 7.2, et 7.3) restent valides pour ces trois variantes. □

Nous allons maintenant démontrer la surveillance partielle pour les trois variantes de l'algorithme.

Proposition 7.9. *CDCL-OP-Surveillance est à surveillance entière.*

Démonstration. Initialement, toutes les clauses sont entièrement surveillées puisqu'aucune variable n'est instanciée.

Lorsqu'un littéral l est instancié, il peut uniquement défaire la surveillance des clauses c dans lesquelles $\neg l$ est un littéral surveillé. Supposons que c est entièrement surveillée avant l'instanciation de l ; alors w, le second littéral surveillé de c, est vrai, indéfini, ou faux mais pas encore propagé. L'algorithme cherche à remplacer $\neg l$ par un littéral vrai ou indéfini dans $c \setminus \{\neg l, w\}$. Si ce remplacement est possible, c reste entièrement surveillée. Sinon, si w est vrai, il suffit à maintenir la surveillance entière de c. Si w est indéfini, c est unitaire et w est instancié comme propagation de c, ce qui assure la surveillance entière de c. Si w est faux, la clause c entière est fausse, et le saut arrière partiel résultant défait l. Comme par hypothèse w n'est pas encore propagé, c est alors entièrement surveillée. Par conséquent, toutes les clauses sont toujours entièrement surveillées après l'instanciation et la propagation de l.

Supposons qu'un saut arrière partiel survient alors que toutes les clauses sont entièrement surveillées. Ce saut arrière peut uniquement briser la surveillance des clauses surveillées par un littéral vrai w_1 et un littéral faux et déjà propagé w_2, si w_1 est défait et w_2 est conservé. Il est possible que w_2 ait été instancié par propagation de la clause unitaire c, qui a alors été détectée par vérification après instanciation de $\neg w_1$. Dans ce cas, par l'unicité du niveau de propagation, on a $\lambda(w_1) = \lambda(w_2)$. Sinon, w_1 était déjà instancié lors de la vérification de c par propagation de $\neg w_2$, et c ne comportait aucun autre littéral vrai ou indéfini. CDCL-OP-Surveillance a alors ajouté la dépendance $\lambda(w_1) \preceq_\Delta \lambda(w_2)$. Par conséquent, si le saut arrière défait w_1, il défait forcément aussi

w_2. Toutes les clauses seront donc toujours entièrement surveillées après le saut arrière partiel. □

Proposition 7.10. *CDCL-OP-Dépendances est à surveillance entière.*

Démonstration. La preuve est entièrement identique à celle de la surveillance entière de CDCL-OP-Surveillance, à deux différences près.

Lors de l'instanciation d'un littéral l et de la vérification d'une clause c où $\neg l$ est surveillé, $\neg l$ n'est pas remplacé si w, le second littéral surveillé de c, est vrai. Dans ce cas, w suffit à conserver la surveillance entière de c.

Lors d'un saut arrière, dans le cas d'une clause c dont un littéral surveillé w_1 est vrai et le second w_2 est faux, si w_1 était déjà instancié lors de la vérification de c par propagation de $\neg w_2$, il est possible que c aie alors contenu d'autres littéraux vrais ou indéfinis. L'algorithme a alors tout de même ajouté la dépendance $\lambda(w_1) \preceq_\Delta \lambda(w_2)$. □

Proposition 7.11. *CDCL-OP-Dépendances$_{LB}$ est à surveillance entière.*

Démonstration. Initialement, toutes les clauses sont entièrement surveillées puisqu'aucune variable n'est instanciée.

Lorsqu'un littéral l est instancié, il peut uniquement défaire la surveillance des clauses c dans lesquelles $\neg l$ est un littéral surveillé. Supposons que c est entièrement surveillée avant l'instanciation de l ; alors w, le second littéral surveillé de c, est vrai, indéfini, ou faux mais pas encore propagé, ou alors son littéral bloqué associé $\beta(c, w)$ est vrai. Si $\beta(c, \neg l)$ est vrai, $\neg l$ n'est pas remplacé dans c. Celle-ci reste entièrement surveillée, puisque le littéral bloqué de $\neg l$ est vrai et w ou son littéral bloqué vérifie une des propriétés nécessaires.

Si $\beta(c, \neg l)$ n'est pas vrai mais w est vrai, $\neg l$ n'est pas remplacé et w suffit à conserver la surveillance entière de c. Si ni $\beta(c, \neg l)$ ni w ne sont vrais, l'algorithme cherche à remplacer $\neg l$ par un littéral vrai ou indéfini dans $c \setminus \{\neg l, w\}$. Si ce remplacement

est possible, c reste entièrement surveillée. Sinon, si w est indéfini, c est unitaire et w est instancié comme propagation de c, ce qui assure la surveillance entière de c. Si w est faux, la clause c entière est fausse, et le saut arrière partiel résultant défait l. Par hypothèse, soit w n'est pas encore propagé, soit $\beta(c, w)$ est vrai ; comme $\neg l$ est indéfini, c est entièrement surveillée. Par conséquent, toutes les clauses sont toujours entièrement surveillées après l'instanciation et la propagation de l.

Supposons qu'un saut arrière partiel survient alors que toutes les clauses sont entièrement surveillées. Ce saut arrière peut briser la surveillance des clauses c où au moins l'un des deux littéraux surveillés w_1 et w_2 est faux et déjà propagé. Si w_1 et w_2 sont tous deux faux et déjà propagés, alors leurs littéraux bloqués respectifs $\beta(c, w_1)$ et $\beta(c, w_2)$ sont vrais. Le saut arrière peut briser la surveillance entière de c si l'un des deux littéraux bloqués est défait mais son littéral surveillé associé est conservé. Or, lorsque la clause c a été vérifiée par propagation de $\neg w_1$ et $\neg w_2$ respectivement, les dépendances $\lambda(\beta(c, w_1)) \preceq_\Delta \lambda(w_1)$ et $\lambda(\beta(c, w_2)) \preceq_\Delta \lambda(w_2)$ ont été ajoutées. Par conséquent, si le retour arrière partiel défait un littéral bloqué, il défait aussi le littéral surveillé correspondant. La clause c reste donc entièrement surveillée.

Supposons maintenant qu'un seul des littéraux surveillés de c est faux et déjà propagé (sans perdre de généralité, nous supposerons qu'il s'agit de w_1). Comme c est entièrement surveillée avant le saut arrière, soit w_2 est vrai et $\beta(c, w_1)$ est faux ou indéfini, soit w_2 est faux ou indéfini et $\beta(c, w_1)$ est vrai, soit w_2 et $\beta(c, w_1)$ sont tous deux vrais.

Si w_2 est vrai et $\beta(c, w_1)$ est faux ou indéfini, alors la surveillance entière de c peut être compromise si le saut arrière défait w_2 mais conserve w_1. Or, lors de la vérification de c par propagation de l'instanciation $\neg w_1$, la dépendance $\lambda(w_2) \preceq_\Delta \lambda(w_1)$ a été ajoutée, que ce soit parce que w_2 était déjà instancié et a permis de ne pas remplacer w_1 ou parce que l'instanciation $\neg w_2$ a rendu c unitaire et a entraîné la propagation de w_1. Si le saut arrière défait w_2, il défait donc également w_1, et c reste entièrement surveillée dans tous les cas.

Si w_2 est faux ou indéfini et $\beta(c, w_1)$ est vrai, alors la surveillance entière de c est

compromise si le saut arrière défait $\beta(c, w_1)$ sans défaire w_1. Or, lors de la vérification de c par propagation de l'instanciation $\neg w_1$, la dépendance $\lambda(\beta(c, w_1)) \preceq_\Delta \lambda(w_1)$ a été ajoutée. Si le saut arrière défait $\beta(c, w_1)$, il défait donc également w_1, et c reste entièrement surveillée dans tous les cas. Notons que si w_2 est faux, il n'est pas encore propagé par hypothèse et n'empêche donc pas la surveillance entière de la clause.

Si w_2 et $\beta(c, w_1)$ sont tous deux vrais, alors la surveillance entière de c est compromise si le saut arrière défait à la fois w_2 et $\beta(c, w_1)$ sans défaire w_1. Lorsque c a été vérifiée par propagation de $\neg w_1$, si $\beta(c, w_1)$ était déjà instancié, la dépendance $\lambda(\beta(c, w_1) \preceq_\Delta \lambda(w_1)$ a été ajoutée. Sinon, w_2 était déjà instancié et la dépendance $\lambda(w_2) \preceq_\Delta \lambda(w_1)$ a été ajoutée. Il est impossible que ni $\beta(c, w_1)$ ni w_2 n'aient été instanciés au moment de la propagation de $\neg w_1$, car le non-remplacement du littéral vrai w_1 signifierait que c était alors unitaire, ce qui est impossible puisque $\beta(c, w_1)$ et w_2 sont tous deux vrais. Au final, dans tous les cas de figures, on a soit $\lambda(\beta(c, w_1) \preceq_\Delta \lambda(w_1)$, soit $\lambda(w_2) \preceq_\Delta \lambda(w_1)$, donc si $\beta(c, w_1)$ et w_2 sont tous deux défaits, w_1 est également défait et c reste entièrement surveillée. □

Les propagations exhaustives, la correction totale, la non-trivialité et la non-redondance des trois algorithmes sont des conséquences directes de leur surveillance entière.

Corollaire 7.4. *CDCL-OP-Surveillance, CDCL-OP-Dépendances et CDCL-OP-Dépendances$_{LB}$ sont à propagations exhaustives.*

Démonstration. Ces algorithmes sont à surveillance entière d'après les propositions 7.9, 7.10 et 7.11 respectivement ; ils sont donc à propagations exhaustives d'après la proposition 5.3. □

Corollaire 7.5. *CDCL-OP-Surveillance, CDCL-OP-Dépendances et CDCL-OP-Dépendances$_{LB}$ sont totalement corrects.*

Démonstration. Ces algorithmes sont à surveillance entière d'après les propositions 7.9,

7.10 et 7.11 respectivement ; ils sont donc également à surveillance partielle d'après le corollaire 5.2, ce qui prouve leur correction par le corollaire 5.3. Comme les algorithmes sont également complets et terminent, d'après la proposition 7.8, ils sont donc totalement corrects. □

Corollaire 7.6. *CDCL-OP-Surveillance, CDCL-OP-Dépendances et CDCL-OP-Dépendances$_{LB}$ sont non-triviaux et non-redondants.*

Démonstration. Ces algorithmes sont à propagations exhaustives d'après le corollaire 7.4. Ils sont également à niveau de propagation unique et utilisent la stratégie d'apprentissage du premier point d'implication unique. Le corollaire 5.5 prouve donc qu'ils sont également non-triviaux et non-redondants. □

7.7.4 Résultats expérimentaux

Les résultats expérimentaux obtenus par les implémentations de CDCL-OP-Surveillance, CDCL-OP-Dépendances et CDCL-OP-Dépendances$_{LB}$ sont résumés dans les tableaux 7.1 à 7.7. Ceux-ci confirment notamment, comme nous l'avons démontré dans la sous-section précédente, qu'aucun conflit trivial ne survient lors de l'exécution de ces algorithmes.

CDCL-OP-Surveillance est caractérisée, comme l'on pouvait s'y attendre, par une fréquence de traitement en baisse, qui ne représente qu'environ 60% de la fréquence du CDCL à ordre partiel de base (sans littéraux bloqués) et moins d'un tiers de la fréquence de l'implémentation originale de GLUCOSE. Cette baisse de fréquence de traitement s'explique par les tâches supplémentaires à effectuer lors des propagations unitaires lorsqu'une clause vérifiée a un littéral vrai et l'autre faux : CDCL-OP-Surveillance doit alors chercher un remplaçant au littéral faux si possible ou rajouter une dépendance entre niveaux dans le cas contraire, tandis que le CDCL à ordre partiel de base n'effectue absolument aucune manipulation dans ce cas de figure. Cependant, malgré ce handicap, les performances en temps et en nombre de résolutions de CDCL-OP-Surveillance sont

comparables à ceux de CDCL-OP, puisqu'elle parvient à résoudre une instance de moins, tout en réduisant le temps de calcul total de près de 50 minutes.

Si le taux de destructivité des conflits reste quasiment identique, c'est en fait l'élimination des conflits redondants qui semble être responsable de ces performances relativement bonnes, compte tenu de la forte baisse de rapidité de l'algorithme. Puisqu'aucun des conflits n'est redondant, aucune partie de la recherche n'est passée à redécouvrir une clause déjà connue qui ne contribue pas à l'élagage de l'espace de recherche. De fait, les tableaux 7.4 et 7.5 indiquent que CDCL-OP-Surveillance parvient à réduire la taille de la résolution par rapport à CDCL-OP et CDCL-OP$_{\text{LB}}$ dans une grande majorité de leurs instances résolues en commun, et réduit le nombre total d'étapes de propagation sur ces instances de respectivement 39% et 46%, alors que la surveillance accrue des clauses devrait au contraire contribuer à l'augmentation des étapes de propagation.

Il est donc probable que l'élimination des conflits redondants contribue à cette réduction de la taille des résolutions. Par ailleurs, la distance moyenne entre conflits est significativement réduite par rapport à CDCL-OP et CDCL-OP$_{\text{LB}}$, et est sensiblement inférieure à celle de l'implémentation originale de GLUCOSE. Cette forte réduction de la taille des résolutions permet à CDCL-OP-Surveillance d'obtenir un temps d'exécution total sensiblement inférieur à celui de CDCL-OP sur les instances que toutes deux parviennent à résoudre, malgré le handicap conséquent de sa fréquence de traitement fortement inférieure.

CDCL-OP-Dépendances, quant à elle, d'après les tableaux 7.4 et 7.5, bénéficie de la même réduction de la taille des résolutions que CDCL-OP-Surveillance par rapport à CDCL-OP et CDCL-OP$_{\text{LB}}$, qui est certainement due de la même façon à l'absence de conflits redondants. Ces mêmes tableaux indiquent par ailleurs que les performances de CDCL-OP-Dépendances et CDCL-OP-Surveillance en nombre d'étapes de propagation sont très proches sur les instances résolues par les deux implémentations, cela malgré une distance moyenne entre conflits supérieure dans le cas de CDCL-OP-Dépendances, qui reste tout de même inférieure à celles de CDCL-OP et CDCL-OP$_{\text{LB}}$.

CDCL-OP-Dépendances a cependant sur CDCL-OP-Surveillance l'avantage d'une fréquence de traitement supérieure d'environ 18%. Cette différence peut s'expliquer par une quantité de travail supplémentaire lors des propagations unitaires sensiblement inférieure. En effet, lorsqu'une clause vérifiée est surveillée par un littéral faux et un autre vrai, CDCL-OP-Dépendances ajoute dans tous les cas une dépendance entre les niveaux de décision des deux littéraux. Dans la même situation, CDCL-OP-Surveillance doit d'abord chercher à remplacer le littéral faux par un littéral vrai ou indéfini si possible, puis, si le littéral faux ne peut être remplacé, elle rajoute finalement aussi une dépendance entre les niveaux des littéraux.

Cette fréquence de traitement supérieure permet donc à CDCL-OP-Dépendances de meilleures performances en nombre d'instances résolues et temps, puisqu'elle résout 5 instances supplémentaires par rapport à CDCL-OP-Surveillance tout en réduisant le temps total de calcul de plus de deux heures. Par conséquent, les performances en temps et en nombre d'instances surveillées sont aussi supérieures à celles de CDCL-OP, malgré une fréquence de traitement plus basse d'environ 30%. Les performances de CDCL-OP-Dépendances sans littéraux surveillés sont plutôt comparables à celles de CDCL-OP avec littéraux surveillés, puisque les deux implémentations résolvent le même nombre d'instances et nécessitent quasiment le même temps de calcul total, à 11 minutes près. Cette comparaison prouve l'importance de l'amélioration des performances due à l'élimination des conflits redondants, malgré leur fréquence relativement basse dans le CDCL à ordre partiel.

Enfin, la gestion des littéraux bloqués dans CDCL-OP-Dépendances$_{LB}$ permet d'augmenter significativement la fréquence moyenne de vérification des clauses par rapport à CDCL-OP-Dépendances, qui représente près de 85% de la fréquence de traitement de CDCL-OP$_{LB}$ et plus de 60% de la fréquence de traitement de CDCL$_{LB}$, l'implémentation originale de GLUCOSE. Si, comme pour le CDCL classique, l'introduction des littéraux bloqués semble augmenter légèrement la taille des résolutions (voir tableaux 7.4 et 7.5), cette augmentation est toutefois négligeable en comparaison de l'augmentation de la fréquence de traitement.

Par conséquent, CDCL-OP-Dépendances$_{LB}$ réussit à résoudre 9 instances de plus que CDCL-OP-Dépendances tout en réduisant le temps total d'exécution de 20%. CDCL-OP-Dépendances$_{LB}$ est donc selon ces critères de loin l'implémentation du CDCL à ordre partiel la plus efficace, et elle surpasse également CDCL-SSA$_{LB}$, l'implémentation la plus efficace du CDCL sans saut arrière (très largement en nombre d'instances résolues, moins largement en temps d'exécution, puisqu'elle ne réduit ce temps que d'un peu moins de 4%). De même, CDCL-OP-Dépendances$_{LB}$ résout seulement une instance de moins que CDCL$_{LB}$, mais augmente son temps d'exécution de 81%. Le temps d'exécution de CDCL-OP-Dépendances$_{LB}$ est également supérieur d'environ 50% à celui de l'implémentation de CDCL sans littéraux bloqués, mais elle parvient toutefois à résoudre 3 instances de plus. Au final, CDCL-OP-Dépendances$_{LB}$ est en pratique la plus efficace des implémentations que nous avons conçues. Ses performances en temps et en nombre d'instances résolues en un temps donné restent cependant inférieures à celles de l'instance originale de GLUCOSE.

Pour confirmer cette différence de performance entre CDCL$_{LB}$ et CDCL-OP-Dépendances$_{LB}$, nous avons comparé ces deux implémentations sur l'ensemble des 300 instances applicatives de la compétition SAT 2011. Les résultats de ces tests sont présentés par le tableau 7.8. Les performances de CDCL$_{LB}$ et CDCL-OP-Dépendances$_{LB}$ y sont fournies par les colonnes *CDCL* et *Chron* respectivement. Pour vérifier l'impact de la présence ou non de la sauvegarde de phase sur les performances, nous avons également considéré une variante de CDCL$_{LB}$ où elle est désactivée, nommée *CDCL-SSP* (CDCL sans sauvegarde de phase) dans le tableau.

Nos évaluations sur ce plus grand ensemble d'instances confirme les meilleurs résultats de CDCL$_{LB}$ par rapport à CDCL-OP-Dépendances$_{LB}$ aussi bien en matière de nombre d'instances résolues que de temps total d'exécution, quoique l'écart entre les deux implémentations s'accentue dans le premier cas mais diminue dans le second par rapport à notre échantillon restreint de 62 instances. En effet, CDCL-OP-Dépendances$_{LB}$ résout environ 17,5% moins d'instances que CDCL$_{LB}$ dans le temps imparti tout en augmentant le temps total d'exécution de 26%.

TABLEAU 7.8: Comparaison des performances en temps et en nombre d'étapes de propagation de différentes implémentations sur l'ensemble des 300 instances applicatives de la compétition SAT 2011. Les implémentations comparées sont l'implémentation originale de GLUCOSE (CDCL), une variante de cette implémentation sans sauvegarde de phase (CDCL-SSP) ainsi que CDCL-OP-Dépendances$_{LB}$ muni de différentes heuristiques pour le choix du niveau d'assertion : l'heuristique chronologique par défaut (Chron) ainsi que les heuristiques MinDép et MaxDép. Les littéraux bloqués sont activés dans toutes ces implémentations. La première moitié du tableau compare les performances des implémentations en terme de *temps* d'exécution : la colonne *rés.* indique pour chacune le nombre total d'instances résolues à l'intérieur d'une limite d'une heure et la colonne *tot.* fournit le temps total d'exécution (en jours, heures et minutes) sur toutes les instances, résolues ou non. La seconde moitié du tableau compare les performances des implémentations en terme de nombre d'*étapes de propagation* : la colonne *rés.* indique pour chacune le nombre total d'instances résolues à l'intérieur d'une limite de 100 milliards d'étapes et la colonne *tot.* fournit le nombre total d'étapes de propagation (en milliards) sur toutes les instances, résolues ou non.

	temps		*étapes de propagation*	
	rés	*tot. (j,h,m)*	*rés*	*tot. (G)*
CDCL	189	5j15h47m	197	12 869
CDCL-SSP	178	6j02h05m	187	13 911
Chron	156	7j03h23m	173	15 200
MinDép	137	7j08h36m	149	16 613
MaxDép	160	6j18h38m	180	14 441

De plus, cet avantage de CDCL$_{LB}$ n'est pas uniquement dû à sa meilleure vitesse de traitement ; en effet, si l'on limite l'exécution des deux implémentations non pas par un temps limite mais par un nombre maximum d'étapes de propagation (en l'occurrence 100 milliards), CDCL-OP-Dépendances$_{LB}$ résout toujours moins d'instances que CDCL$_{LB}$, bien que la proportion baisse à 12%, et elle augmente le nombre total d'étapes d'exécution nécessaire de 18%. Cela confirme qu'au-delà du surcoût de traitement imputable à la gestion de l'ordre partiel, le CDCL à ordre partiel semble aussi ne pas parvenir à réduire la taille des résolutions en règle générale.

La variante de CDCL$_{LB}$ sans sauvegarde de phase permet de constater que la suppression de ce dispositif est bien un inconvénient, puisqu'elle dégrade les performances du CDCL classique, que ce soit en termes de temps d'exécution, de nombre d'étapes de propagation ou du nombre d'instances résolues à l'intérieur d'une limite de temps ou de

taille de résolution donnée. L'effet de la suppression de la sauvegarde de phase n'explique cependant qu'en partie les moins bonnes performances de CDCL-OP-Dépendances$_{LB}$, car cette implémentation reste moins performante que CDCL$_{LB}$ sans sauvegarde de phase, que ce soit en termes de temps d'exécution ou d'étapes de propagation.

En résumé, les trois implémentations de CDCL à ordre partiel qui assurent l'exhaustivité des propagations améliorent substantiellement les performances de l'algorithme en terme de taille de résolution, ce qui est certainement une conséquence de la non-trivialité et la non-redondance de ces variantes. CDCL-OP-Dépendances$_{LB}$, qui permet de plus l'utilisation des littéraux bloqués, est de loin la variante de CDCL à ordre partiel la plus efficace en terme de temps d'exécution. Si cet algorithme ne permet toutefois pas de concurrencer les performances du CDCL classique sur un ensemble typique d'instances difficiles, nous allons toutefois montrer dans la section 7.8 qu'il permet tout de même d'améliorer l'efficacité de la résolution sur certains types d'instances.

7.8 Influence de la densité des dépendances entre niveaux sur les performances du CDCL à ordre partiel

Les résultats expérimentaux présentés dans la section 7.6 et la sous-section 7.7.4 indiquent que les implémentations de nos différentes variantes du CDCL à ordre partiel restent moins efficaces en temps que l'implémentation originale de GLUCOSE sur un sous-ensemble représentatif des différents types d'instances applicatives utilisées lors de la compétition SAT 2011. Ce constat a par ailleurs été confirmé sur l'ensemble de ces instances pour l'implémentation de la variante la plus efficace en pratique, CDCL-OP-Dépendances$_{LB}$. Cela peut toutefois s'expliquer en partie par le fait que, sur une grande partie des instances de cette compétition, les dépendances entre niveaux de décision sont si fréquentes que l'ordre partiel de notre implémentation reste très proche d'un ordre total, ce qui a pour conséquence de restreindre ses possibilités.

Cette ressemblance de l'ordre partiel à un ordre total peut être quantifiée. Soit $\mathcal{R}$ une relation binaire sur un ensemble E. On peut définir la **densité** de $\mathcal{R}$, notée $\delta(\mathcal{R})$,

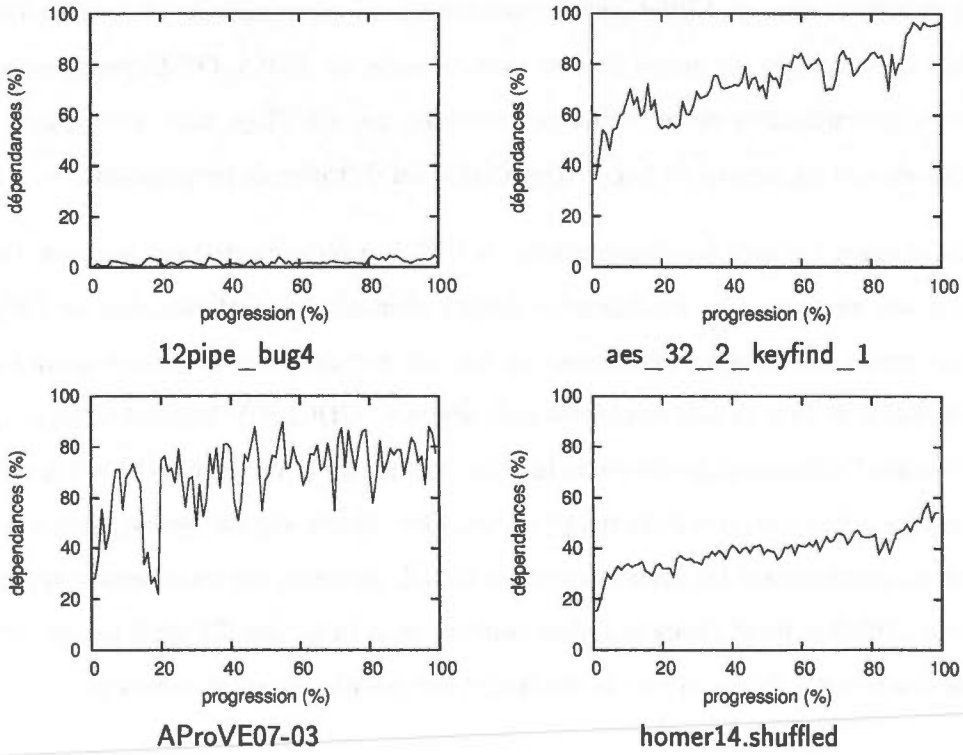


FIGURE 7.2: Évolution de la densité relative de la relation Δ pendant la résolution de différentes instances SAT par l'algorithme CDCL-OP-Dépendances$_{\text{LB}}$. Pour chaque instance, la progression de l'algorithme est représentée en abscisse, en pourcentage du nombre total de conflits, et la densité relative de Δ est indiquée en ordonnée.

comme le nombre total de relations valides entre éléments de E : $\delta(\mathcal{R}) = |\mathcal{R}| = |\{(i,j) \in E^2 \mid i\mathcal{R}j\}|$. La densité maximale d'une relation binaire quelconque sur E est $|E|^2$. Si $\mathcal{R}$ est de plus irréflexive et asymétrique, alors la densité maximale de $\mathcal{R}$ est $\frac{|E|(|E|-1)}{2}$, qui est atteinte si et seulement si $\mathcal{R}$ est un ordre total sur E. Nous noterons $\delta_{\max}(i) = \frac{i(i-1)}{2}$ la densité maximale d'une relation binaire irréflexive et asymétrique sur un ensemble à $i \in \mathbb{N}$ éléments. Alors, pour toute relation $\mathcal{R}$ irréflexive et asymétrique sur un ensemble E, on peut définir la **densité relative** $\delta_{\text{rel}}(\mathcal{R}) = \frac{\delta(\mathcal{R})}{\delta_{\max}(|E|)}$. Quelque soient $\mathcal{R}$ et E, $0 \leq \delta_{\text{rel}}(\mathcal{R}) \leq 1$ et plus $\delta_{\text{rel}}(\mathcal{R})$ est proche de 1, plus $\mathcal{R}$ est similaire à un ordre total sur E.

Dans le cadre du CDCL à ordre partiel, il est évident que si l'ordre partiel Δ^+ a une densité relative proche de 1, alors l'ordre partiel est quasi-identique à un ordre total

TABLEAU 7.9: Comparaison des performances en temps de différentes implémentations sur plusieurs familles d'instances SAT de vérification formelle de microprocesseurs. Les implémentations comparées sont l'implémentation originale de GLUCOSE (CDCL), une variante de cette implémentation sans sauvegarde de phase (CDCL-SSP) ainsi que CDCL_{LB}-OP-Dépendances muni de différentes heuristiques pour le choix du niveau d'assertion : l'heuristique chronologique par défaut (Chron) ainsi que les heuristiques MinDésinst, MaxDésinst, MinDép et MaxDép. Pour chaque *famille*, nous indiquons le nombre d'instances qu'elle contient (*inst*), puis, pour chaque implémentation, le nombre de ces instances résolues en moins d'une heure (*rés*) et le temps total d'exécution (*tot.*, en secondes). Des statistiques sont aussi fournies sur l'ensemble de toutes les instances considérées.

famille	*inst*	CDCL		CDCL-SSP		Chron		MinDésinst		MaxDésinst		MinDép		MaxDép	
		rés	*tot.*	*rés*	*tot.*	*rés*	*tot.*	*rés*	*tot.*	*rés*	*tot.*	*rés*	*tot.*	*rés*	*tot.*
pipe_sat_1.0	10	10	6 955	4	25 364	10	6 601	10	7 334	10	2 042	10	1 264	8	10 399
pipe_sat_1.1	10	10	1 181	9	7 258	9	3 766	9	4 010	10	186	10	185	9	3 820
pipe_unsat_1.0	13	6	25 627	8	23 172	8	19 456	8	19 697	8	19 192	8	20 338	9	17 742
pipe_unsat_1.1	14	7	28 460	9	20 706	8	23 591	8	23 130	8	22 837	8	23 149	8	22 198
pipe_ooo_unsat_1.0	9	8	6 989	7	10 757	7	11 670	6	13 321	7	10 613	7	10 803	7	10 420
pipe_ooo_unsat_1.1	10	6	16 163	9	5 116	9	5 146	8	8 985	8	8 709	8	9 297	9	4 604
total	66	47	85 375	46	92 463	51	70 231	49	76 478	51	63 580	51	65 036	50	69 182

TABLEAU 7.10: Comparaison des performances en nombre d'étapes de propagation de différentes implémentations sur plusieurs familles d'instances SAT de vérification formelle de microprocesseurs. Les implémentations comparées sont les mêmes que dans le tableau 7.9. Pour chaque *famille*, nous indiquons le nombre d'instances qu'elle contient (*inst*), puis, pour chaque implémentation, le nombre de ces instances résolues en moins d'une heure (*rés*) et le nombre total d'étapes de propagation (*tot.*, en millions). Des statistiques sont aussi fournies sur l'ensemble de toutes les instances considérées.

famille	*inst*	CDCL		CDCL-SSP		Chron		MinDésinst		MaxDésinst		MinDép		MaxDép	
		rés	*tot.*	*rés*	*tot.*	*rés*	*tot.*	*rés*	*tot.*	*rés*	*tot.*	*rés*	*tot.*	*rés*	*tot.*
pipe_sat_1.0	10	10	174 962	4	421 799	10	74 034	10	104 876	10	23 970	10	12 817	8	139 328
pipe_sat_1.1	10	10	38 492	9	236 908	9	47 891	9	50 888	10	1 361	10	1 123	9	44 745
pipe_unsat_1.0	13	6	456 250	8	363 586	8	269 638	8	275 440	8	226 145	8	269 658	9	223 828
pipe_unsat_1.1	14	7	545 812	9	329 251	8	302 579	8	281 740	8	266 754	8	264 576	8	217 171
pipe_ooo_unsat_1.0	9	8	214 550	7	224 510	7	193 006	6	221 104	7	177 716	7	176 938	7	169 463
pipe_ooo_unsat_1.1	10	6	524 960	9	232 491	9	125 247	8	131 227	8	124 341	8	133 570	9	114 425
total	66	47	1 955 025	46	1 808 546	51	1 012 395	49	1 065 276	51	820 287	51	858 684	50	908 959

conventionnel et l'exécution de l'algorithme équivaut alors à celle d'un CDCL classique. Ce cas de figure est très défavorable à l'algorithme, puisqu'il conserve les inconvénients de la gestion de l'ordre partiel, qui ralentit l'exécution, sans pouvoir en retirer aucun bénéfice, puisqu'au final le saut arrière partiel se comporte de façon identique à un saut arrière conventionnel. Or, en pratique, les ordres partiels à haute densité relative sont très fréquents sur les instances que nous avons considérées.

La figure 7.2 illustre l'évolution de la densité relative de la relation Δ, qui est une borne inférieure sur la densité relative de l'ordre partiel Δ^+, sur différentes instances applicatives de la compétition SAT 2011. On observe facilement que, dans de nombreux cas, Δ atteint très vite une densité relative importante, qui a donc comme résultat probable d'empêcher une exécution significativement différente de celle d'un CDCL classique. Toutefois, certaines instances, comme **12pipe_bug4**, qui est également représentée dans cette figure, conservent au contraire une relation Δ de très basse densité relative, ce qui permet alors une exécution du CDCL à ordre partiel radicalement différente de celle d'un CDCL classique, et donc de potentiels gains de performances.

En fait, cette instance appartient à un groupe de familles d'instances SAT générées par la vérification formelle de microprocesseurs superscalaires (Velev et Bryant, 2003), qui ont pour point commun cette basse densité relative des dépendances entre niveaux. Cette forte indépendance entre les niveaux de décision au cours de la résolution des instances pourrait provenir directement de la nature des problèmes encodés, puisque les microprocesseurs superscalaires ont pour caractéristique d'être hautement parallélisés. De plus, le CDCL à ordre partiel parvient effectivement à améliorer les performances de résolution du CDCL classique sur ces instances.

Les tableaux 7.9 à 7.17 et la figure 7.3 fournissent différentes statistiques sur la résolution des 6 familles de ces instances dont nous disposons (Velev, 2004) par les mêmes implémentations comparées dans le tableau 7.8 (CDCL_{LB} avec et sans sauvegarde de phase, ainsi que $\text{CDCL-OP-Dépendances}_{\text{LB}}$) :[3]

3. Les tableaux 7.15 à 7.17 sont situés dans la sous-section 7.9.2 car ils ne sont pas utilisés dans

TABLEAU 7.11: Comparaison de différentes implémentations de CDCL et CDCL-OP-Dépendances sur le nombre d'instances de vérification formelle de microprocesseurs qu'elles résolvent en moins d'étapes de propagation. Pour chaque paire d'implémentations, le tableau indique le nombre total d'instances que toutes deux résolvent dans le temps imparti, puis, parmi celles-ci, le nombre d'instances résolues avec moins d'étapes de propagation par l'implémentation en ordonnée et l'implémentation en abscisse respectivement. Lorsque la même implémentation est à la fois en abscisse et en ordonnée, nous indiquons uniquement le nombre d'instances qu'elle résout dans le temps imparti. Les implémentations comparées et les instances utilisées sont les mêmes que dans le tableau 7.9.

	CDCL	CDCL-SSP	Chron	MinDésinst	MaxDésinst	MinDép	MaxDép
CDCL	47	39 (14/25)	45 (5/40)	44 (9/35)	46 (7/39)	46 (4/42)	43 (5/38)
CDCL-SSP	39 (25/14)	46	44 (18/26)	42 (13/29)	44 (10/34)	44 (14/30)	44 (10/34)
Chron	45 (40/5)	44 (26/18)	51	49 (24/25)	50 (13/37)	50 (20/30)	49 (12/37)
MinDésinst	44 (35/9)	42 (29/13)	49 (25/24)	49	49 (18/31)	49 (20/29)	47 (13/34)
MaxDésinst	46 (39/7)	44 (34/10)	50 (37/13)	49 (31/18)	51	51 (32/19)	48 (23/25)
MinDép	46 (42/4)	44 (30/14)	50 (30/20)	49 (29/20)	51 (19/32)	51	48 (22/26)
MaxDép	43 (38/5)	44 (34/10)	49 (37/12)	47 (34/13)	48 (25/23)	48 (26/22)	50

TABLEAU 7.12: Comparaison de différentes implémentations de CDCL et CDCL-OP-Dépendances sur le nombre total d'étapes de propagation pour les instances de vérification formelle de microprocesseurs qu'elles résolvent en commun. Pour chaque paire d'implémentations, le tableau indique le nombre total d'étapes de propagation nécessitées respectivement par l'implémentation en ordonnée et l'implémentation en abscisse pour la résolution de l'ensemble des instances que les deux implémentations sont capables de résoudre dans le temps imparti. Lorsque la même implémentation est à la fois en abscisse et en ordonnée, nous indiquons uniquement le nombre total d'étapes de propagation sur les instances qu'elle résout dans le temps imparti. Les implémentations comparées et les instances utilisées sont les mêmes que dans le tableau 7.9.

	CDCL	CDCL-SSP	Chron	MinDésinst	MaxDésinst	MinDép	MaxDép
CDCL	533 419	371 212/358 628	495 874/176 848	445 770/176 014	498 944/95 417	498 944/91 733	407 737/111 930
CDCL-SSP	358 628/371 212	738 678	593 057/275 092	487 359/172 001	578 389/140 628	578 389/183 984	593 057/189 234
Chron	176 848/495 874	275 092/593 057	313 888	224 873/263 821	264 402/155 613	264 402/192 208	279 339/191 578
MinDésinst	176 014/445 770	172 001/487 359	263 821/224 873	263 821	263 821/120 497	263 821/152 280	219 590/111 329
MaxDésinst	95 417/498 944	140 628/578 389	155 613/264 402	120 497/263 821	155 847	155 847/192 528	148 912/142 634
MinDép	91 733/498 944	183 984/578 389	192 208/264 402	152 280/263 821	192 528/155 847	192 528	187 506/142 634
MaxDép	111 930/407 737	189 234/593 057	191 578/279 339	111 329/219 590	142 634/148 912	142 634/187 506	234 392

- pipe_sat_1.0 et pipe_sat_1.1 sont deux familles d'instances satisfaisables encodant la vérification formelle de spécifications incorrectes de microprocesseurs superscalaires ;
- pipe_unsat_1.0 et pipe_unsat_1.1 sont deux familles d'instances insatisfaisables encodant la vérification formelle de spécifications correctes de microprocesseurs superscalaires ;
- pipe_unsat_1.0 et pipe_unsat_1.1 sont deux familles d'instances insatisfaisables encodant la vérification formelle de spécifications correctes de microprocesseurs superscalaires qui gèrent la réorganisation des instructions avant exécution (*out-of-order execution*).

Pour ces trois types d'instances, les familles 1.0 et 1.1 contiennent des encodages différents des mêmes problèmes. À titre d'exemple, la densité moyenne de Δ lors de l'exécution des instances de pipe_sat_1.0 et pipe_unsat_1.0 est de 1,5% et 5% respectivement.

En comparant uniquement les performances en temps et en nombre d'instances résolues des deux implémentations de CDCL classique, on peut remarquer que la présence ou non de sauvegarde de phase a une très grande influence sur les performances de l'algorithme, qui dépend fortement de la satisfaisabilité des instances considérées. En effet, l'activation de la sauvegarde améliore spectaculairement la résolution des deux familles d'instances satisfaisables, puisqu'elle permet la résolution de toutes les 20 instances en moins d'une heure chacune, alors que l'implémentation sans sauvegarde de phase n'en résout que 13. Au contraire, la sauvegarde de phase détériore significativement la résolution sur les instances insatisfaisables, puisque seules 27 des 46 instances sont résolues, contre 33 pour l'implémentation sans sauvegarde de phase. Par conséquent, le temps total d'exécution est bien inférieur pour l'implémentation avec sauvegarde de phase sur les instances satisfaisables, mais supérieur sur les instances insatisfaisables. Le même constat peut être fait au niveau du nombre d'étapes de propagation, puisque la présence ou l'absence de sauvegarde de phase n'a pas d'incidence notable sur la fréquence de

---

cette sous-section.

TABLEAU 7.13: Comparaison de la destructivité des conflits de différentes implémentations sur plusieurs familles d'instances SAT de vérification formelle de microprocesseurs. Les implémentations comparées sont les mêmes que dans le tableau 7.9. Pour chaque *famille*, nous indiquons le nombre d'instances qu'elle contient (*inst*), puis, pour chaque implémentation, la moyenne sur les instances de cette famille de la destructivité des conflits, c'est-à-dire de la proportion moyenne de variables défaites lors des conflits. Nous fournissons également pour chaque implémentation une *moyenne* sur l'ensemble de toutes les instances considérées.

famille	*inst*	CDCL	CDCL-SSP	Chron	MinDésinst	MaxDésinst	MinDép	MaxDép
pipe_sat_1.0	10	1,04%	1,78%	1,48%	1,44%	3,12%	3,91%	1,68%
pipe_sat_1.1	10	1,36%	2,17%	2,41%	1,97%	4,71%	6,14%	2,09%
pipe_unsat_1.0	13	3,59%	3,67%	4,03%	3,79%	4,65%	4,82%	3,53%
pipe_unsat_1.1	14	3,68%	3,94%	4,13%	4,02%	4,90%	4,93%	3,83%
pipe_ooo_unsat_1.0	9	4,85%	5,32%	5,18%	5,12%	5,58%	5,70%	4,88%
pipe_ooo_unsat_1.1	10	5,16%	4,90%	5,49%	5,24%	5,76%	5,68%	5,23%
moyenne	11	3,30%	3,63%	3,80%	3,61%	4,78%	5,16%	3,54%

TABLEAU 7.14: Comparaison de la fréquence des redémarrages de différentes implémentations sur plusieurs familles d'instances SAT de vérification formelle de microprocesseurs. Les implémentations comparées sont les mêmes que dans le tableau 7.9. Pour chaque *famille*, nous indiquons le nombre d'instances qu'elle contient (*inst*), puis, pour chaque implémentation, le nombre moyen (en millions) d'étapes de propagation entre deux redémarrages consécutifs. Nous fournissons également pour chaque implémentation une *moyenne* sur l'ensemble de toutes les instances considérées.

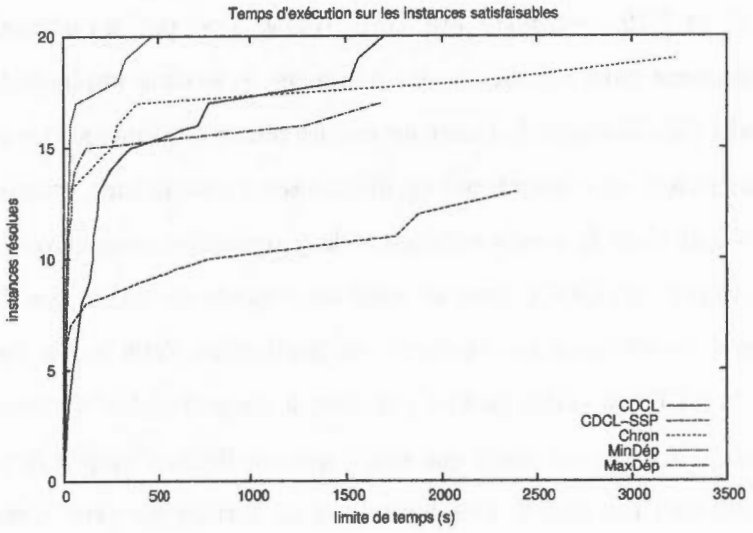
famille	*inst*	CDCL	CDCL-SSP	Chron	MinDésinst	MaxDésinst	MinDép	MaxDép
pipe_sat_1.0	10	276	5 164	18	23	13	9	37
pipe_sat_1.1	10	16	2 999	11	17	6	10	25
pipe_unsat_1.0	13	1 499	1 214	21	28	18	10	39
pipe_unsat_1.1	14	1 271	203	19	21	13	9	19
pipe_ooo_unsat_1.0	9	13	99	7	8	5	4	7
pipe_ooo_unsat_1.1	10	1 283	25	9	9	6	6	8
moyenne	11	805	1 546	14	19	11	8	23

traitement (qui n'est pas indiquée dans ces résultats).

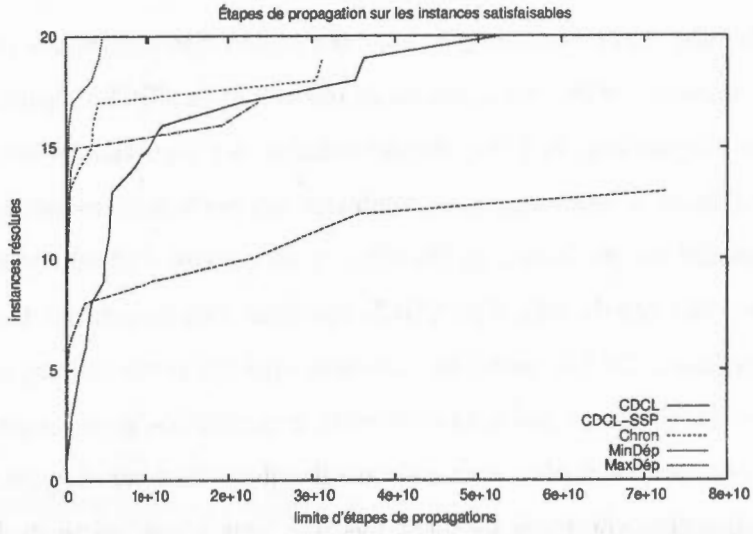
En comparaison, le CDCL à ordre partiel et dépendances additionnelles a un comportement bien plus équilibré, puisqu'il a des performances en temps et en nombre d'instances résolues très proche de la meilleure version de CDCL sur chaque type d'instances. Par conséquent, si l'on considère l'ensemble de toutes les instances, le CDCL à ordre partiel est l'implémentation la plus performante, car elle résout respectivement 4 et 5 instances de plus que le CDCL avec et sans sauvegarde de phase, tout en réduisant le temps total d'exécution de 18% et 24% respectivement.

Les figures 7.3a et 7.4a permettent d'avoir un aperçu plus précis des performances des différentes implémentations sur les instances individuelles, respectivement satisfaisables et insatisfaisables. La figure 7.3a indique que le CDCL à ordre partiel parvient à résoudre la plupart des instances satisfaisables bien plus vite que le CDCL classique avec sauvegarde de phase ; par exemple, 13 des 20 instances sont résolues en moins de 30 secondes. Le CDCL classique reprend toutefois l'avantage sur les 3 instances les plus difficiles, qu'il parvient à résoudre en moins de 30 minutes chacune ; le CDCL à ordre partiel a besoin de plus de temps sur deux d'entre elles et ne parvient pas à résoudre la dernière en moins d'une heure. Ses performances restent toutefois bien supérieures à celles du CDCL sans sauvegarde de phase. Sur les instances insatisfaisables, d'après la figure 7.4a, le CDCL à ordre partiel reste significativement plus performant que le CDCL avec sauvegarde de phase, l'implémentation originale de GLUCOSE, quelque soit la limite de temps considérée. Il est selon les limites de temps considérées légèrement plus ou légèrement moins performant que le CDCL sans sauvegarde de phase, mais parvient finalement à résoudre une instance de moins que celui-ci.

Étant donnée la forte baisse de fréquence de traitement dans le CDCL à ordre partiel, la différence de performances avec les implémentations de CDCL classique est d'autant plus importante si l'on considère la taille totale des résolutions en nombre d'étapes de propagation. Comme l'indique le tableau 7.10, le CDCL à ordre partiel réduit de près de moitié le nombre total d'étapes de propagation par rapport aux deux



(a)



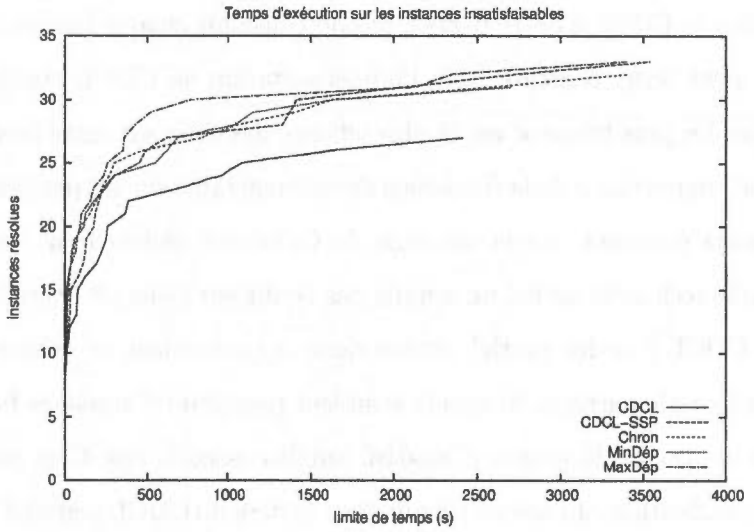
(b)

FIGURE 7.3: Performances comparées de diverses implémentations de CDCL et CDCL à ordre partiel sur les instances satisfaisables de vérification formelle de microprocesseurs (c'est-à-dire l'ensemble des familles pipe_sat_1.0 et pipe_sat_1.1) avec différentes limites de temps et de nombre d'étapes de propagation. La figure 7.3a compare le nombre d'instances résolues par chaque implémentation en une limite de temps donnée, et la figure 7.3b compare le nombre d'instances résolues par chaque implémentation en un nombre limite d'étapes de propagation donné. Pour chaque figure, la limite en temps ou en nombre d'étapes de propagation est indiquée en abscisse et le nombre d'instances résolues en ordonnée. Les implémentations comparées sont les mêmes que dans le tableau 7.8.

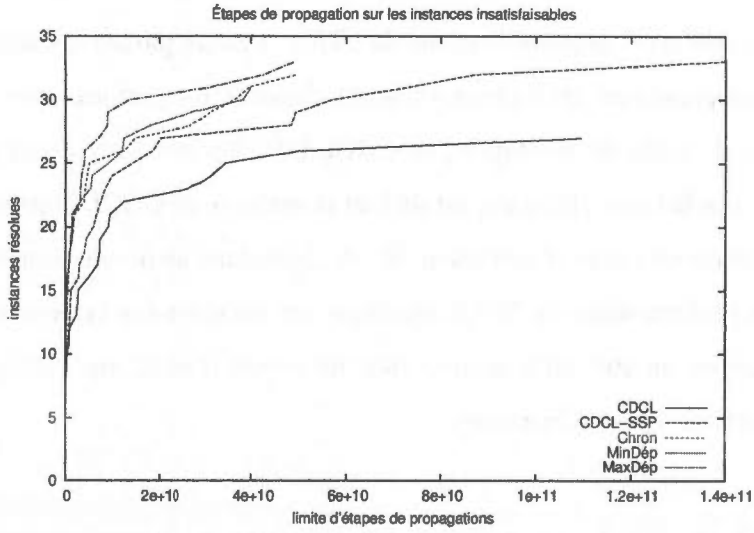
implémentations de CDCL classique, tout en augmentant le nombre d'instances résolues. Les tableaux 7.11 et 7.12 confirment que cette baisse n'est pas simplement due à la fréquence de traitement plus réduite sur les instances qu'aucune implémentation n'est capable de résoudre ; au contraire, la baisse est encore plus importante si l'on ne considère que les instances résolues en commun par les différentes instanciations, puisque le CDCL à ordre partiel réduit alors le nombre d'étapes de propagation respectivement de 64% et de 54% par rapport au CDCL avec et sans sauvegarde de phase. Les figures 7.3b et 7.4b confirment visuellement ces mesures ; en particulier, dans le cas des instances insatisfaisables, le CDCL à ordre partiel parvient à résoudre plus d'instances que le CDCL sans sauvegarde de phase quelle que soit le nombre limite d'étapes de propagation considéré, à l'exception des limites très hautes où ce dernier parvient à résoudre une instance de plus que le CDCL à ordre partiel dans le temps imparti.

Ces familles d'instances constituent donc une preuve que le CDCL à ordre partiel peut effectivement améliorer les performances de résolution du CDCL classique dans certaines conditions. Cependant, la faible densité relative des dépendances entre niveaux n'est pas une explication suffisante pour expliquer les performances particulières du CDCL à ordre partiel sur ces instances. En effet, si elle permet d'obtenir une exécution significativement différente de celle d'un CDCL classique, cela ne garantit pas que cette exécution sera meilleure. De fait, parmi les instances applicatives de la compétition SAT 2011, il en existe d'autres avec des faibles densités comparables, pour lesquelles toutefois les performances de résolution sont au contraire dégradées par rapport au CDCL classique. L'amélioration constatée ne vient pas non plus d'une baisse de la destructivité des conflits ; au contraire, le tableau 7.13 montre que le CDCL à ordre partiel augmente sensiblement cette destructivité par rapport aux implémentations du CDCL classique. Toutefois, au vu de la faible densité relative sur ces instances, il est tout de même probable que certains sauts arrière conservent localement de nombreux niveaux de décision sans lien avec le conflit à résoudre, ce qui permet d'effectuer une exécution significativement différente, et finalement plus efficace, que le CDCL classique.

En fait, un facteur important dans cette amélioration des performances semble être



(a)



(b)

FIGURE 7.4: Performances comparées de diverses implémentations de CDCL et CDCL à ordre partiel sur les instances insatisfaisables de vérification formelle de microprocesseurs (c'est-à-dire l'ensemble des familles pipe_unsat_1.0, pipe_unsat_1.1, pipe_ooo_unsat_1.0 et pipe_ooo_unsat_1.1) avec différentes limites de temps et de nombre d'étapes de propagation. La figure 7.4a compare le nombre d'instances résolues par chaque implémentation en une limite de temps donnée, et la figure 7.4b compare le nombre d'instances résolues par chaque implémentation en un nombre limite d'étapes de propagation donné. Pour chaque figure, la limite en temps ou en nombre d'étapes de propagation est indiquée en abscisse et le nombre d'instances résolues en ordonnée. Les implémentations comparées sont les mêmes que dans le tableau 7.8.

la fréquence des redémarrages, qui, comme le montre le tableau 7.14, sont en général bien plus fréquents dans le CDCL à ordre partiel. Si l'on considère chaque famille d'instances séparément, on peut aussi constater que l'implémentation de CDCL classique où les redémarrages sont les plus fréquent est la plus efficace des deux sur cette famille, ce qui semble confirmer l'importance de la fréquence de redémarrages sur les performances. Ce paramètre est assez étonnant, car la stratégie de GLUCOSE déclenche un redémarrage lorsque l'état de la recherche actuel ne semble pas se diriger assez vite vers la solution. L'exécution du CDCL à ordre partiel atteint donc apparemment ce critère bien plus fréquemment, et les redémarrages fréquents semblent permettre d'accélérer fortement la découverte d'un modèle ou la preuve d'insatisfaisabilité selon le cas. Cela prouve entre autres que nos modifications du mécanisme de saut arrière du CDCL peuvent aussi avoir des effets positifs sur d'autres composantes de l'algorithme.

En résumé, les trois implémentations de CDCL à ordre partiel qui assurent l'exhaustivité des propagations améliorent substantiellement les performances de l'algorithme en terme de taille de résolution, et CDCL-OP-Dépendances$_{LB}$, qui permet de plus l'utilisation des littéraux bloqués, est de loin la variante de CDCL à ordre partiel la plus efficace en terme de temps d'exécution. Si cet algorithme ne permet toutefois pas de concurrencer les performances du CDCL classique sur un ensemble typique d'instances difficiles, nous avons montré qu'il permet tout de même d'améliorer l'efficacité de la résolution sur certains types d'instances.

7.9 Heuristiques de choix des niveaux d'assertion

Comme nous l'avons vu dans la sous-section 7.1.3, le CDCL à ordre partiel rajoute une liberté de choix supplémentaire par rapport au CDCL classique en introduisant une définition non-unique du niveau d'assertion lors d'un saut arrière partiel. Toutefois, les résultats expérimentaux présentés jusqu'ici n'explorent pas les conséquences de cette liberté, car ils utilisent tous une même stratégie de choix conçue pour rester la plus proche possible de la définition unique dans le CDCL classique (voir section 7.6).

La stratégie de choix des niveaux d'assertion a cependant un impact potentiellement important sur le déroulement global de l'exécution : ainsi, lors de l'exécution de l'algorithme CDCL-OP-Dépendances$_{LB}$ sur l'ensemble des instances du tableau 7.9, l'algorithme peut choisir le niveau d'assertion parmi au moins 2 niveaux différents lors de 31% des conflits et dans le cas d'un tel choix multiple, il existe en moyenne 10 niveaux candidats. Les choix multiples de niveaux d'assertion sont donc clairement assez fréquents et significatifs pour avoir une incidence non-négligeable sur le déroulement de la recherche.

Cette section est donc dédiée à la conception de différentes heuristiques pour le choix des niveaux d'assertion et à l'évaluation expérimentale de leur incidence sur le CDCL à ordre partiel. La sous-section 7.9.1 discute différentes heuristiques de choix de niveaux d'assertion et leur effet recherché sur l'exécution de l'algorithme. La sous-section 7.9.2 évalue expérimentalement l'impact de ces heuristiques sur les performances de l'algorithme CDCL-OP-Dépendances$_{LB}$ et démontre que certaines heuristiques peuvent améliorer significativement les performances de l'algorithme.

7.9.1 Critères de choix des niveaux d'assertion

L'heuristique chronologique proposée dans la section 7.6 se rapproche autant que possible du choix unique du niveau d'assertion dans le cas du CDCL classique, mais l'utilisation de cette heuristique n'est motivée par aucun effet attendu de la pertinence de ce choix. Les heuristiques que nous allons proposer ici ont au contraire comme objectif d'intervenir sur l'aspect qui a principalement motivé la conception du CDCL à ordre partiel, c'est-à-dire la quantité d'instanciations défaites par le saut arrière partiel.

En effet, le choix du niveau d'assertion a un effet direct sur la quantité d'instanciations défaites, puisque le saut arrière partiel défait le niveau de conflit ainsi que tous les niveaux qui dépendent récursivement du niveau d'assertion. On peut donc facilement minimiser localement le nombre d'instanciations défaites par le saut arrière partiel : pour chaque niveau candidat, le nombre de désinstanciations que son choix provoquerait est

égal au total des instanciations situées dans des niveaux qui dépendent récursivement de ce niveau candidat (auquel il faut ajouter le nombre d'instanciations du niveau de conflit, si celui-ci ne dépend pas récursivement du niveau candidat). Il suffit alors de choisir le niveau candidat qui minimise ce nombre de désinstanciations potentielles.

Nous appellerons MinDésinst cette heuristique de choix de niveau d'assertion. Celle-ci garantit de choisir le niveau candidat qui provoquera le moins de désinstanciations possible. Toutefois, elle nécessite de calculer une partie de la fermeture transitive de Δ pour chaque niveau candidat. Dans un but d'efficacité en temps des heuristiques, nous ne considérerons en pratique que les dépendances directes entre niveaux. Notre implémentation de MinDésinst choisit donc le niveau candidat qui minimise le nombre d'instanciations situées dans des niveaux qui dépendent directement de ce niveau. Il s'agit donc d'une approximation de la minimisation de la destructivité du saut arrière, car un niveau candidat non-retenu peut provoquer moins de désinstanciations que le niveau choisi si moins d'instanciations en dépendent indirectement.

On peut remarquer que si l'heuristique (exacte) MinDésinst garantit de choisir le niveau d'assertion qui minimise la destructivité du saut arrière courant, elle néglige toutefois l'impact de ce choix sur les sauts arrière ultérieurs. Cette incidence est considérable, notamment sur l'évolution de la relation de dépendances entre les niveaux Δ. En effet, après le saut arrière partiel, le niveau d'assertion devient le nouveau niveau courant, et d'éventuelles propagations unitaires peuvent provoquer de nouvelles dépendances pour ce niveau. En particulier, la propagation de l'assertion provoque la dépendance du niveau d'assertion à tous les autres niveaux impliqués dans la clause de conflit (hormis l'ancien niveau de conflit qui est défait).

Cet ajout de dépendances supplémentaires au niveau d'assertion a une influence sur la destructivité des sauts arrière partiels à venir, puisque plus un niveau de décision dépend d'autres niveaux, plus il est susceptible d'être défait lors d'un saut arrière partiel, puisqu'il suffit que le niveau d'assertion soit un des niveaux dont il dépend récursivement. De plus, l'augmentation de la densité de Δ implique une restriction des choix de niveau

d'assertion lors des sauts arrière.

Il est donc souhaitable, lors du choix des niveaux d'assertion, de chercher à réduire l'augmentation de la densité de Δ. Cette augmentation dépend toutefois de l'ensemble des propagations déclenchées par la propagation de l'assertion, qui est évidemment très difficile à prévoir. Nous nous restreindrons donc à chercher à réduire l'impact de la propagation de l'assertion uniquement sur la densité de Δ. Soit Θ l'ensemble des niveaux de décision présents dans la clause de conflit, hormis le niveau de conflit, et soit Γ l'ensemble des niveaux d'assertion candidats, soit les éléments maximaux de Θ selon Δ^+. Pour tout niveau candidat $i \in \Gamma$, la propagation de l'assertion provoquerait la dépendance de ce niveau à tous les niveaux dans $\Theta \setminus \{i\}$, ainsi que de potentielles dépendances indirectes supplémentaires. On peut donc choisir le niveau candidat pour lequel la propagation de l'assertion rajoute le moins de dépendances dans Δ^+, c'est-à-dire le candidat $i \in \Gamma$ qui minimise la valeur $|(\Delta \cup \{\{j,i\} \mid j \in \Theta \setminus \{i\}\})^+| - |\Delta^+|$. Nous nommerons cette heuristique MinDép, puisqu'elle cherche à minimiser les nouvelles dépendances.

Comme pour l'heuristique MinDésinst, nous considérerons en pratique uniquement les dépendances directes pour obtenir un calcul de l'heuristique plus efficace. Comme la propagation de l'assertion rajoute potentiellement $|\Theta| - 1$ dépendances directes supplémentaires quelque soit le niveau d'assertion choisi, notre implémentation de MinDép choisit le niveau candidat qui maximise le nombre de ces dépendances directes déjà existantes, c'est-à-dire le niveau candidat $i \in \Gamma$ qui maximise la valeur $|\{(j,i) \mid j \in \Theta \setminus \{i\}\} \cap \Delta|$.

Nous avons également implémenté MaxDésinst et MaxDép, les stratégies opposées à MinDésinst et MinDép, c'est-à-dire qui choisissent le niveau candidat qui maximise le nombre d'instanciations défaites par le saut arrière partiel ou le nombre de nouvelles dépendances dues à la propagation de l'assertion, respectivement. Bien que ces heuristiques soient à l'opposé de notre but de réduction de la destructivité des sauts arrière, nous verrons qu'elles peuvent parfois également améliorer les performances du CDCL à

ordre partiel.

7.9.2 Résultats expérimentaux

Les quatre heuristiques pour le choix du niveau d'assertion décrites dans la sous-section précédente, MinDésinst, MaxDésinst, MinDép et MaxDép, ont été testées au sein de l'algorithme CDCL-OP-Dépendances$_{LB}$ et comparées à ce même algorithme utilisant l'heuristique chronologique par défaut ainsi qu'au CDCL classique avec et sans sauvegarde de phase sur les instances de vérification formelle de microprocesseurs utilisées précédemment dans la sous-section 7.7.4. Les résultats obtenus sont présentés par les tableaux 7.9 à 7.17. La figure 7.3 ne présente que les données des heuristiques MinDép et MaxDép, dans un soucis de clarté.

Comme l'indique le tableau 7.13, les heuristiques MinDésinst et MaxDésinst ont bien comme effet de respectivement diminuer et augmenter la destructivité moyenne des conflits par rapport à l'heuristique chronologique. Cependant, on peut remarquer que la baisse de destructivité occasionnée par MinDésinst est dans la plupart des cas très minime, ce qui semble indiquer que la stratégie chronologique du choix de niveau d'assertion choisit souvent le niveau d'assertion qui minimise la quantité d'instanciations défaites. C'est assez logique, puisqu'avant le premier conflit aucun niveau ne dépend d'un autre niveau chronologiquement postérieur. Parmi les différents niveaux d'assertion candidat, celui qui provoque le moins de désinstanciations est alors forcément le plus récent. Le choix systématique du candidat le plus récent doit tendre à maintenir partiellement cette propriété.

Cette faible différence semble empêcher l'heuristique MinDésinst d'améliorer les performances par rapport à l'heuristique par défaut ; au contraire, elle augmente sensiblement le temps total de résolution pour la plupart des familles et parvient même à résoudre deux instances insatisfaisables de moins dans le temps imparti.

De façon surprenante, l'heuristique MaxDésinst est bien plus intéressante. En effet, elle provoque une nette augmentation de la destructivité des conflits par rapport

à l'heuristique chronologique, particulièrement sur les instances satisfaisables où cette destructivité est plus que doublée. Étonnamment, cette forte augmentation de la destructivité des conflits a un impact très bénéfique sur les performances de résolution sur ces instances insatisfaisables : l'heuristique MaxDésinst permet en effet de toutes les solutionner en moins d'une heure, comme l'implémentation originale de GLUCOSE, alors que l'heuristique chronologique échoue à résoudre une des instances. De plus MaxDésinst améliore fortement les performances de résolution par rapport à l'implémentation originale de GLUCOSE, puisqu'elle résout les 20 instances en plus de 3,5 fois moins de temps et plus de 8 fois moins d'étapes de propagation.

Cette amélioration des performances va à l'encontre de notre intuition selon laquelle la destructivité des conflits est nocive aux performances de résolution et doit être réduite. Elle rejoint cependant le raisonnement des méthodes à saut arrière illimité (voir section 3.5) qui supposent que le modèle d'une instance satisfaisable peut être trouvé plus efficacement si l'on peut se déplacer rapidement à l'intérieur de l'espace de recherche. C'est bien ce qui semble se passer dans notre cas : en doublant la quantité moyenne d'instanciations défaites par conflit, l'heuristique MaxDésinst semble permettre à la recherche d'échapper plus facilement à des parties de l'espace de recherche qui ne contiennent aucune solution, et donc de trouver un modèle plus rapidement.

Ce raisonnement ne tient toutefois pas dans le cas d'instances insatisfaisables ; par conséquent, les performances de MaxDésinst sur ces instances sont sensiblement inférieures à celles de l'heuristique chronologique (tout en restant curieusement supérieures à celles de MinDésinst), notamment car elle résout une instance de moins. La différence est tout de même mineure, car l'augmentation de la destructivité des conflits est bien moindre que dans le cas des instances satisfaisables.

Comme nous l'avions suspecté, les heuristiques MinDésinst et MaxDésinst sont trop locales pour totalement optimiser la destructivité moyenne des conflits dans un sens ou dans l'autre, et les heuristiques raisonnant à partir des dépendances supplémentaires provoquées par la propagation des clauses apprises parviennent dans la plupart

TABLEAU 7.15: Comparaison de différentes implémentations de CDCL et CDCL-OP-Dépendances sur le nombre d'instances insatisfaisables de vérification formelle de microprocesseurs qu'elles résolvent avec une plus petite destructivité des conflits. Pour chaque paire d'implémentations, le tableau indique le nombre total d'instances que toutes deux résolvent dans le temps imparti, puis, parmi celles-ci, le nombre d'instances résolues avec moins d'étapes de propagation par l'implémentation en ordonnée et l'implémentation en abscisse respectivement. Lorsque la même implémentation est à la fois en abscisse et en ordonnée, nous indiquons uniquement le nombre d'instances qu'elle résout dans le temps imparti. Les implémentations comparées sont les même que dans le tableau 7.9 ; les instances utilisées sont les instances insatisfaisables de ce même tableau, soit les familles pipe_unsat_1.0, pipe_unsat_1.1, pipe_ooo_unsat_1.0 et pipe_ooo_unsat_1.1.

	CDCL	CDCL-SSP	Chron	MinDésinst	MaxDésinst	MinDép	MaxDép
CDCL	27	26 (9/17)	26 (15/11)	25 (11/14)	26 (23/3)	26 (24/2)	26 (8/18)
CDCL-SSP	26 (17/9)	33	32 (24/8)	30 (18/12)	31 (29/2)	31 (29/2)	32 (13/19)
Chron	26 (11/15)	32 (8/24)	32	30 (4/26)	31 (30/1)	31 (27/4)	32 (3/29)
MinDésinst	25 (14/11)	30 (12/18)	30 (26/4)	30	30 (30/0)	30 (30/0)	30 (7/23)
MaxDésinst	26 (3/23)	31 (2/29)	31 (1/30)	30 (0/30)	31	31 (15/16)	31 (1/30)
MinDép	26 (2/24)	31 (2/29)	31 (4/27)	30 (0/30)	31 (16/15)	31	31 (1/30)
MaxDép	26 (18/8)	32 (19/13)	32 (29/3)	30 (23/7)	31 (30/1)	31 (30/1)	33

TABLEAU 7.16: Comparaison de différentes implémentations de CDCL et CDCL-OP-Dépendances sur la destructivité moyenne des conflits sur les instances insatisfaisables de vérification formelle de microprocesseurs qu'elles résolvent en commun. Pour chaque paire d'implémentations, le tableau indique la moyenne de la destructivité des conflits respectivement dans l'implémentation en ordonnée et l'implémentation en abscisse pour la résolution de l'ensemble des instances que les deux implémentations sont capables de résoudre dans le temps imparti. Lorsque la même implémentation est à la fois en abscisse et en ordonnée, nous indiquons uniquement la destructivité moyenne des conflits sur les instances qu'elle résout dans le temps imparti. Les implémentations comparées et les instances utilisées sont les mêmes que dans le tableau 7.15.

	CDCL	CDCL-SSP	Chron	MinDésinst	MaxDésinst	MinDép	MaxDép
CDCL	5,96%	6,07%/5,72%	6,07%/6,31%	6,19%/6,19%	6,07%/7,15%	6,07%/7,28%	6,07%/5,83%
CDCL-SSP	5,72%/6,07%	5,17%	5,26%/5,76%	5,41%/5,7%	5,33%/6,59%	5,33%/6,71%	5,26%/5,33%
Chron	6,31%/6,07%	5,76%/5,26%	5,76%	5,95%/5,7%	5,85%/6,59%	5,85%/6,71%	5,76%/5,33%
MinDésinst	6,19%/6,19%	5,7%/5,41%	5,7%/5,95%	5,7%	5,7%/6,71%	5,7%/6,83%	5,7%/5,5%
MaxDésinst	7,15%/6,07%	6,59%/5,33%	6,59%/5,85%	6,71%/5,7%	6,59%	6,59%/6,71%	6,59%/5,4%
MinDép	7,28%/6,07%	6,71%/5,33%	6,71%/5,85%	6,83%/5,7%	6,71%/6,59%	6,71%	6,71%/5,4%
MaxDép	5,83%/6,07%	5,33%/5,26%	5,33%/5,76%	5,5%/5,7%	5,4%/6,59%	5,4%/6,71%	5,24%

des cas à surpasser les valeurs atteintes par MinDésinst et MaxDésinst. Cependant, ces heuristiques ont l'effet inverse à celui que nous attendions ; ainsi, MinDép provoque des taux de destructivité particulièrement élevés, tandis que MaxDép les réduit. Ainsi, en choisissant les niveaux d'assertion qui provoquent le moins d'ajout de dépendances supplémentaires, on augmente globalement la destructivité des conflits, et vice versa. Ce comportement à l'opposé de notre intuition s'explique peut-être par le fait que nous considérons seulement les dépendances directes ; l'effet de nos choix pourrait être différent sur leur fermeture transitive. Il est aussi possible que des mécanismes plus complexes que ceux auxquels nous pensions provoquent cette inversion de l'effet recherché.

MinDép a donc pour effet d'augmenter la destructivité des conflits par rapport à MaxDésinst. Si cette augmentation est minime dans le cas des instances insatisfaisables, ce qui produit des exécutions très proches pour les deux heuristiques, elle est conséquente dans le cas des instances satisfaisables : le taux de destructivité y représente plus du triple du taux obtenu avec l'heuristique chronologique par défaut. Cette augmentation conduit à une amélioration supplémentaire des performances par rapport à MaxDésinst : le temps total d'exécution et le nombre total d'étapes de propagation sont réduits de 35% et 45% respectivement. MaxDép résout donc ces 20 instances 5,5 fois plus rapidement et en 15 fois moins d'étapes de propagation que l'implémentation originale de GLUCOSE. La figure 7.3a montre que MaxDép résout 17 des 20 instances en moins de 70 secondes chacune, les 3 instances restantes nécessitant au plus 500 secondes. En comparaison, GLUCOSE requiert plus de 100 secondes pour 11 des instances et plus de 1 500 secondes pour les trois plus difficiles. La figure 7.3b est encore plus éloquente, puisqu'elle montre que MaxDép résout toutes les instances en moins de 4,5 milliards d'étapes de propagation, alors que seules 8 instances sont résolues par GLUCOSE à l'intérieur de cette limite.

MaxDép, quant à elle, augmente légèrement la destructivité moyenne des conflits par rapport à MinDésinst sur les instances satisfaisables, mais la réduit notablement sur la plupart des familles insatisfaisables. Le tableau 7.15 montre notamment que MaxDép réduit la destructivité par rapport à MinDésinst sur les trois quarts de leurs instances

TABLEAU 7.17: Comparaison de la distance moyenne des conflits pour différentes implémentations sur plusieurs familles d'instances insatisfaisables de vérification formelle de microprocesseurs. Les implémentations comparées sont les mêmes que dans le tableau 7.9. Pour chaque *famille*, nous indiquons le nombre d'instances qu'elle contient (*inst*), puis, pour chaque implémentation, la moyenne sur les instances de cette famille de la distance moyenne entre les conflits consécutifs, en nombre d'étapes de propagation. Nous fournissons également pour chaque implémentation une *moyenne* sur l'ensemble de toutes les instances considérées.

famille	*inst*	CDCL	CDCL-SSP	Chron	MinDésinst	MaxDésinst	MinDép	MaxDép
pipe_unsat_1.0	13	5 663	5 407	5 801	5 901	6 273	6 240	4 976
pipe_unsat_1.1	14	5 650	4 743	4 994	4 713	5 160	5 274	3 796
pipe_ooo_unsat_1.0	9	3 781	4 245	3 646	3 909	3 766	3 766	3 443
pipe_ooo_unsat_1.1	10	4 220	3 534	3 355	3 252	3 367	3 488	3 048
moyenne	11	4 977	4 570	4 602	4 574	4 812	4 864	3 898

résolues en commun, et par rapport aux autres heuristiques sur la quasi-totalité des instances. Cette réduction semble avoir un effet bénéfique sur les performances de résolution, puisque d'après le tableau 7.9, MaxDép est l'implémentation qui nécessite le moins de temps de calcul sur l'ensemble des instances insatisfaisables, tout en résolvant autant d'instances que le CDCL sans sauvegarde de phase. Cette performance est également constatée sur le plan de la taille de résolution par le tableau 7.12, qui montre que MaxDép nécessite au total moins d'étapes de propagation que toute autre implémentation sur leurs instances résolues en commun, malgré ses mauvaises performances sur les instances satisfaisables (voir tableau 7.10). Les figures 7.4a et 7.4b montrent en outre que MaxDép parvient à résoudre autant ou plus d'instances insatisfaisables que le CDCL sans sauvegarde de phase, quelle que soit la limite de temps ou d'étapes de propagation considérée. Cette supériorité est bien plus marquée dans le cas de la taille des résolutions.

Ces résultats de MaxDép semblent confirmer que notre intuition du bénéfice d'une destructivité des sauts arrière réduite, bien qu'invalidée dans le cas des instances satisfaisables, reste pertinente sur les instances insatisfaisables. De plus, le tableau 7.17 indique que MaxDép réduit de façon substantielle la distance moyenne entre les conflits par rapport à toutes les autres instanciations. Cela semble donc bien indiquer que la plus petite destructivité des conflits améliore l'efficacité de la résolution en permettant d'atteindre plus rapidement de nouveaux conflits. Comme l'insatisfaisabilité d'une instance ne peut être prouvée qu'en élaguant totalement son espace de recherche et que cet élagage est effectué par les conflits, il est logique qu'une fréquence plus élevée des conflits permette une résolution plus rapide des instances insatisfaisables. Dans le cas des instances satisfaisables, cet élagage peut être utile pour s'orienter vers un modèle mais n'est pas indispensable, ce qui explique qu'une plus grande mobilité dans l'espace de recherche est alors préférable à une plus grande fréquence des conflits.

Ces résultats prouvent clairement que l'heuristique de choix du niveau d'assertion a une influence importante sur l'efficacité du CDCL à ordre partiel et que des heuristiques bien choisies peuvent permettre d'améliorer considérablement ces résultats. Toutefois,

il n'est pas clair que ces heuristiques aient une grande efficacité sur les instances où le CDCL à ordre partiel muni de l'heuristique chronologique par défaut dégrade les performances de résolution par rapport au CDCL classique. Par exemple, le tableau 7.8 montre que, sur l'ensemble des 300 instances applicatives de la compétition SAT, l'heuristique MinDép dégrade fortement les performances du CDCL à ordre partiel par rapport à l'heuristique chronologique, tandis que MaxDép les améliore sensiblement, sans toutefois atteindre celles d'aucune des deux variantes du CDCL classique.

Un enseignement très intéressant est que les heuristiques de choix des niveaux d'assertion permettent de considérablement moduler la destructivité des sauts arrière. De plus, le but à suivre concernant cette destructivité dépend fortement de la satisfaisabilité des instances vérifiées : si la résolution des instances insatisfaisables bénéficie d'une destructivité moyenne plus basse, qui permet des conflits, et donc des élagages de l'espace de recherche, plus fréquents, les instances satisfaisables sont au contraire résolues plus efficacement avec une destructivité des conflits élevée, qui permet plus de mobilité dans l'espace de recherche et donc une découverte plus rapide d'un modèle.

7.10 Conclusion

Ce chapitre a présenté l'algorithme du CDCL à ordre partiel, conçu comme un compromis entre le CDCL classique et le CDCL sans saut arrière. En effet, le CDCL à ordre partiel réduit la destructivité du saut arrière par rapport au CDCL classique, sans toutefois totalement l'éliminer comme le CDCL sans saut arrière. Pour cela, le CDCL à ordre partiel maintient un ordre partiel des dépendances entre les différents niveaux de décision, qui lui permettent lors des sauts arrière de ne pas défaire les niveaux de décision qui ne dépendent pas du niveau d'assertion. De plus, cet ordre partiel modifie la notion de niveau d'assertion, qui n'est plus défini uniquement comme dans le CDCL classique et le CDCL sans saut arrière. Cette non-unicité du niveau d'assertion introduit une nouvelle dimension de choix dans l'algorithme, rendant possible l'utilisation d'heuristiques qui peuvent influer grandement sur le déroulement et les performances de l'algorithme.

Le CDCL à ordre partiel présente deux avantages très importants en pratique par rapport au CDCL sans saut arrière. Le plus important est qu'en n'éliminant pas totalement l'étape de saut arrière, le CDCL à ordre partiel, contrairement au CDCL sans saut arrière, conserve l'unicité du niveau de propagation présente dans le CDCL classique ; cette propriété permet à certaines variantes du CDCL à ordre partiel de garantir la non-trivialité et la non-redondance de tous les conflits, ce qui permet un meilleur élagage de l'espace de recherche.

Le second avantage du CDCL à ordre partiel est qu'il manipule les dépendances à un niveau de granularité plus élevé, c'est-à-dire entre niveaux de décision plutôt qu'entre instanciations. Si cette granularité implique une analyse moins fine des dépendances, et donc une réduction moins importante de la destructivité des sauts arrière, elle permet aussi une gestion bien moins compliquée de ces dépendances ; en particulier, elle évite la gestion très lourde des propagations alternatives, qui est indispensable à l'exhaustivité des propagations dans le CDCL sans saut arrière.

Grâce à ces deux différences principales, CDCL-OP-Dépendances$_{LB}$, la variante la plus efficace du CDCL à ordre partiel, parvient à être significativement plus performante que toutes les variantes du CDCL sans saut arrière que nous avons développées au chapitre précédent. Cependant, dans le cas général, le CDCL à ordre partiel reste moins performant que le CDCL classique, pour des raisons difficiles à déterminer mais sûrement liées aux interactions du saut arrière partiel avec les autres aspects de l'algorithme, notamment l'heuristique de décision et le lien entre sauvegarde de phase et redémarrage, sans oublier le surcoût de gestion des dépendances qui reste considérable.

Malgré ces résultats mitigés dans le cas général, le CDCL à ordre partiel permet d'améliorer notablement les performances de résolution sur certaines familles d'instances SAT, qui présentent la particularité de produire des relations entre niveaux de très faible densité tout au long de l'exécution. De plus, l'utilisation de certaines heuristiques pour le choix de niveau d'assertion améliore encore plus les performances. Étrangement, ces heuristiques amélioreront la résolution de certaines instances en augmentant la destructivité

moyenne des sauts arrière, ce qui va à l'encontre de notre intuition de saut arrière partiel mais semble permettre d'explorer plus efficacement l'espace de recherche. D'autres instances, au contraire, sont résolues plus efficacement lorsque la destructivité moyenne des conflits est réduite, ce qui permet alors comme nous le prévoyions d'améliorer l'élagage de l'espace de recherche en augmentant la fréquence des conflits.

Ce constat nous permet d'expliquer l'apparente contradiction entre les méthodes de réduction de destructivité des conflits et les stratégies de sauts arrière illimités, qui au contraire on pour but d'accroître cette destructivité : l'efficacité de ces deux stratégies opposées dépend des instances considérées. Plus précisément, il semble que l'augmentation de la destructivité profite à la résolution des instances satisfaisables tandis que sa réduction améliore la résolution des instances insatisfaisables. L'absence de garantie de réduction de la destructivité du CDCL à ordre partiel, qui semblait être un handicap par rapport au CDCL sans saut arrière, se révèle donc finalement être un avantage, puisqu'il accroît la liberté de manipulation de cette destructivité et permet de la moduler dans un sens ou dans l'autre.

Au final, nous sommes donc parvenus à justifier l'intérêt du CDCL à ordre partiel tant sur le plan théorique que sur le plan pratique. Il reste toutefois nécessaire de mieux comprendre son comportement pour pouvoir l'utiliser de façon satisfaisante. D'une part, il serait utile de mieux déterminer les caractéristiques qui rendent le CDCL à ordre partiel particulièrement efficace sur les instances de vérification de microprocesseurs superscalaires, ce qui nous permettrait d'avoir une idée plus précise des autres types d'instances que cet algorithme est susceptible de résoudre plus efficacement que le CDCL classique. D'autre part, il serait également très bénéfique de comprendre plus en détail les différentes interactions du saut arrière partiel avec les autres composantes du CDCL. Cela permettrait possiblement d'atténuer les interactions négatives qui au contraire rendent la recherche du CDCL à ordre partiel moins efficace sur d'autres instances. Si ces deux aspects sont mieux compris et corrigés, il serait possible d'obtenir un algorithme qui améliore significativement les performances de résolution sur certains types d'instances sans les dégrader en règle générale.

CONCLUSION

Dans cette thèse, nous avons développé plusieurs pistes pour améliorer l'efficacité de la résolution du problème SAT par l'algorithme CDCL, en nous focalisant sur la problématique de la destructivité des sauts arrière. Nous avons tout d'abord travaillé sur l'applicabilité en pratique d'une méthode déjà existante, la décomposition implicite, en proposant des heuristiques simples de construction de décomposition qui permettent le traitement d'instances de plus grande taille possible. Nous avons aussi proposé deux familles de variantes du CDCL, le CDCL sans saut arrière et le CDCL à ordre partiel. La première abolit totalement la notion de saut arrière en permettant la propagation de clauses unitaires à des niveaux de décision non-maximaux ; la seconde conserve la notion de saut arrière, mais défait plus sélectivement les niveaux de décision en considérant leurs relations avec le niveau d'assertion. Le CDCL à ordre partiel a de plus la particularité de rendre la définition de niveau d'assertion non-unique, ce qui permet de développer des heuristiques pour cette dimension de choix supplémentaire.

Nos travaux dans ces différentes directions montre qu'il est très difficile en pratique d'obtenir une amélioration de performances substantielle par rapport au CDCL de base sur des instances quelconques, ce qui est compréhensible étant donnée l'étendue des avancées en efficacité qui ont déjà été obtenues au cours des vingt dernières années, et qui font des solveurs de l'état de l'art actuel des implémentations hautement optimisées. Le principal problème vient du fait que toutes nos approches nécessitent un travail supplémentaire par rapport au CDCL classique, qui handicape dès le départ nos modifications et leur impose une réduction importante du nombre d'étapes de propagation pour obtenir au final une amélioration du temps d'exécution.

Ce dilemme est flagrant dans le cas des décompositions implicites, où l'emploi d'heuristiques de constructions très simples est nécessaire pour pouvoir construire des

décompositions de grandes instances dans un temps et une quantité d'espace raisonnable. Cependant, des heuristiques plus simples produisent des décompositions moins fines, qui ont alors moins de chances d'améliorer la résolution du problème en elle-même.

Dans le cas du CDCL sans saut arrière ou à ordre partiel, le problème est également double : d'une part, la suppression ou la modification du saut arrière nécessite des tâches supplémentaires dans des parties très fréquemment utilisées de l'algorithme, ce qui peut donc grandement ralentir sa vitesse d'exécution. D'autre part, les interactions entre les différentes composantes du CDCL sont si complexes que nos modifications semblent avoir des effets secondaires qui souvent rallongent la taille de la résolution au lieu de la raccourcir.

Malgré ces problèmes, nous avons pu isoler plusieurs familles d'instances SAT sur lesquelles une variante du CDCL à ordre partiel parvient à améliorer substantiellement les performances du CDCL classique, d'autant plus lorsque des heuristiques appropriées pour le choix des niveaux d'assertion sont utilisées. Nos expériences sur ces familles semblent indiquer que ces heuristiques permettent de moduler la destructivité des sauts arrière, et que si la réduction de la destructivité bénéficie à la résolution des instances insatisfaisables, les instances satisfaisables semblent au contraire être plus efficacement résolues lorsque cette destructivité est augmentée.

Ces expérimentations mettent en lumière la difficulté de l'analyse de l'efficacité des solveurs SAT et des interactions complexes entre leurs différentes composantes. Nous pensons que l'utilité du CDCL sans saut arrière et surtout du CDCL à ordre partiel pourrait être accrue en améliorant notre compréhension de ces interactions et en adaptant plus en profondeur leur fonctionnement à nos modifications de la résolution de conflit. D'autre part, il serait intéressant de parvenir à mieux comprendre les raisons de l'efficacité du CDCL à ordre partiel sur ces familles particulières, ce qui nous permettrait peut-être de déterminer des critères pour détecter d'autres instances sur lesquelles cet algorithme est susceptible d'améliorer les performances de résolution.

ANNEXE A

DONNÉES EXPÉRIMENTALES COMPLÉMENTAIRES

A.1 Données du chapitre 4

TABLEAU A.1: Résultats de l'exécution de MINISEP et MINISAT sur diverses instances du problème SAT. Pour chaque instance sont donnés son nombre de variables (*#var*) et de clauses (*#cl*) après les propagations unitaires initiales, le temps nécessaire à la décomposition de l'instance par MINISEP (*tdc*), le nombre total de nœuds de cette décomposition (*#n*), le nombre de nœuds non-décidés, c'est-à-dire dans lesquels aucune décision n'est prise pendant la recherche CDCL (*#nnd*), et enfin le temps total de résolution par MINISEP (*tmsep*) et par MINISAT (*tmsat*). La mention « échec » signifie que MINISEP a été interrompu pendant la construction de la séparation arborescente. Certaines instances ont été résolues dès la phase initiale de propagation unitaire ; dans ce cas, la séparation arborescente n'est pas construite.

instance	*#nvar*	*#cl*	*tdc*	*#n*	*#nnd*	*tmn*	*tmsep*	*tminisat*
AProVE07-01	6654	23237	0.00	3	2	2842	>3600.00	>3600.00
AProVE07-02	5593	19248	0.00	3	2	2473	319.37	336.61
AProVE07-03	3080	10580	0.00	5	2	1231	693.75	727.46
AProVE07-04	9747	63263	0.01	7	6	2114	228.93	33.87
AProVE07-06	10160	72301	0.01	7	6	2532	179.52	111.52
AProVE07-08	4516	16192	0.00	15	12	611	1245.31	773.36
AProVE07-09	29275	135355	0.01	3	1	14259	524.12	1368.69
AProVE07-11	997189	4392589	échec					289.84
AProVE07-15	20888	73465	0.01	19	6	2318	3168.07	135.38
AProVE07-16	52201	181871	0.04	21	20	5657	1425.06	975.30
AProVE07-20	3680	17217	0.00	7	6	929	143.76	80.23
AProVE07-21	3156	10929	0.00	15	14	462	231.37	244.91
AProVE07-22	15477	53512	0.01	15	14	2099	107.06	54.32
AProVE07-25	8690	30972	0.00	7	6	2064	>3600.00	>3600.00
AProVE07-26	21508	78817	0.02	29	27	1731	>3600.00	>3600.00

TABLEAU A.1: Résultats de l'exécution de MINISEP et MINISAT

instance	*#nvar*	*#cl*	*tdc*	*#n*	*#nnd*	*tmn*	*tmsep*	*tminisat*
AProVE07-27	7503	28202	0.00	11	7	1 686	3 050.17	1 733.10
AProVE09-01	46018	141579	0.02	13	0	7 830	50.98	0.55
AProVE09-03	38782	109324	0.01	7	0	9 936	4.14	0.37
AProVE09-05	14639	49029	0.01	15	0	2 031	0.29	0.07
AProVE09-06	77185	262886	0.04	23	0	8 216	6.47	9.98
AProVE09-07	8544	28676	0.00	15	0	1 157	1.16	3.80
AProVE09-08	8541	28667	0.00	15	0	1 162	2.00	0.05
AProVE09-10	64967	238869	0.03	11	0	13 525	2.24	1.36
AProVE09-11	19942	77131	0.01	31	0	1 944	0.24	0.23
AProVE09-12	27066	100426	0.01	17	0	2 903	0.19	0.32
AProVE09-13	7533	26082	0.00	17	0	864	0.03	0.04
AProVE09-15	92834	300532	0.04	23	0	10 541	4.15	3.06
AProVE09-17	32980	106474	0.01	15	0	3 968	27.87	12.77
AProVE09-19	30381	112235	0.02	17	0	3 350	0.12	0.25
AProVE09-20	32885	107686	0.01	15	0	4 296	1 036.86	1 929.83
AProVE09-21	29671	89948	0.01	9	0	6 249	0.59	2.17
AProVE09-22	11199	37518	0.00	33	0	655	0.05	0.06
AProVE09-24	60679	207812	0.03	31	0	4 354	1.88	3.67
AProVE09-25	33451	109992	0.02	27	0	3 025	0.25	0.26
dspam_dump_vc949	109099	355887	0.10	127	123	4 990	>3 600.00	>3 600.00
dspam_dump_vc950	109227	356143	0.12	127	123	4 969	1.24	>3 600.00
dspam_dump_vc962	97619	317504	0.07	31	28	5 888	>3 600.00	115.22
dspam_dump_vc972	268866	902136	0.24	127	122	4 144	>3 600.00	19.62
dspam_dump_vc973	268866	902072	0.22	127	122	4 144	>3 600.00	1.72
dspam_dump_vc1080	114467	370825	0.09	125	122	6 150	3.73	8.80
dspam_dump_vc1081	114595	371081	0.09	127	124	4 446	0.42	>3 600.00
dspam_dump_vc1093	102987	332442	0.05	41	38	5 352	1.51	127.21
dspam_dump_vc1103	275215	920567	0.23	127	126	4 947	>3 600.00	1.50
dspam_dump_vc1104	275215	920503	0.25	127	126	4 947	>3 600.00	>3 600.00
hsat_vc11773	210120	657018	0.09	11	0	39 314	8.72	2.69
hsat_vc11803	244431	765332	0.10	11	7	46 966	8.63	0.32
hsat_vc11813	276994	868354	0.12	11	1	54 118	8.19	8.21
hsat_vc11817	211982	661788	0.10	11	1	38 657	9.12	2.00
hsat_vc11935	156848	485058	0.10	15	10	20 926	2.18	0.92
hsat_vc11944	156753	484901	0.09	15	10	20 998	2.77	0.95
hsat_vc12016	141521	434469	0.07	11	1	26 987	6.98	1.66
hsat_vc12062	175078	540630	0.08	11	6	32 028	6.44	2.58
hsat_vc12070	174186	538090	0.07	11	6	32 014	2.84	2.87
hsat_vc12072	207745	643748	0.10	11	3	41 867	13.06	0.85
itox_vc1033	111213	339049	0.06	15	0	13 683	2.16	3.64

TABLEAU A.1: Résultats de l'exécution de MINISEP et MINISAT

instance	*#nvar*	*#cl*	*tdc*	*#n*	*#nnd*	*tmn*	*tmsep*	*tminisat*
itox_vc1044	114083	347827	0.06	15	0	13 165	1.74	3.83
itox_vc1130	141466	427637	0.10	63	0	13 074	1.64	5.15
itox_vc1138	140218	424212	0.11	63	0	12 429	1.96	4.05
itox_vc1216	141102	426856	0.11	63	62	12 765	0.27	0.14
itox_vc909	104855	318902	0.06	15	0	14 034	4.58	2.70
itox_vc965	résolu par propagation unitaire						0.12	0.10
itox_vc979	108300	329540	0.06	15	14	13 778	0.31	0.15
xinetd_vc56687	résolu par propagation unitaire						0.09	0.09
xinetd_vc56703	résolu par propagation unitaire						0.11	0.10
q_query_2_L324_coli	479545	2194684	0.16	5	4	141 352	2.31	13.87
q_query_3_l37_lambda	21770	114463	0.02	9	0	4 883	7.64	1.57
q_query_3_l38_lambda	22437	118244	0.02	9	0	4 980	29.62	1.88
q_query_3_l39_lambda	23108	122059	0.02	9	0	5 077	20.04	13.95
q_query_3_l40_lambda	23783	125908	0.01	9	0	5 173	62.77	3.94
q_query_3_l41_lambda	24462	129791	0.02	9	0	5 267	96.01	18.61
q_query_3_l42_lambda	25145	133708	0.02	9	0	5 360	94.61	37.76
q_query_3_l43_lambda	25832	137659	0.02	11	0	5 452	282.89	58.80
q_query_3_l44_lambda	26523	141644	0.02	11	2	5 544	456.94	235.11
q_query_3_l45_lambda	27218	145663	0.02	11	10	5 631	439.96	218.49
q_query_3_l46_lambda	27917	149716	0.02	11	10	5 719	412.00	281.71
q_query_3_l47_lambda	28620	153803	0.02	11	10	5 805	399.76	206.37
q_query_3_l48_lambda	29327	157924	0.02	11	10	5 892	435.23	288.98
q_query_3_L60_coli	142680	694061	0.06	5	0	42 327	150.93	175.02
q_query_3_L70_coli	168015	827186	0.09	5	0	49 752	950.50	249.05
q_query_3_L80_coli	193850	965211	0.10	5	2	57 354	1 426.18	431.21
q_query_3_L90_coli	220185	1108136	0.08	5	2	65 082	2 047.30	978.14
q_query_3_L100_coli	247020	1255961	0.10	5	2	72 994	1 980.92	735.89
q_query_3_L150_coli	388695	2068586	0.13	5	3	114 639	>3 600.00	992.57
q_query_3_L200_coli	542870	3003711	0.19	5	3	159 976	3 226.21	1 120.49
eq.atree.braun.7	502	1667	0.00	3	2	208	1.11	1.40
eq.atree.braun.8	681	2267	0.00	3	2	286	8.66	7.37
eq.atree.braun.9	889	2969	0.00	3	2	349	46.21	44.06
eq.atree.braun.10	1108	3715	0.00	3	2	435	290.19	291.80
eq.atree.braun.11	1397	4687	0.00	3	2	536	2 354.62	1 674.00
eq.atree.braun.12	1691	5677	0.00	3	2	672	>3 600.00	>3 600.00
eq.atree.braun.13	2007	6749	0.00	5	4	727	>3 600.00	>3 600.00
blocks-4-ipc5-h21	136631	905605	0.05	3	2	71 489	159.17	90.56
blocks-4-ipc5-h22	148028	936867	0.04	3	2	77 259	272.73	106.13
cube-11-h13	454296	1363529	0.12	5	4	220 331	1 010.53	825.52
cube-11-h14	508181	1525052	0.12	5	3	240 980	>3 600.00	>3 600.00

TABLEAU A.1: Résultats de l'exécution de MINISEP et MINISAT

instance	*#nvar*	*#cl*	*tdc*	*#n*	*#nnd*	*tmn*	*tmsep*	*tminisat*
cube-9-h10	179541	538985	0.04	3	2	91 165	115.11	146.67
cube-9-h11	208415	625631	0.04	3	0	105 617	277.80	222.56
emptyroom-4-h21	70856	207960	0.02	5	4	30 813	3 559.42	>3 600.00
emptyroom-4-h22	75570	221792	0.02	5	4	32 569	>3 600.00	>3 600.00
safe-30-h29	130898	437704	0.18	3	2	66 632	>3 600.00	>3 600.00
safe-30-h30	135756	453084	0.03	3	2	69 100	>3 600.00	>3 600.00
safe-50-h49	619987	2098099	0.11	3	2	313 165	>3 600.00	>3 600.00
safe-50-h50	633342	2141406	0.12	3	2	319 903	>3 600.00	>3 600.00
sortnet-6-ipc5-h11	27660	95688	0.00	3	2	14 080	3 247.27	2 733.18
sortnet-7-ipc5-h15	106573	373870	0.03	3	2	53 978	>3 600.00	>3 600.00
sortnet-7-ipc5-h16	114420	398984	0.03	3	2	58 000	>3 600.00	>3 600.00
sortnet-8-ipc5-h18	339923	1186821	0.07	3	2	170 910	>3 600.00	>3 600.00
sortnet-8-ipc5-h19	360869	1254005	0.07	3	2	181 415	>3 600.00	>3 600.00
uts-l05-ipc5-h26	87447	414447	0.03	3	2	45 984	37.52	22.82
uts-l05-ipc5-h27	95393	439298	0.03	3	2	49 965	82.57	82.55
uts-l06-ipc5-h28	115864	648970	0.04	3	2	60 738	40.16	58.37
uts-l06-ipc5-h29	127804	686758	0.03	3	2	66 738	57.37	87.21
uts-l06-ipc5-h30	139784	724204	0.04	3	2	72 989	1 006.91	259.81
uts-l06-ipc5-h31	151734	761606	0.05	3	2	78 771	226.03	100.61
uts-l06-ipc5-h32	163721	799863	0.04	3	2	85 098	527.21	589.76
uts-l06-ipc5-h33	175636	837166	0.05	3	2	91 000	866.37	627.54
uts-l06-ipc5-h34	187633	874688	0.06	3	0	96 983	59.54	191.51
uts-l06-ipc5-h35	199537	911986	0.05	3	0	102 618	22.41	62.61
abp1-1-k31	14613	47636	0.01	15	2	1 590	14.85	31.21
abp4-1-k31	14613	47636	0.01	15	2	1 590	14.82	31.33
bc56-sensors-1-k391	553353	1750402	0.16	7	6	123 461	1 597.52	2 251.54
bc56-sensors-2-k592	842195	2665180	échec					>3 600.00
bc57-sensors-1-k303	427755	1351691	0.12	7	6	100 486	>3 600.00	>3 600.00
dme-03-1-k247	259307	766556	0.12	15	0	29 530	>3 600.00	>3 600.00
guidance-1-k56	97861	304245	0.06	31	0	6 454	>3 600.00	309.12
motors-stuck-1-k407	640012	2017014	0.18	7	6	136 569	>3 600.00	>3 600.00
motors-stuck-2-k314	483259	1518954	0.14	7	6	104 245	243.28	680.19
motors-stuck-2-k315	484868	1524037	0.14	7	0	104 921	194.94	695.60
valves-gates-1-k617	970456	3062334	échec					>3 600.00
clauses-2	59476	226124	0.02	5	0	18 551	0.86	1.16
clauses-4	206802	823658	0.06	5	0	65 911	30.69	32.97
clauses-6	516596	2128114	0.14	5	0	176 973	965.00	>3 600.00
clauses-8	1085746	4572347	échec					>3 600.00
clauses-10	1685481	7147419	échec					61.71

A.2 Données des chapitres 6 et 7

TABLEAU A.2: Liste des instances utilisées dans les tests du chapitre 6, ainsi que dans certains des tests du chapitre 7

1) aaai10-planning-ipc5-pathways-17-step21
2) aaai10-planning-ipc5-pipesworld-18-step16
3) aaai10-planning-ipc5-TPP-30-step11
4) aes_32_2_keyfind_1
5) AProVE11-11
6) blocks-blocks-37-1.120-NOTKNOWN
7) E02F20
8) E02F22
9) E05X15
10) grid-strips-grid-y-3.035-NOTKNOWN
11) hwmcc10-timeframe-expansion-k45-bobsm5378d2-tseitin
12) hwmcc10-timeframe-expansion-k45-pdtviseisenberg1-tseitin
13) hwmcc10-timeframe-expansion-k45-pdtvissoap1-tseitin
14) hwmcc10-timeframe-expansion-k50-pdtswvsam6x8p3-tseitin
15) korf-15
16) openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.085-SAT
17) slp-synthesis-aes-bottom12
18) slp-synthesis-aes-bottom13
19) smtlib-qfbv-aigs-lfsr_008_063_080-tseitin
20) smtlib-qfbv-aigs-lfsr_008_079_112-tseitin
21) sokoban-sequential-p145-microban-sequential.030-NOTKNOWN
22) traffic_fb_unknown
23) traffic_pcb_unknown
24) transport-transport-two-cities-sequential-15nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.040-SAT
25) 12pipe_bug6_q0.used-as.sat04-725
26) 6pipe_6_ooo.shuffled-as.sat03-413
27) 9dlx_vliw_at_b_iq6.used-as.sat04-347
28) 9dlx_vliw_at_b_iq7
29) 9vliw_m_9stages_iq3_C1_bug5
30) abb313GPIA-9-tr.used-as.sat04-321
31) ACG-15-5p1
32) all.used-as.sat04-986
33) AProVE07-03
34) AProVE07-21
35) bc57-sensors-1-k303-unsat.shuffled-as.sat03-406
36) cube-11-h14-sat
37) dated-5-11-u

38) dp04s04.shuffled
39) eq.atree.braun.11.unsat
40) gss-19-s100
41) hard-18-U-10652
42) homer14.shuffled
43) ibm-2002-21r-k95
44) ibm-2004-23-k100
45) IBM_FV_2004_rule_batch_30_SAT_dat.k55
46) k2fix_gr_rcs_w9.shuffled
47) k2mul.miter.shuffled-as.sat03-355
48) manol-pipe-c10nidw
49) manol-pipe-f7idw
50) md5_48_3
51) ndhf_xits_19_UNKNOWN
52) post-c32s-gcdm16-23
53) post-cbmc-zfcp-2.8-u2-noholes
54) q_query_3_L150_coli.sat
55) rand_net60-40-10.shuffled
56) rbcl_xits_15_SAT
57) UCG-20-5p1
58) UR-10-10p1
59) velev-npe-1.0-9dlx-b71
60) velev-pipe-o-uns-1.1-6
61) vmpc_25.renamed-as.sat05-1913
62) vmpc_29.renamed-as.sat05-1916

BIBLIOGRAPHIE

Aloul, F. A., I. L. Markov, et K. A. Sakallah. 2004. « MINCE : A static global variable-ordering heuristic for SAT search and BDD manipulation », *Journal Of Universal Computer Science (J.UCS)*, vol. 10, no. 12, p. 1562–1596. 142

Amir, E. 2010. « Approximation algorithms for treewidth », *Algorithmica*, vol. 56, no. 4, p. 448–479. 132

Amir, E. et S. McIlraith. 2005. « Partition-based logical reasoning for first-order and propositional theories », *Artificial Intelligence*, vol. 162, no. 1–2, p. 49–88. 107

Argelich, J., A. Cabiscol, I. Lynce, et F. Manyà. 2012. « Efficient encodings from CSP into SAT, and from MaxCSP into MaxSAT », *Journal of Multi-Valued Logic and Soft Computing*, vol. 19, no. 1–3, p. 3–23. 39

Arnborg, S., D. G. Corneil, et A. Proskurowski. 1987. « Complexity of finding embeddings in a k-tree », *SIAM Journal on Algebraic and Discrete Methods*, vol. 8, no. 2, p. 277–284. 99

Arnborg, S. et A. Proskurowski. 1989. « Linear time algorithms for NP-hard problems restricted to partial k-trees », *Discrete Applied Mathematics*, vol. 23, no. 1, p. 11–24. 99

Audemard, G. et L. Simon. 2009. « Predicting learnt clauses quality in modern SAT solvers ». In Boutilier, C., éditeur, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*, p. 399–404. 5, 70, 118, 190, 257

Babić, D. et A. J. Hu. 2007. « Structural abstraction of software verification conditions ». In Damm, W. et H. Hermanns, éditeurs, *Computer Aided Verification, 19th*

International Conference (CAV 2007). Coll. « Lecture Notes in Computer Science », no 4590, p. 366–378. Springer. 135

Bacchus, F., S. Dalmao, et T. Pitassi. 2003. DPLL with caching : A new algorithm for #SAT and bayesian inference. Rapport no. TR03-003, Electronic Colloquium on Computational Complexity. 116

Baker, A. B. 1994. « The hazards of fancy backtracking ». In Hayes-Roth, B. et R. E. Korf, éditeurs, *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*. T. 1, p. 288–293. AAAI Press / The MIT Press. 126

Baptista, L. et J. Marques-Silva. 2000. « Using randomization and learning to solve hard real-world instances of satisfiability ». In Dechter, R., éditeur, *Principles and Practice of Constraint Programming, 6th International Conference (CP 2000)*. Coll. « Lecture Notes in Computer Science », no 1894, p. 489–494. Springer. 72

Barrett, C. W., R. Sebastiani, S. A. Seshia, et C. Tinelli. 2009. « Satisfiability modulo theories ». In Biere, A., M. Heule, H. van Maaren, et T. Walsh, éditeurs, *Handbook of Satisfiability*. Coll. « Frontiers in Artificial Intelligence and Applications », no 185, p. 825–885. IOS Press. 39

Bayardo Jr., R. J. et J. D. Pehoushek. 2000. « Counting models using connected components ». In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00) and Twelfth Conference on Innovative Applications of Artificial Intelligence (IAAI-00*, p. 157–162. AAAI Press / The MIT Press. 116

Bayardo Jr., R. J. et R. C. Schrag. 1997. « Using CSP look-back techniques to solve real-world SAT instances ». In Kuipers, B. et B. L. Webber, éditeurs, *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI 97) and Ninth Innovative Applications of Artificial Intelligence Conference (IAAI 97)*, p. 203–208. AAAI Press / The MIT Press. 59, 71, 92

Bennaceur, H. 1996. « The satisfiability problem regarded as a constraint satisfaction

problem ». In Wahlster, W., éditeur, *12th European Conference on Artificial Intelligence (ECAI 96)*, p. 155–159. Wiley. 87

Bhalla, A., I. Lynce, J. T. de Sousa, et J. Marques-Silva. 2005. « Heuristic-based backtracking relaxation for propositional satisfiability », *Journal of Automated Reasoning*, vol. 35, no. 1–3, p. 3–24. 127

Biere, A. 2005. « Resolve and expand ». In Hoos, H. H. et D. G. Mitchell, éditeurs, *Theory and Applications of Satisfiability Testing, 7th International Conference (SAT 2004)*. Coll. « Lecture Notes in Computer Science », no 3542, p. 59–70. Springer. 39

———. 2008. « PicoSAT essentials », *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 4, no. 2–4, p. 75–97. 73

Biere, A., A. Cimatti, E. M. Clarke, O. Strichman, et Y. Zhu. 2003. « Bounded model checking », *Advances in Computers*, vol. 58, p. 117–148. 1, 39

Biere, A. et C. Sinz. 2006. « Decomposing SAT problems into connected components », *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, p. 201–208. 3, 108, 115

Bjesse, P., J. H. Kukula, R. F. Damiano, T. Stanion, et Y. Zhu. 2004. « Guiding SAT diagnosis with tree decompositions ». In Giunchiglia, E. et A. Tacchella, éditeurs, *Theory and Applications of Satisfiability Testing, 6th International Conference (SAT 2003)*. Coll. « Lecture Notes in Computer Science », no 2919, p. 315–329. Springer. 3, 112, 133, 134, 135

Bliek, C. 1998. « Generalizing partial order and dynamic backtracking ». In Mostow, J. et C. Rich, éditeurs, *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI 98) and Tenth Innovative Applications of Artificial Intelligence Conference, (IAAI 98)*, p. 319–325. AAAI Press / The MIT Press. 3, 123, 124

Bodlaender, H. L. 1988. « Dynamic programming on graphs with bounded treewidth ». In Lepistö, T. et A. Salomaa, éditeurs, *Automata, Languages and Programming, 15th*

International Colloquium (ICALP88). Coll. « Lecture Notes in Computer Science », no 317, p. 105–118. Springer. 99

———. 1996. « A linear time algorithm for finding tree-decompositions of small treewidth », *SIAM Journal on Computing*, vol. 25, no. 6, p. 1305–1317. 99

Buro, M. et H. Kleine Büning. 1993. « Report on a SAT competition », *Bulletin of the EATCS*, vol. 49, p. 143–151. 57

Clautiaux, F., A. Moukrim, S. Nègre, et J. Carlier. 2004. « Heuristic and metaheuristic methods for computing graph treewidth », *RAIRO – Operations Research*, vol. 38, no. 1, p. 13–26. 133

Cohen, D., P. Jeavons, et M. Gyssens. 2008. « A unified theory of structural tractability for constraint satisfaction problems », *Journal of Computer and System Sciences*, vol. 74, no. 5, p. 721–743. 102

Cohen, D. A. et M. J. Green. 2006. « Typed guarded decompositions for constraint satisfaction ». In Benhamou, F., éditeur, *Principles and Practice of Constraint Programming, 12th International Conference (CP 2006)*. Coll. « Lecture Notes in Computer Science », no 4204, p. 122–136. Springer. 102

Cook, S. A. 1971. « The complexity of theorem-proving procedures ». In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC '71)*, p. 151–158. ACM. 1, 30

Cooper, M. C. 1989. « An optimal k-consistency algorithm », *Artificial Intelligence*, vol. 41, no. 1, p. 89–95. 89

Corblin, F., L. Bordeaux, E. Fanchon, Y. Hamadi, et L. Trilling. 2011. « Connections and integration with SAT solvers : A survey and a case study in computational biology ». In van Hentenryck, P. et M. Milano, éditeurs, *Hybrid Optimization – The Ten Years of CPAIOR*. Coll. « Springer Optimization and Its Applications », no 45, p. 425–461. Springer. 135

Darwiche, A. 2001. « Recursive conditioning », *Artificial Intelligence*, vol. 126, no. 1–2, p. 5–41. 101, 102

Darwiche, A. et M. Hopkins. 2001. « Using recursive decomposition to construct elimination orders, jointrees, and dtrees ». In Benferhat, S. et P. Besnard, éditeurs, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty, 6^{th} European Conference (ECSQARU 2001)*. Coll. « Lecture Notes in Computer Science », no 2143, p. 180–191. Springer. 133

Davis, M., G. Logemann, et D. W. Loveland. 1962. « A machine program for theorem-proving », *Communications of the ACM*, vol. 5, no. 7, p. 394–397. 47

Davis, M. et H. Putnam. 1960. « A computing procedure for quantification theory », *Journal of The ACM*, vol. 7, no. 3, p. 201–215. 40, 45

de Givry, S., T. Schiex, et G. Verfaillie. 2006. « Exploiting tree decomposition and soft local consistency in weighted CSP ». In *Proceedings of The Twenty-First National Conference on Artificial Intelligence (AAAI-06) and the Eighteenth Innovative Applications of Artificial Intelligence Conference (IAAI-06)*, p. 22–27. AAAI Press. 112

de Kleer, J. 1989. « A comparison of ATMS and CSP techniques ». In *Eleventh International Joint Conference on Artificial Intelligence (IJCAI-89)*. T. 1, p. 290–296. Morgan Kaufmann. 85

Dechter, R. 1990. « Enhancement schemes for constraint processing : Backjumping, learning, and cutset decomposition », *Artificial Intelligence*, vol. 41, no. 3, p. 273–312. 92, 93

———. 1992. « From local to global consistency », *Artificial Intelligence*, vol. 55, no. 1, p. 87–107. 89

———. 2003. *Constraint Processing*. Coll. « The Morgan Kaufmann Series in Artificial Intelligence ». Morgan Kaufmann. 103

Dechter, R. et Y. E. Fattah. 2001. « Topological parameters for time-space tradeoff », *Artificial Intelligence*, vol. 125, no. 1–2, p. 93–118. 106

Dechter, R. et J. Pearl. 1987. « Network-based heuristics for constraint-satisfaction problems », *Artificial Intelligence*, vol. 34, no. 1, p. 1–38. 90, 106

———. 1989. « Tree clustering for constraint networks », *Artificial Intelligence*, vol. 38, no. 3, p. 353–366. 84, 101, 104, 105

Drabent, W. et M. Miłkowska. 2005. « Proving correctness and completeness of normal programs – a declarative approach », vol. 5, no. 6, p. 669–711. 27

Drias, H. 1998. « A Monte Carlo algorithm for the satisfiability problem ». In Mira, J. et A. P. D. Pobil, éditeurs, *Methodology and Tools in Knowledge-Based Systems – 11th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE-98)*, T. I. Coll. « Lecture Notes in Computer Science », no 1415, p. 159–168. Springer. 28

Durairaj, V. et P. Kalla. 2004. « Exploiting hypergraph partitioning for efficient boolean satisfiability ». In *Ninth IEEE International High-Level Design Validation and Test Workshop (HLDVT 2004)*, p. 141–146. IEEE Computer Society. 3, 109, 111, 133, 134, 135, 138

Eén, N. et N. Sörensson. 2004. « An extensible SAT-solver ». In Giunchiglia, E. et A. Tacchella, éditeurs, *Theory and Applications of Satisfiability Testing, 6th International Conference (SAT 2003)*. Coll. « Lecture Notes in Computer Science », no 2919, p. 502–518. Springer. 5, 69, 70, 136, 146, 190

Eibach, T., E. Pilz, et G. Völkel. 2008. « Attacking bivium using SAT solvers ». In Kleine Büning, H. et X. Zhao, éditeurs, *Theory and Applications of Satisfiability Testing, 11th International Conference (SAT 2008)*. Coll. « Lecture Notes in Computer Science », no 4996, p. 63–76. Springer. 1, 39

Fiduccia, C. M. et R. M. Mattheyses. 1982. « A linear-time heuristic for improving

network partitions ». In Crabbe, J. S., C. E. Radke, et H. Ofek, éditeurs, *Proceedings of the 19th Design Automation Conference (DAC '82)*, p. 175–181. ACM/IEEE. 133

Franco, J. et J. Martin. 2009. « A history of satisfiability ». In Biere, A., M. Heule, H. van Maaren, et T. Walsh, éditeurs, *Handbook of Satisfiability*. Coll. « Frontiers in Artificial Intelligence and Applications », no 185, p. 3–74. IOS Press. 57

Freuder, E. C. 1994. « Exploiting structure in constraint satisfaction problems ». In Mayoh, B., E. Tyugu, et J. Penjaam, éditeurs, *Proceedings of the NATO Advanced Study Institute on Constraint Programming*, p. 54–79. Springer. 100

Garey, M. R. et D. S. Johnson. 1979. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. Coll. « A Series of books in the mathematical sciences ». W.H. Freeman & Co Ltd. 26, 31

Gaschnig, J. 1979. « Performance measurement and analysis of certain search algorithms ». Thèse de Doctorat, Carnegie-Mellon University. 92

Giesl, J., P. Schneider-Kamp, et R. Thiemann. 2006. « AProVE 1.2 : Automatic termination proofs in the dependency pair framework ». In Furbach, U. et N. Shankar, éditeurs, *Automated Reasoning, Third International Joint Conference (IJCAR 2006)*. Coll. « Lecture Notes in Artificial Intelligence », no 4130, p. 281–286. Springer. 135

Ginsberg, M. L. 1993. « Dynamic backtracking », *Journal of Artificial Intelligence Research*, vol. 1, p. 25–46. 3, 118, 119, 126

Ginsberg, M. L. et D. McAllester. 1994. « GSAT and dynamic backtracking ». In Borning, A., éditeur, *Principles and Practice of Constraint Programming, Second International Workshop (PPCP'94)*. Coll. « Lecture Notes in Computer Science », no 874, p. 243–265. Springer. 3, 121, 123

Goldberg, A. 1979. « Average case complexity of the satisfiability problem ». In *Proceedings of the fourth Workshop on Automated Deduction*, p. 1–6. 54

Goldberg, E. I., M. R. Prasad, et R. K. Brayton. 2001. « Using SAT for combinational equivalence checking ». In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2001)*, p. 114–121. IEEE Computer Society. 1, 39

Gomes, C. P., B. Selman, N. Crato, et H. Kautz. 2000. « Heavy-tailed phenomena in satisfiability and constraint satisfaction problems », *Journal of Automated Reasoning*, vol. 24, no. 1–2, p. 67–100. 72

Gottlob, G., M. Hutle, et F. Wotawa. 2002. « Combining hypertree, bicomp, and hinge decomposition ». In van Harmelen, F., éditeur, *Proceedings of the 15th Eureopean Conference on Artificial Intelligence (ECAI'2002)*, p. 161–165. IOS Press. 102

Gottlob, G., N. Leone, et F. Scarcello. 2000. « A comparison of structural CSP decomposition methods », *Artificial Intelligence*, vol. 124, no. 2, p. 243–282. 102

Grastien, A., Anbulagan, J. Rintanen, et E. Kelareva. 2007. « Diagnosis of discrete-event systems using satisfiability algorithms ». In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI 2007)*, p. 305–310. AAAI Press. 1, 39

Gyssens, M., P. G. Jeavons, et D. A. Cohen. 1994. « Decomposing constraint satisfaction problems using database techniques », *Artificial Intelligence*, vol. 66, no. 1, p. 57–89. 100

Habbas, Z., K. Amroun, et D. Singer. 2011. « Solving non binary constraint satisfaction problems with dual backtracking on hypertree decomposition ». In Filipe, J. et A. L. N. Fred, éditeurs, *Proceedings of the 3rd International Conference on Agents and Artificial Intelligence (ICAART 2011), Volume 1 – Artificial Intelligence*, p. 146–156. SciTePress. 3, 111

Habet, D., L. Paris, et C. Terrioux. 2009. « A tree decomposition based approach to solve structured SAT instances ». In *21st International Conference on Tools with Artificial Intelligence (ICTAI 2009)*, p. 115–122. IEEE Computer Society. 112

Haim, S. et M. Heule. 2010. Towards ultra rapid restarts. Rapport, UNSW and TU Delft. 73

Haken, A. 1985. « The intractability of resolution », *Theoretical Computer Science*, vol. 39, p. 297–308. 46, 78

Hammerl, T. et N. Musliu. 2010. « Ant colony optimization for tree decompositions ». In Cowling, P. I. et P. Merz, éditeurs, *Evolutionary Computation in Combinatorial Optimization, 10th European Conference (EvoCOP 2010)*. Coll. « Lecture Notes in Computer Science », no 6022, p. 95–106. Springer. 133

Haralick, R. M. et G. L. Elliott. 1980. « Increasing tree search efficiency for constraint satisfaction problems », *Artificial Intelligence*, vol. 14, no. 3, p. 263–313. 91

Huang, J. 2007. « The effect of restarts on the efficiency of clause learning ». In Veloso, M. M., éditeur, *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI 2007)*, p. 2318–2323. 73

Huang, J. et A. Darwiche. 2003. « A structure-based variable ordering heuristic for SAT ». In Gottlob, G. et T. Walsh, éditeurs, *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, p. 1167–1172. Morgan Kaufmann. 3, 110, 111, 133, 134, 135, 138

Järvisalo, M. 2007. Equivalence checking hardware multiplier designs. `http://www.satcompetition.org/2007/satcomp07.benchmark-description.jarvisalo.pdf`. 135

Jégou, P., S. N. Ndiaye, et C. Terrioux. 2009. « Combined strategies for decomposition-based methods for solving CSPs ». In *21st International Conference on Tools with Artificial Intelligence (ICTAI 2009)*, p. 184–192. IEEE Computer Society. 112

Jégou, P. et C. Terrioux. 2003. « Hybrid backtracking bounded by tree-decomposition of constraint networks », vol. 146, no. 1, p. 43–75. 3, 106, 111, 112, 133

———. 2004. « Decomposition and good recording for solving Max-CSPs ». In de Mántaras, R. L. et L. Saitta, éditeurs, *Proceedings of the 16th Eureopean Conference on Artificial Intelligence (ECAI'2004), including Prestigious Applicants of Intelligent Systems (PAIS 2004)*, p. 196–200. IOS Press. 112

Jeroslow, R. G. et J. Wang. 1990. « Solving propositional satisfiability problems », *Annals of Mathematics and Artificial Intelligence*, vol. 1, p. 167–187. 57

Jussien, N., R. Debruyne, et P. Boizumault. 2000. « Maintaining arc-consistency within dynamic backtracking ». In Dechter, R., éditeur, *Principles and Practice of Constraint Programming, 6th International Conference (CP 2000)*. Coll. « Lecture Notes in Computer Science », no 1894, p. 249–261. Springer. 124

Jussien, N. et O. Lhomme. 2002. « Local search with constraint propagation and conflict-based heuristics », *Artificial Intelligence*, vol. 139, no. 1, p. 21–45. 128

Karypis, G., R. Aggarwal, V. Kumar, et S. Shekhar. 1999. « Multilevel hypergraph partitioning : Applications in VLSI domain », *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, p. 69–79. 133, 138

Kautz, H. et B. Selman. 1992. « Planning as satisfiability ». In Neumann, B., éditeur, *10th European Conference on Artificial Intelligence (ECAI'92)*, p. 359–363. Wiley. 1, 39

Koster, A. M. C. A., S. P. M. van Hoesel, et A. W. J. Kolen. 2002. « Solving partial constraint satisfaction problems with tree decomposition », *Networks*, vol. 40, no. 3, p. 170–180. 107

Le Berre, D., L. Simon, E. S. Hirsch, C. Sinz, O. Roussel, M. Järvisalo, A. Balint, et A. Belov. 2012. The international SAT Competitions web page. `http://www.satcompetition.org`. 5, 134

Li, W. et P. van Beek. 2004. « Guiding real-world SAT solving with dynamic hypergraph separator decomposition ». In *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, p. 542–548. IEEE Computer Society. 3, 110, 114, 133, 134, 135

Lin, Z., Y. Zhang, et H. Hernandez. 2006. « Fast SAT-based answet set solver ». In *Proceedings of The Twenty-First National Conference on Artificial Intelligence (AAAI-*

06) and the Eighteenth Innovative Applications of Artificial Intelligence Conference (IAAI-06), p. 92–97. AAAI Press. 39

Lynce, I. et J. Marques-Silva. 2002. « Complete unrestricted backtracking algorithms for satisfiability ». In *Proceedings of the Fifth International Symposium on Theory and Applications of Satisfiability Testing (SAT 2002)*, p. 214–221. 127, 128

———. 2008. « Haplotype inference with boolean satisfiability », *International Journal on Artificial Intelligence Tools*, vol. 17, no. 2, p. 355–387. 1, 39

Mackworth, A. K. 1977. « Consistency in networks of relations », *Artificial Intelligence*, vol. 8, no. 1, p. 99–118. 84, 92

Marques-Silva, J. 1999. « The impact of branching heuristics in propositional satisfiability algorithms ». In Barahona, P. et J. J. Alferes, éditeurs, *Progress in Artificial Intelligence, 9^{th} Portuguese Conference on Artificial Intelligence (EPIA '99)*. Coll. « Lecture Notes in Computer Science », no 1695, p. 62–74. Springer. 57

Marques-Silva, J., I. Lynce, et S. Malik. 2009. « Conflict-driven clause learning SAT solvers ». In Biere, A., M. Heule, H. van Maaren, et T. Walsh, éditeurs, *Handbook of Satisfiability*. Coll. « Frontiers in Artificial Intelligence and Applications », no 185, p. 131–153. IOS Press. 71

Marques-Silva, J. P. et A. L. Oliveira. 1997. « Improving satisfiability algorithms with dominance and partitioning ». In *Workshop Notes of the ACM/IEEE International Workshop on Logic Synthesis (IWLS'97)*. 116

Marques-Silva, J. P. et K. A. Sakallah. 1999. « GRASP : A search algorithm for propositional satisfiability », *IEEE Transactions on Computers*, vol. 48, no. 5, p. 506–521. 2, 59, 64, 71, 79

McAllester, D. A. 1993. Partial order backtracking. Research note, Artificial Intelligence Laboratory, MIT. 3, 119, 120, 121, 122

Memik, S. O. et F. Fallah. 2002. « Accelerated SAT-based scheduling of control/data flow graphs ». In *20th International Conference on Computer Design (ICCD 2002), VLSI in Computers and Processors*, p. 395–400. IEEE Computer Society. 1, 39

Mironov, I. et L. Zhang. 2006. « Applications of SAT solvers to cryptanalysis of hash functions ». In Biere, A. et C. P. Gomes, éditeurs, *Theory and Applications of Satisfiability Testing, 9th International Conference (SAT 2006)*. Coll. « Lecture Notes in Computer Science », no 4121, p. 102–115. Springer. 1, 39

Mitchell, D. G. 1998. « Hard problems for CSP algorithms ». In Mostow, J. et C. Rich, éditeurs, *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI 98) and Tenth Innovative Applications of Artificial Intelligence Conference, (IAAI 98)*, p. 398–405. AAAI Press / The MIT Press. 91, 93

Mohr, R. et T. C. Henderson. 1986. « Arc and path consistency revisited », *Artificial Intelligence*, vol. 28, no. 2, p. 225–233. 89

Monnet, A. et R. Villemaire. 2010. « Scalable formula decomposition for propositional satisfiability ». In Desai, B. C., éditeur, *Proceedings of the Third C* Conference on Computer Science and Software Engineering (C^{3}S^{2}E '10)*, p. 43–52. ACM Press. 3

———. 2012a. « CDCL with less destructive backtracking through partial ordering ». In *Third Workshop on Practical Aspects of Automated Reasoning (PAAR-2012)*, p. 124–138. 4

———. 2012b. « Efficient partial order CDCL using assertion level choice heuristics ». In Angelopoulos, N. et R. Bagnara, éditeurs, *12th International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS 2012)*, p. 26–41. 4

Moskewicz, M. W., C. F. Madigan, Y. Zhao, L. Zhang, et S. Malik. 2001. « Chaff : engineering an efficient SAT solver ». In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, p. 530–535. ACM Press. 49, 64, 69, 70, 134

Mueller, E. T. 2004. « Event calculus reasoning through satisfiability », *Journal of Logic and Computation*, vol. 14, no. 5, p. 703–730. 39

Musliu, N. 2008. « An iterative heuristic algorithm for tree decomposition ». In Cotta, C. et J. I. van Hemert, éditeurs, *Recent Advances in Evolutionary Computation for Combinatorial Optimization*. Coll. « Studies in Computational Intelligence », no 153, p. 133–150. Springer. 133

Nam, G.-J., K. A. Sakallah, et R. A. Rutenbar. 1999. « Satisfiability-based detailed FPGA routing ». In *12th International Conference on VLSI Design (VLSI Design 1999)*, p. 574–577. IEEE Computer Society. 1, 39

Narain, S. 2005. « Network configuration management via model finding ». In *Proceedings of the 19th Conference on Systems Administration (LISA 2005)*, p. 155–168. USENIX. 136

Palacios, H. et H. Geffner. 2006. « Mapping conformant planning into SAT through compilation and projection ». In Marín, R., E. Onaindía, A. Bugarín, et J. Santos, éditeurs, *Current Topics in Artificial Intelligence – 11th Conference of the Spanish Association for Artificial Intelligence (CAEPIA 2005)*. Coll. « Lecture Notes in Computer Science », no 4177, p. 311–320. Springer. 135

Park, T. J. et A. van Gelder. 1996. « Partitioning methods for satisfiability testing on large formulas ». In McRobbie, M. A. et J. K. Slaney, éditeurs, *13th International Conference on Automated Deduction (CADE-13)*. Coll. « Lecture Notes in Computer Science », no 1104, p. 748–762. Springer. 109, 133, 135

Pipatsrisawat, K. et A. Darwiche. 2007. « A lightweight component caching scheme for satisfiability solvers ». In Marques-Silva, J. P. et K. A. Sakallah, éditeurs, *Theory and Applications of Satisfiability Testing, 10th International Conference (SAT 2007)*. Coll. « Lecture Notes in Computer Science », no 4501, p. 294–299. Springer. 3, 116, 117

———. 2009. « Width-based restart policies for clause-learning satisfiability solvers ». In Kullmann, O., éditeur, *Theory and Applications of Satisfiability Testing, 12th In-*

ternational Conference (SAT 2009). Coll. « Lecture Notes in Computer Science », no 5584, p. 341–355. Springer. 73

Pralet, C. et G. Verfaillie. 2005. « About the choice of the variable to unassign in a decision repair algorithm », *RAIRO – Operations Research*, vol. 39, no. 1, p. 55–74. 129

Prestwich, S. 2000. « A hybrid search architecture applied to hard random 3-SAT and low-autocorrelation binary sequences ». In Dechter, R., éditeur, *Principles and Practice of Constraint Programming, 6th International Conference (CP 2000)*. Coll. « Lecture Notes in Computer Science », no 1894, p. 337–352. Springer. 128

Prosser, P. 1993. « Hybrid algorithms for the constraint satisfaction problem », *Computational Intelligence*, vol. 9, no. 3, p. 268–299. 3, 92

Robertson, N. et P. D. Seymour. 1986. « Graph minors. II. Algorithmic aspects of tree-width », *Journal of Algorithms*, vol. 7, no. 3, p. 309–322. 96, 100

Saad, E. 2009. « Probabilistic reasoning by SAT solvers ». In Sossai, C. et G. Chemello, éditeurs, *Symbolic and Quantitative Approaches to Reasoning with Uncertainty, 10th European Conference (ECSQARU 2009)*. Coll. « Lecture Notes in Computer Science », no 5590, p. 663–675. Springer. 39

Sabin, D. et E. C. Freuder. 1994. « Contradicting conventional wisdom in constraint satisfaction ». In Borning, A., éditeur, *Principles and Practice of Constraint Programming, Second International Workshop (PPCP'94)*. Coll. « Lecture Notes in Computer Science », no 874, p. 10–20. Springer. 91

Samulowitz, H. et F. Bacchus. 2007. « Dynamically partitioning for solving QBF ». In Marques-Silva, J. P. et K. A. Sakallah, éditeurs, *Theory and Applications of Satisfiability Testing, 10th International Conference (SAT 2007)*. Coll. « Lecture Notes in Computer Science », no 4501, p. 215–229. Springer. 116

Sánchez, M., J. Larrosa, et P. Meseguer. 2005. « Tree decomposition with function filtering ». In van Beek, P., éditeur, *Principles and Practice of Constraint Program-*

ming, 11th International Conference (CP 2005). Coll. « Lecture Notes in Computer Science », no 3709, p. 593–606. Springer. 107

Sang, T., F. Bacchus, P. Beame, H. A. Kautz, et T. Pitassi. 2004. « Combining component caching and clause learning for effective model counting ». In *Theory and Applications of Satisfiability Testing, 7th International Conference (SAT 2004), Online Proceedings*. 116

Schuppan, V. et A. Biere. 2004. « Efficient reduction of finite state model checking to reachability analysis », *International Journal on Software Tools for Technology Transfer*, vol. 5, no. 2–3, p. 185–204. 136

Selman, B., H. Kautz, et B. Cohen. 1996. « Local search strategies for satisfiability testing ». In Johnson, D. S. et M. A. Trick, éditeurs, *Cliques, Coloring, and Satisfiability – Second DIMACS Implementation Challenge*. Coll. « DIMACS Series in Discrete Mathematics and Theoretical Computer Science », no 26, p. 521–533. American Mathematical Society. 28

Singer, D. et A. Monnet. 2008. « JaCk-SAT : A new parallel scheme to solve the satisfiability problem (SAT) based on join-and-check ». In Wyrzykowski, R., J. Dongarra, K. Karczewski, et J. Wasniewski, éditeurs, *Parallel Processing and Applied Mathematics, 7th International Conference (PPAM 2007)*. Coll. « Lecture Notes in Computer Science », no 4967, p. 249–258. Springer. 107

Sörensson, N. et N. Eén. 2009. « MiniSat 2.1 and MiniSat++ 1.0 – SAT race 2008 editions ». In *SAT 2009 competitive events booklet : preliminary version*, p. 31–32. `http://www.cril.univ-artois.fr/SAT09/solvers/booklet.pdf`. 67

Stephan, P. R., R. K. Brayton, et A. L. Sangiovanni-Vincentelli. 1996. « Combinational test generation using satisfiability », *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 9, p. 1167–1176. 1, 39

Tarjan, R. E. et M. Yannakakis. 1984. « Simple linear-time algorithms to test chordality

of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs », *SIAM Journal on Computing*, vol. 13, no. 3, p. 566–579. 101

Terrioux, C. et P. Jégou. 2003. « Bounded backtracking for the valued constraint satisfaction problems ». In Rossi, F., éditeur, *Principles and Practice of Constraint Programming, 9th International Conference (CP 2003)*. Coll. « Lecture Notes in Computer Science », no 2833, p. 709–723. Springer. 112

Thiffault, C., F. Bacchus, et T. Walsh. 2004. « Solving non-clausal formulas with DPLL search ». In Wallace, M., éditeur, *Principles and Practice of Constraint Programming, 10th International Conference (CP 2004)*. Coll. « Lecture Notes in Computer Science », no 3258, p. 663–678. Springer. 37

Tseitin, G. S. 1983. « On the complexity of proofs in propositional logics ». In Siekmann, J. H. et G. Wrightson, éditeurs, *Automation of reasoning, Classical papers on computational logic*. T. 2. Springer. 35

van Dongen, M., C. Lecoutre, et O. Roussel. 2009. Fourth international CSP solver competition (CSP, Max-CSP and Weighted-CSP competition). `http://www.cril.univ-artois.fr/CPAI09/`. 150

van Leeuwen, J. 1990. « Graph algorithms ». In van Leeuwen, J., éditeur, *Algorithms and Complexity*. T. A, série *Handbook of Theoretical Computer Science*, p. 525–631. Elsevier / MIT Press. 99

Velev, M. 2004. Miroslav Velev's SAT benchmarks. `http://www.miroslav-velev.com/sat_benchmarks.html`. 284

Velev, M. N. et R. E. Bryant. 2003. « Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors », *Journal of Symbolic Computation*, vol. 35, no. 2, p. 73–106. 1, 39, 284

Yannakakis, M. 1981. « Computing the minimum fill-in is NP-complete », *SIAM Journal on Algebraic and Discrete Methods*, vol. 2, no. 1, p. 77–79. 101

Zhang, H. et M. Stickel. 2000. « Implementing the Davis-Putnam method », *Journal of Automated Reasoning*, vol. 24, no. 1/2, p. 277–296. 66

Zhang, L. 2003. « Searching for truth : Techniques for satisfiability of boolean formulas ». Thèse de Doctorat, Princeton University. 76

Zhang, L., C. F. Madigan, M. W. Moskewicz, et S. Malik. 2001. « Efficient conflict driven learning in boolean satisfiability solver ». In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'01)*, p. 279–285, Piscataway, NJ, USA. ACM. 71, 169

Zheng, Y. et B. Y. Choueiry. 2005. « New structural decomposition techniques for constraint satisfaction problems ». In Faltings, B., A. Petcu, F. Fages, et F. Rossi, éditeurs, *Recent Advances in Constraints, Joint ERCIM/CoLogNet International Workshop on Constraint Solving and Constraint Logic Programming (CSCLP 2004)*. Coll. « Lecture Notes in Computer Science », no 3419, p. 113–127. Springer. 102

Zivan, R., U. Shapen, M. Zazone, et A. Meisels. 2009. « MAC-DBT revisited ». In Larrosa, J. et B. O'Sullivan, éditeurs, *Recent Advances in Constraints – 14^{th} Annual ERCIM International Workshop on Constraint Solving and Constraint Logic Programming (CSCLP 2009)*. Coll. « Lecture Notes in Computer Science », no 6384, p. 139–153. Springer. 93