UNIVERSITÉ DU QUÉBEC À MONTRÉAL

SOUS-ARBRES INDUITS PLEINEMENT FEUILLUS

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN MATHÉMATIQUES

PAR

ÉMILE NADEAU

NOVEMBRE 2018

REMERCIEMENTS

Je voudrais d'abord remercier Franco, mon directeur de recherche, pour m'avoir initié à la recherche. Ta joie de vivre est contagieuse et ta confiance toujours réconfortante. Merci de m'avoir guidé et conseillé tout au long de la rédaction de ce mémoire. Tes commentaires m'ont grandement aidé à améliorer la qualité de ce travail.

Merci à Alexandre, Alain, Élise et Mélodie pour les nombreuses heures de rédaction. Travailler avec vous était toujours un plaisir et vous m'avez fait découvrir le sujet qui aura finalement été celui de mon mémoire. Avec vous, j'ai pu améliorer la qualité de ma rédaction scientifique.

Merci à Renaud, Rox-Anne, Joey, Alexis, Nadia, Pauline, Fanny, Stéphanie qui ont organisé plusieurs projets avec moi ces deux dernières années. Je garderai de très bon souvenirs de tous ces moments avec vous.

Merci à Herman, Antoine, Aram, Florence et tous ceux et celles qui animent le LaCIM et en font un endroit agréable à travailler.

Merci au CRSNG et au FRQNT pour leur soutien financier qui m'a permis de consacrer ces deux dernières années à mon projet de maîtrise.

Finalement, merci à Marielle et Simon, mes parents, qui m'ont toujours encouragé à poursuivre mes rêves et merci à Myllie, mon amoureuse, qui m'a soutenu dans mon projet et mes nombreux questionnements.

TABLE DES MATIÈRES

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ALGORITHMES

RÉSUMÉ

L'étude des sous-arbres induits pleinement feuillus est un nouveau sujet d'intérêt en théorie des graphes. Un sous-graphe induit est un sous-graphe constitué d'un ensemble de sommets et de toutes les arêtes qui les relient. Un sous-graphe induit qui est un arbre est appelé sous-arbre induit. On dit, de plus, qu'il est pleinement feuillu, s'il maximise son nombre de feuilles parmi tous les sous-arbres induits de même taille dans le graphe. L'objectif de ce travail est d'étudier ces objets d'un point de vue algorithmique et de tenter de les caractériser.

Dans un premier temps, on introduit la fonction feuille d'un graphe qui à chaque entier n, associe le nombre maximal de feuilles qu'un sous-arbre induit de taille n peut réaliser, c'est-à-dire le nombre de feuilles d'un sous-arbre induit pleinement feuillu de taille n. On montre que le calcul de la fonction feuille est un problème difficile, puis on introduit une structure de données que l'on utilise dans un algorithme de type séparation et évaluation progressive, pour le calcul de la fonction feuille. La démonstration de l'exactitude de l'algorithme est présentée et des données empiriques sur sa performance sont analysées.

Dans un deuxième temps, on introduit le problème de réalisation de la fonction feuille, qui consiste à décider si une fonction donnée est la fonction feuille d'un certain graphe. On utilise des outils de la combinatoire des mots pour donner des conditions nécessaires afin qu'une fonction soit réalisable. On présente, ensuite, un outil supplémentaire, les mots préfixes normaux. Ces derniers se définissent ainsi : il s'agit des mots binaires dont chaque préfixe contient plus de 1 que tous les facteurs de la même longueur que ce préfixe. Finalement, à l'aide de cet ensemble de mots, on décrit les fonctions feuilles qui peuvent être réalisées par un graphe chenille et on donne une caractérisation des graphes chenilles ayant la même fonction feuille.

Mots-clés : Théorie des graphes, arbres, NP-complet, chenilles, graphes induits

ABSTRACT

Fully leafed induced subtrees

The study of fully leafed induced subtree is a new area of interest in graph theory. An induced subgraph is a subgraph given by a subset of the graph vertices and all the edges that connect those together. An induced subgraph which is a tree is called and induced subtree. Moreover, we say that it is fully leafed if it maximizes its number of leaves among all induced subtrees of the same size in the graph. The purpose this thesis is to study those objects with an algorithmic perspective and to try to characterized them.

First, we introduce the leaf function of a graph. This function maps each integer n to the number of leaves of a fully leafed subtree of size n in the graph. We show that deciding if a graph has a induced subtree with i vertices and ℓ leaves is NP-complete and therefore that computing the leaf function is NP-hard. We then introduce a data structure called graph configuration that is used to iterate over all induced subtrees of graph. We prove an upper bound on the number of leaves that can be reached by extending a subtree. From that bound and the previous iterator, we derive a branch-and-bound algorithm for the computation of the leaf function and analyze the performance.

Second, we introduce the leaf realization problem which consist of finding if a function is the leaf function of a given graph. We use tools from combinatorics on words, in particular the discrete derivative, to give some necessary condition for a function to be realizable. Moreover, we completely characterize the leaf functions that can be realized by a caterpillar graph and give a characterization of all caterpillars with the same leaf function. In order to do this, we define a bijection between caterpillar and prefix normal words, the binary words such that for each length, the prefix is, among all factors of that length, the ones that contains the most 1s.

Keywords : Graph theory, trees, NP-complete, caterpillars, induced subgraphs

INTRODUCTION

En théorie des graphes, l'idée d'étudier des sous-arbres avec beaucoup de feuilles n'est pas nouvelle. Ce genre de question a déjà été étudié pour les arbres couvrants. En effet, depuis les années 1990, des chercheurs s'intéressent aux sous-arbres de ce type, particulièrement ceux comportant un maximum de feuilles. Ils ont entre autres démontrés à l'aide d'algorithme constructif, des bornes inférieures et supérieures du nombre maximal de feuilles d'un arbre couvrant en fonction de son nombre de sommets et son degré minimal (Kleitman et West, 1991). Les applications en réseautique de cette question ont aussi été étudiées dans (Boukerche *et al.*, 2005) où les auteurs développent un protocole écoénergétique de diffusion de l'information dans un réseau sans fil en utilisant un arbre de diffusion avec beaucoup de feuilles. L'étude des sous-arbres induits est aussi un sujet qui a attiré l'attention des chercheurs. Citons, entre autres, le problème IS, consistant à décider s'il existe un sous-arbre induit de plus de i sommets dans un graphe G, qui a été étudié par Erdős, Saks et Sós dans (Erdős *et al.*, 1986). Ces derniers ont, entre autres, démontré que ce problème est NP-complet. Une autre classe de problèmes reliés aux sous-arbres induits est celle des sous-chaînes induites. Le problème *snake-in-the-box* (Abbott et Katchalski, 1988; Goupil *et al.*, 2018), consistant à trouver une sous-chaîne induite de longueur maximale dans les graphes hypercubiques, en est une illustration. Cette dernière question reste d'ailleurs toujours sans réponse malgré l'intérêt qu'elle suscite dans les communautés mathématique et informatique.

Récemment, dans (Blondin Massé *et al.*, 2018; Blondin Massé *et al.*, 2018a), les auteurs se sont intéressés aux polyominos et polycubes arbres, des sous-ensembles

des grilles $\mathbb{Z}^2$ et $\mathbb{Z}^3$ en forme d'arbre, maximisant le nombre de feuilles pour leur taille. Ces travaux ont mené à la découverte de structures intéressantes et inattendues.

Dans ce mémoire, on présente une généralisation de ce problème aux graphes quelconques. Soit $G = (V, E)$ un graphe. Le sous-graphe induit par un sous-ensemble de sommets U noté $G[U]$ est la restriction de G aux sommets de U et aux arêtes de E qui relient des sommets de U. On dit que $G[U]$ est un sous-arbre pleinement feuillu si $G[U]$ est un arbre et qu'il n'existe pas de sous-arbre induit à $|U|$ sommets dans G qui ait plus de feuilles que $G[U]$. Le problème des polyominos et polycubes se décrit alors comme la recherche de sous-arbres pleinement feuillus dans les graphes infinis des réseaux $\mathbb{Z}^2$ et $\mathbb{Z}^3$. Ajoutons que l'on nomme *fonction feuille* de G, la fonction L_G qui, à chaque taille de sous-arbre, associe le nombre de feuilles d'un *sous-arbre pleinement feuillu* de cette taille dans G.

Dans le chapitre 1, on introduit les définitions de base sur les graphes pour poursuivre avec celles sur les sous-arbres pleinement feuillus, ainsi que le calcul des fonctions feuilles de certaines classes de graphe. Le chapitre se poursuit avec une brève introduction à la combinatoire des mots, ainsi qu'à la théorie de la complexité. On y aborde l'analyse asymptotique d'algorithme, la théorie la NP-complétude et l'analyse de générateur. En particulier, on procède à une réduction pour démontrer que le problème de décision IS décrit précédement est NP-complet.

Dans le chapitre 2, on s'attaque au problème du calcul de la fonction feuille. On démontre d'abord que décider si un graphe possède un sous-arbre induit à i sommets et ℓ feuilles est NP-complet, en procédant à une réduction à partir du problème IS posé par Erdős. On présente ensuite une structure de données appelée *configuration de graphe* et un algorithme de génération récursif permettant de parcourir tous les sous-arbres induits d'un graphe. Finalement, on établit une

borne sur le nombre de feuilles possibles pour une extension d'une configuration appelée *potentiel de feuille* et on l'utilise pour construire un algorithme de type séparation et évaluation afin de calculer la fonction feuille. On analyse finalement les performances de l'algorithme. Les résultats présentés dans ce chapitre sont publiés dans (Blondin Massé *et al.*, 2018c) dont je suis coauteur et l'implémentation des algorithmes est disponible dans (Blondin Massé et Nadeau, 2017).

Dans le chapitre 3, on aborde le problème de réalisation de la fonction feuille. Ce dernier qui est conceptuellement l'inverse du calcul de la fonction feuille est inspiré du problème des suites graphiques présenté dans (Erdős et Gallai, 1961). Le problème des suites graphiques consiste à déterminer si, étant donnée une suite d'entiers $(x_1, \ldots, x_n)$ il existe un graphe avec les sommets $1, \ldots, n$ tels que le sommet i soit de degré x_i. Plusieurs chercheurs ont démontré que l'on pouvait résoudre ce problème en temps polynomial, que ce soit en vérifiant une système d'inégalités ou à l'aide d'un algorithme constructif (Havel, 1955; Hakimi, 1962; Erdős et Gallai, 1961). Le problème de réalisation étudié dans ce texte consiste à déterminer s'il existe un graphe qui réalise une certaine fonction feuille. On introduit le *mot feuille* d'un graphe qui est la dérivée discrète de sa fonction feuille. On décrit l'alphabet des mots feuilles et on caractérise les mots feuilles des arbres. Les *mots préfixes normaux*, c'est-à-dire les mots binaires dont les préfixes sont les facteurs contenant le plus de 1, introduit dans (Burcsi *et al.*, 2014), permettent de caractériser complètement les fonctions qui peuvent être réalisées par les graphes chenilles introduits dans (Harary et Schwenk, 1973). On démontre que les fonctions pouvant être réalisées par les chenilles sont celles dont la dérivée discrète est un mot préfixe normal. Une bijection entre mots binaires et chenilles permet de caractériser les chenilles qui ont la même fonction feuille. Les résultats de ce chapitre sont publiés dans (Blondin Massé *et al.*, 2018) dont je suis un des coauteurs.

CHAPITRE I

PRÉLIMINAIRES ET SOUS-ARBRES PLEINEMENT FEUILLUS

Dans ce chapitre, on pose les définitions et notations des différentes théories utilisées dans ce mémoire. On introduit aussi les définitions précises et quelques résultats concernant les sous-arbres induits pleinement feuillus.

1.1 Terminologie de base sur les graphes

On débute avec les notions de base en théorie des graphes. La terminologie présentée ici est largement inspirée de (Bollobás, 1998).

Définition 1.1. Un graphe G est un couple (V, E) où V est l'ensemble des *sommets* du graphe et E est un ensemble de paires non ordonnées de sommets. Les éléments de E sont appelés les *arêtes* du graphe.

Il est d'usage de représenter les graphes sur le plan en plaçant un point pour chaque sommet et une ligne pour chaque arête. On visualise en effet beaucoup plus facilement le graphe

$$(\{0,1,2,3,4\}, \{\{0,1\},\{0,2\},\{0,3\},\{1,2\},\{1,3\},\{2,3\},\{2,4\},\{3,4\}\})$$

en regardant la figure 1.1.

On dit que deux sommets u et v sont *adjacents* ou *voisins* si $\{u, v\}$ est dans l'ensemble des arêtes. On dénote par $N(v)$ (ou $G.\textsc{Voisins}(v)$ dans le pseudo-

code) l'ensemble des voisins d'un sommet v. Le nombre de voisins d'un sommet est son *degré* et est noté $\deg(v)$. On appelle *taille* d'un graphe son nombre de sommets et on la dénote $|G|$.

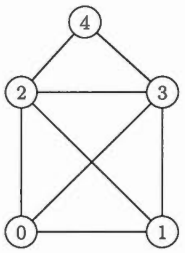


Figure 1.1 Représentation d'un graphe dans le plan.

Une *chaîne* dans un graphe est une suite de sommets $\gamma = (v_0, v_1, \ldots, v_k)$ telle que $\{v_i, v_{i+1}\}$ soit une arête pour $0 \leq i < k$. Le sommet v_0 est la *source* de la chaîne et v_k en est le *but*. Deux sommets u et v sont dit *connectés* s'il existe une chaîne de source u et de but v. La *longueur* d'une chaîne est le nombre d'arêtes qu'elle parcourt, et la *distance* entre deux sommets la longueur de la plus courte chaîne les reliant. Un graphe est dit *connexe* si toute paire de sommets est connectée. La relation « être connecté » forme une relation d'équivalence sur les sommets. Une classe de cette relation est appelée *composante connexe* du graphe. Un *cycle* est une chaîne contenant au moins 3 sommets et dont la source et le but coïncident. Un graphe est dit *acyclique* s'il ne contient aucun cycle. En combinant la propriété d'être acyclique et connexe, on obtient une classe de graphes importante pour la suite.

Définition 1.2. Un *arbre* est un graphe acyclique et connexe. Dans un arbre T, une *feuille* est un sommet de degré 1. On dénote $|T|_{\mathscr{D}}$ le nombre de feuilles de T.

La figure 1.2 illustre l'arbre

$$T = (\{1,2,3,4,5,6,7\}, \{\{1,2\},\{1,3\},\{1,5\},\{3,4\},\{3,7\},\{5,6\}\})$$

dont les feuilles sont les sommets 2, 4, 6 et 7.

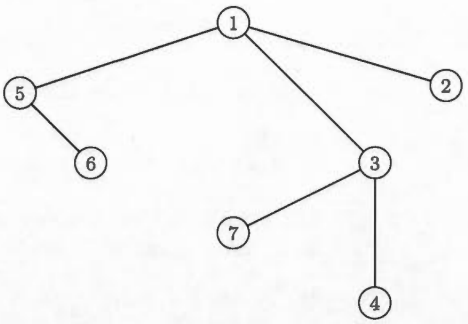


Figure 1.2 Arbre à 7 sommets et 4 feuilles.

Au chapitre 3, on porte notre attention sur un sous-ensemble particulier des arbres : les graphes chenilles. Ces derniers ont été introduits par Harary et Schwenk dans (Harary et Schwenk, 1973) où ils prouvent de deux façons une formule pour compter le nombre de chenilles de taille fixée.

Définition 1.3. Une *chenille* est un arbre de taille au moins 3 contenant un chaîne γ tel que toutes les feuilles soient adjacentes à un sommet de γ.

De façon équivalente, une chenille est un arbre qui lorsqu'on lui retire toutes ses feuilles devient une simple chaîne. Cette chaîne est appelée *colonne* de la chenille.

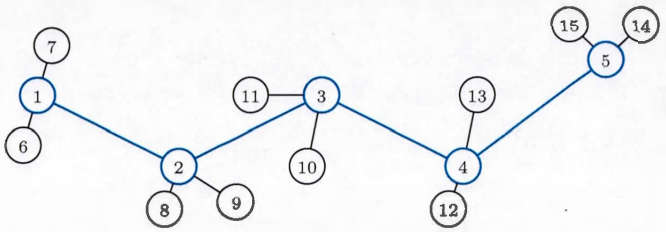


Figure 1.3 Une chenille avec sa colonne en bleu.

Une dernière notion importante est celle de sous-graphe. Pour un graphe $G = (V, E)$, on dit qu'un graphe $G' = (V', E')$ est un *sous-graphe* de G si $V' \subseteq V$ et

$E' \subseteq E \cap \mathcal{P}_2(V')$ où $\mathcal{P}_2(V') = \{\{x, y\} : x, y \in V'\}$. En particulier, on s'intéresse aux sous-graphes induits.

Définition 1.4. Soit $G = (V, E)$ un graphe et U un sous-ensemble de V. Le sous-graphe induit par U sur G est $G[U] = (U, E \cap \mathcal{P}_2(U))$.

Un sous-graphe induit est donc un sous-graphe défini par un sous-ensemble de sommets et dont les arêtes sont toutes celles qui relient ces sommets dans le graphe de départ. En particulier, un *sous-arbre induit* d'un graphe G est un sous-graphe induit de G qui est un arbre. La figure 1.4 (a) montre un sous-graphe induit qui n'est pas un sous-arbre induit car les sommets 7, 8 et 9 induisent un cycle. La figure 1.4 (b) illustre un sous-arbre induit.

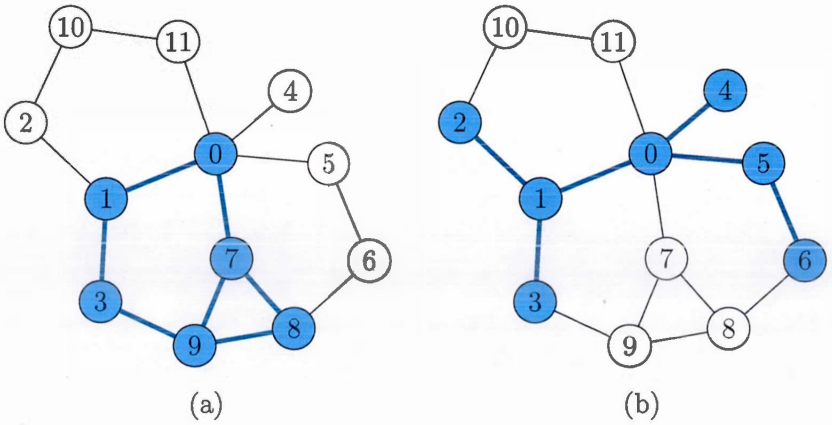


Figure 1.4 En bleu, le (a) sous-graphe induit par $\{0, 1, 3, 7, 8, 9\}$ et (b) sous-arbre induit par $\{0, 1, 2, 3, 4, 5, 6\}$.

1.2 Fonction feuille

On veut étudier les sous-arbres induits qui contiennent beaucoup de feuilles. Précisément, on s'intéresse à ceux qui contiennent le plus de feuilles pour leur taille.

Définition 1.5. Soit G un graphe. Un *sous-arbre induit pleinement feuillu* de G est un sous-arbre induit de G tel que tout sous-arbre induit de G de même taille n'ait pas plus de feuilles.

Par exemple, le sous-arbre induit de la figure 1.4 (b) n'est pas pleinement feuillu car il contient 4 feuilles alors que les sommets $\{0, 1, 4, 5, 6, 7, 11\}$ induisent un sous-arbre à 5 feuilles. Ce dernier est, par ailleurs, un sous-arbre induit pleinement feuillu. En effet, pour trouver un sous-arbre induit à 7 sommets et 6 feuilles, il faudrait que le sous-arbre induit n'ait qu'un seul sommet interne et conséquement un sommet de degré 6. Or ceci est impossible, car le degré maximal dans le graphe de la figure 1.4 est 5.

Lorsqu'on considère le nombre de feuilles d'un sous-arbre induit pleinement feuillu, on parle alors de fonction feuille.

Définition 1.6. La *fonction feuille* $L_G : \{0, 1, \ldots, |G|\} \to \mathbb{N} \cup \{-\infty\}$ d'un graphe G, est la fonction qui à i associe le nombre de feuilles d'un sous-arbre pleinement feuillu de taille i. Autrement dit,

$$L_G(i) = \max \{|T|_{\mathscr{L}} : T \text{ est sous-arbre induit de } G \text{ et } |T| = i\}.$$

Par convention on considère que $\max \emptyset = -\infty$.

La fonction feuille vaut donc $-\infty$ si et seulement si le graphe ne contient pas de sous-arbre induit de taille i. De plus, lorsque i vaut 0 ou 1, la fonction feuille vaut systématiquement 0 puisqu'il ne peut pas y avoir de sommet de degré 1. On peut facilement calculer la fonction feuille pour plusieurs familles classiques de graphes. On remarque d'abord que pour les graphes complets $K_n = (\{1, \ldots, n\}, \{\{i, j\} : i \neq j\})$ avec $n \geq 3$, la fonction feuille est

$$L_{K_n}(i) = \begin{cases} 0 & \text{si } i = 0,1 \\ 2 & \text{si } i = 2 \\ -\infty & \text{sinon .} \end{cases}$$

En effet, si on prend deux sommets, ils sont reliés par une arête et donc forment un arbre avec deux feuilles. Dès que l'on prend trois sommets ou plus, on forme un 3-cycle et le sous-graphe induit n'est pas acyclique.

Pour les cycles $C_n = (\{1, 2, \ldots, n\}, \{\{i, i+1\} : 1 \leq i < n\} \cup \{\{1, n\}\})$, avec $n \geq 3$, la fonction feuille est

$$L_{C_n}(i) = \begin{cases} 0 & \text{si } i = 0,1 \\ -\infty & \text{si } i = n \\ 2 & \text{sinon .} \end{cases}$$

En effet, si on prend n sommets alors le graphe induit est un cycle. On ne peut donc pas former d'arbre induit par n sommets. Dans les autres cas, il suffit de prendre une chaîne de longueur i dans le graphe pour obtenir deux feuilles et il est clair qu'on ne peut pas faire mieux.

Pour la roue W_n construite à partir de C_n en ajoutant un sommet 0 (le moyeu) et des arêtes entre tous les sommets de C_n et 0, on a pour $n \geq 3$ que

$$L_{W_n}(i) = \begin{cases} 0 & \text{si } i = 0,1 \\ 2 & \text{si } i = 2 \\ i-1 & \text{si } 3 \leq i \leq \left\lfloor \frac{n}{2} \right\rfloor + 1 \\ 2 & \text{si } \left\lfloor \frac{n}{2} \right\rfloor + 2 \leq i \leq n-1 \\ -\infty & \text{si } i = n, n+1. \end{cases}$$

Tout d'abord, les cas $0, 1, 2, n, n+1$ sont évidents. Si $i-1$ est plus petit ou égal à

$\frac{n}{2}$ alors on peut prendre le moyeu et $i-1$ sommets non adjacents dans C_{n-1}. On a alors un arbre à $i-1$ feuilles. Dans les autres cas, on ne prend pas le moyeu et on prend une chaîne dans C_n de longueur i.

Pour les graphes bipartis complets

$$K_{n,m} = (\{1, 2, \ldots, m+n\}, \{\{i,j\} : 1 \leq i \leq n, n+1 \leq j \leq m+n\})$$

la fonction feuille est

$$L_{K_{n,m}}(i) = \begin{cases} 0 & \text{si } i = 0, 1 \\ 2 & \text{si } i = 2 \\ i-1 & \text{si } 2 < i \leqslant \max(n,m)+1 \\ -\infty & \text{sinon .} \end{cases}$$

En effet, on remarque que dès que l'on prend deux sommets dans chacune des parties du graphe, on crée automatiquement un cycle. Pour avoir un sous-graphe induit qui soit un arbre, il faut donc qu'il y ait une des deux parties où l'on prend un seul sommet. Le reste en découle facilement.

Le calcul des fonctions feuilles à aussi fait l'objet de publications. On retrouve dans (Abdenbi *et al.*, 2018) des équations décrivant la fonction feuille des graphes séries-parrallèles. Les sous-arbres pleinement feuillus de certains graphes infinis ont aussi déjà été étudiés. Dans (Blondin Massé *et al.*, 2018) et (Blondin Massé *et al.*, 2018b), les auteurs étudient les polyominos, polycubes, polyhexes et polyamants arbres pleinement feuillus. On reformule ici deux de leurs résultats dans notre terminologie.

Théorème 1.7. *La fonction feuille du graphe dont l'ensemble des sommets est* $\mathbb{Z}^2$ *et l'ensemble des arêtes est*

$$\bigcup_{(i,j)\in\mathbb{Z}^2} \{\{(i,j),(i+1,j)\},\{(i,j),(i,j+1)\}\},$$

c'est-à-dire la grille des entiers munie de la relation de 4-adjacence, est donnée par la récurrence suivante :

$$L_{\mathbb{Z}^2}(i) = \begin{cases} 0 & si\ i = 0, 1 \\ 2 & si\ i = 2 \\ i - 1 & si\ i = 3, 4, 5 \\ L_{\mathbb{Z}^2}(n-4) + 2 & si\ i \geq 6. \end{cases}$$

Théorème 1.8. *La fonction feuille du graphe dont l'ensemble des sommets est* $\mathbb{Z}^3$ *et l'ensemble des arêtes est*

$$\bigcup_{(i,j,k)\in\mathbb{Z}^2} \{\{(i,j,k),(i+1,j,k)\},\{(i,j,k),(i,j+1,k),\{(i,j,k),(i,j,k+1)\}\}\},$$

c'est-à-dire $\mathbb{Z}^3$ *muni de la relation de 6-adjacence, est donnée par la récurrence suivante :*

$$L_{\mathbb{Z}^3}(i) = \begin{cases} 0 & si\ i = 0, 1 \\ f(i) + 1 & si\ i = 6, 7, 13, 19, 25 \\ f(i) & si\ 2 \leq i \leq 40 \quad et \quad n \neq 6, 7, 13, 19, 25 \\ f(i-41) + 28 & si\ 41 \leq i \leq 81 \\ L_{\mathbb{Z}^3}(i-41) + 28 & si\ i \geq 82, \end{cases}$$

$$avec \quad f(i) = \begin{cases} \lfloor (2i+2)/3 \rfloor & si\ 0 \leq i \leq 11 \\ \lfloor (2i+3)/3 \rfloor & si\ 12 \leq i \leq 27 \\ \lfloor (2i+4)/3 \rfloor & si\ 28 \leq i \leq 40. \end{cases}$$

1.3 Combinatoire des mots

On aborde maintenant la terminologie de base sur les mots. Pour une introduction détaillée, voir (Lothaire, 1997). Un *mot* est une suite finie de symboles (appelés *lettres*) $w_1 w_2 \ldots w_n$ où chaque symbole w_i est pris dans un ensemble fini A appelé *alphabet*. On note $\mathrm{Alph}(w)$ l'ensemble des lettres d'un mot w. On nomme *longueur* d'un mot, et on note $|w|$, le nombre de lettres de w. Pour une lettre $a \in A$, on note $|w|_a$ le nombre d'occurrences de la lettre a dans le mot w. Deux mots w et w'

sont dit *abéliens équivalents* si $|w|_a = |w'|_a$ pour tout $a \in A$. Le mot ne contenant aucun symbole est appelé *mot vide*. On note ce dernier ε. On appelle *langage* un ensemble de mots. En particulier, le langage contenant tous les mots de longueur finie sur l'alphabet A est noté A^* et celui des mots de longueur i est noté A^i. Pour deux mots $w = w_1 \dots w_n$ et $u = u_1 \dots u_m$ on définit la *concaténation* de w et u comme le mot $w_1 \dots w_n u_1 \dots u_m$ et on la note $w \cdot u$ ou simplement wu. L'ensemble A^* muni de l'opération de concaténation forme un monoïde d'élément neutre ε.

Si $w = pus$ on dit alors que u est un facteur de w. De plus, si $p = \varepsilon$, on dit que u est préfixe de w et si $s = \varepsilon$, on dit que u est suffixe de w. On note $\text{pref}_i(w)$ (resp. $\text{suff}_i(w)$) le préfixe (resp. suffixe) de longueur i de w. Finalement, on note $Fact(w)$ l'ensemble des facteurs de w et $Fact_i(w) = Fact(w) \cap A^i$ l'ensemble des facteurs de longueur i.

Un langage particulier que l'on utilise au chapitre 2 est celui des mots bien parenthésés.

Définition 1.9. On dit qu'un mot de $\{[,]\}^*$ est *bien parenthésé* si l'on peut obtenir le mot vide en supprimant successivement des facteurs $[]$.

Ce type de mots est bien connu en informatique théorique ainsi qu'en théorie des langages. On y fait aussi souvent référence sous le nom de *mot de Dyck*. Par exemple, le mot $[[][[]]]$ est un mot bien parenthésé mais $[[][$ ne l'est pas. Remarquons qu'un mot de Dyck est toujours de longueur paire puisqu'on retrouve le mot vide en enlevant un nombre pair de lettres. De plus, lorsqu'on supprime un facteur $[]$ d'un mot de Dyck, on retrouve aussi un mot de Dyck.

1.4 Théorie de la complexité

Cette section se veut une brève introduction à la théorie de la complexité. Seules les notions essentielles pour la suite du texte sont abordées. Pour une introduction

détaillée, le lecteur peut consulter (Cormen *et al.*, 2009).

1.4.1 Analyse d'algorithme

Lors de la conception d'algorithmes, il est important de pouvoir déterminer si un algorithme est plus rapide ou efficace qu'un autre. Pour ce faire, on utilise l'analyse asymptotique. Grossièrement, on cherche à comparer la croissance du temps nécessaire pour exécuter l'algorithme en fonction de la taille de l'entrée.

Par exemple, imaginons que l'on veuille résoudre le problème suivant :

Problème 1.10. Étant donné L une liste triée de n entiers, est-ce que L contient un entier donné x ?

On compare deux approches pour résoudre celui-ci : la recherche linéaire et la recherche binaire. Pour bien comprendre les approches, on les illustre sur la liste suivante :

$$L = [\mathbf{0}, \mathbf{1}, \mathbf{2}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{5}, \mathbf{5}, \mathbf{5}, \mathbf{7}, \mathbf{7}, \mathbf{8}, \mathbf{9}, \mathbf{9}, \mathbf{10}]$$

L'algorithme de recherche linéaire consiste à regarder l'un après l'autre les éléments de la liste et vérifier si l'élément recherché est le même. Pour notre exemple, il faut donc regarder les 13 premières valeurs pour vérifier si la liste contient un 8. En général, pour une liste de longueur n on pourrait avoir à faire jusqu'à n comparaisons. Remarquons que cette approche n'utilise pas l'hypothèse que la liste est triée.

Pour la recherche binaire, on compare la valeur de l'élément au milieu de la liste avec l'élément recherché x. Si cet élément n'est pas x, comme la liste est triée, on peut alors déterminer s'il doit se trouver à la gauche ou à la droite de cet élément. On élimine alors au moins la moitié des valeurs restantes et on répète l'algorithme avec la moitié de la liste dans laquelle x peut se trouver. Ainsi, à chaque fois que

l'on regarde une valeur on élimine la moitié des valeurs restantes. Par exemple sur L, on regarde seulement les valeurs $5, 7, 9, 8$ pour rechercher l'élément 8. La figure 1.5, illustre les étapes d'une recherche binaire dans L.

$$[0, 1, 2, 2, 3, 4, 5, 5, \mathbf{5}, \mathbf{5}, \mathbf{7}, \mathbf{7}, \mathbf{8}, \mathbf{9}, \mathbf{9}, \mathbf{10}]$$
$$[0, 1, 2, 2, 3, 4, 5, 5, 5, 5, 7, 7, \mathbf{8}, \mathbf{9}, \mathbf{9}, \mathbf{10}]$$
$$[0, 1, 2, 2, 3, 4, 5, 5, 5, 5, 7, 7, \mathbf{8}, 9, 9, 10]$$
$$[0, 1, 2, 2, 3, 4, 5, 5, 5, 5, 7, 7, 8, 9, 9, 10]$$

Figure 1.5 Étapes de la recherche binaire de 8 dans la liste triée L. Les valeurs éliminées sont en rouge et les valeurs comparées en bleu.

Avec cette stratégie, comme on élimine à chaque fois la moitié des valeurs restantes, on examine donc au plus $\lceil \log_2(n) \rceil$ valeurs pour une liste de longueur n.

Pour formaliser l'intuition de « environ au plus », on introduit la notation $\mathcal{O}$.

Définition 1.11. Soit $f : \mathbb{N} \to \mathbb{R}_+$, alors $\mathcal{O}(f)$ est l'ensemble de fonctions

$$\{g : \mathbb{N} \to \mathbb{R} \ : \ \exists n_0, c \in \mathbb{R}_+ \text{ tel que } 0 \leq g(n) \leq cf(n) \quad \forall n \geq n_0\}$$

Intuitivement, on dit qu'une fonction g est dans $\mathcal{O}(f)$ si f borne sa croissance à une constante c près. Ainsi on retrouve dans $\mathcal{O}(n^3)$ tous les polynômes de degré inférieur à 3, mais pas le polynôme n^4. On dit, par exemple, que le nombre de comparaisons à effectuer pour la recherche linéaire dans une liste de taille n est dans $\mathcal{O}(n)$ alors qu'elle est dans $\mathcal{O}(\log_2(n))$ pour la recherche binaire. Comme le logarithme croit significativement plus lentement qu'une fonction linéaire, on conclut que la recherche binaire est plus efficace car elle demande beaucoup moins d'opérations. En général, on utilise la notation asymptotique $\mathcal{O}$ pour borner le « nombre d'étapes » (affectations/lectures de valeur, opérations mathématiques

de base) d'un algorithme. Si le nombre d'étapes d'un algorithme est dans $\mathcal{O}(f)$, on dit alors que la *complexité temporelle* de l'algorithme est $\mathcal{O}(f)$. On dit qu'un algorithme est *polynomial* si sa complexité temporelle est dans $\mathcal{O}(n^k)$ pour une certaine constante entière k. On utilise aussi parfois cette notation pour estimer l'espace requis (le nombre de valeurs à stocker) d'un algorithme. On parle alors de *complexité spatiale.*

1.4.2 NP-complétude

Dans la section précédente, on a abordé la complexité des algorithmes. On généralise ici cette définition aux problèmes. On dit qu'un problème P est *polynomial* s'il existe un algorithme avec une complexité temporelle polynomiale permettant de résoudre le problème.

Il existe un type particulier de problèmes que l'on appelle *problème de décision.* Intuitivement, ces problèmes sont ceux dont la réponse est « oui » ou « non ». Par exemple, déterminer si un nombre est premier est un problème de décision, mais trouver la factorisation d'un nombre en facteurs premiers n'en est pas un. Aussi, dans un graphe, trouver la plus courte chaîne entre deux sommets n'est pas un problème de décision, mais déterminer s'il existe une chaîne de longueur k entre ces deux derniers en est un. Formellement, étant donné un ensemble E et un sous-ensemble $S \subseteq E$, un *problème de décision* consiste à déterminer, étant donné $x \in E$, si $x \in S$. On nomme *instance* d'un problème un élément de E. Si x est élément de S alors x est une *instance positive* du problème.

L'ensemble des problèmes de décision est organisé en fonction de la complexité des problèmes. Deux des classes les plus connues sont les classes P et NP.

Définition 1.12. La classe P (pour *Polynomial*) est l'ensemble des problèmes de décision pour lesquels il existe un algorithme polynomial permettant de le

résoudre.

On considère normalement qu'un problème dans P est un problème traitable sur des ordinateurs puisque le temps de calcul requis ne croit pas « trop vite » lorsque la taille des instances augmente.

Pour un problème de décision, un *algorithme de vérification* est un algorithme qui prend en entrée x (une instance du problème) et c (un *certificat*). Il doit satisfaire la propriété suivante : pour toute instance x, il existe un certificat c tel que l'algorithme retourne vrai si et seulement si x est une instance positive du problème. Par exemple, pour le problème consistant à décider si un nombre est composé, on peut avoir comme certificat c une paire de nombres et l'algorithme serait de vérifier si leur produit donne x.

Définition 1.13. La classe NP (pour *Non-deterministic Polynomial*) est l'ensemble des problèmes de décision pour lesquels il existe un algorithme de vérification polynomial utilisant un certificat de taille polynomiale.

Par exemple, tous les problèmes dans P sont dans NP, car on peut utiliser comme algorithme de vérification l'algorithme polynomial permettant de résoudre le problème. Toutefois, il n'est pas du tout clair que tout problème dans NP soit dans P. En effet, pour plusieurs problèmes dans NP, il n'y a pas, à ce jour, d'algorithme connu pour les résoudre en temps polynomial. Par exemple, déterminer si un graphe possède une clique de taille k (un sous-graphe induit à k sommets qui est isomorphe au graphe complet) est dans NP, mais on ne sait pas montrer qu'il est ou qu'il n'est pas dans P. Cette question, connue sous le nom « P vs NP », est posée pour la première fois dans l'article *The Complexity of Theorem-proving Procedure* (Cook, 1971). Toujours irrésolu malgré l'intérêt qu'il suscite, ce problème fait d'ailleurs partie des sept problèmes du millénaire du *Clay Mathematics Institute* (Jaffe, 2006).

Certains problèmes NP jouent un rôle particulier en théorie de la complexité et sont appelés NP-complets. Intuitivement, on pense à ces problèmes comme étant les problèmes les plus difficiles de la classe NP. Formellement, on définit un préordre $\preceq$ sur les problèmes comme suit :

Définition 1.14. Soit P et Q des problèmes de décision. On dit que $P \preceq Q$ s'il existe un algorithme polynomial pour résoudre P en supposant que l'on puisse résoudre Q en temps constant.

On peut aisément vérifier que la relation $\preceq$ est réflexive et transitive. On dit que $P \equiv Q$ lorsque $P \preceq Q$ et $Q \preceq P$. On constate alors que $\preceq$ définit une relation d'ordre sur l'ensemble des problèmes de décision dans NP quotienté par $\equiv$. Les problèmes NP-complets sont les problèmes de la classe d'équivalence qui est le maximum de cette relation d'ordre.

Définition 1.15. P est un problème NP-complet si P est dans NP et pour tout problème Q dans la classe NP on a $Q \preceq P$.

On sait que l'ensemble des problèmes NP-complets n'est pas vide. En effet, le premier problème pour lequel on a démontré sa NP-complétude est le problème de satisfaisabilité d'expressions logiques.

Théorème 1.16 (Cook-Levin, 1971). *Le problème de satisfaisabilité d'une expression logique sous forme normale conjonctive (SAT-CNF) est NP-complet.*

Précisément, ce problème consiste à déterminer s'il existe une distribution de valeurs de vérité pour n variables booléennes $\{P_1, \ldots, P_n\}$ qui satisfasse une expression logique de la forme $\bigwedge_{i=1}^{m} \left(\bigvee_{j=1}^{k} \ell_{i,j} \right)$ avec $\ell_{i,j} \in \{P_1, \neg P_1, \ldots, P_n, \neg P_n\}$.

Les problèmes NP-complets étant plus difficiles que tous les problèmes dans NP, s'il existe un algorithme polynomial pour résoudre un seul des problèmes NP-

complets alors tous les problèmes de NP peuvent être résolus en temps polynomial. Ainsi, trouver un algorithme polynomial résolvant un problème NP-complet démontrerait que $P = NP$. À ce jour, on ne connait aucun algorithme polynomial permettant de résoudre un problème NP-complet.

1.4.3 Réduction

La technique usuelle pour démontrer qu'un problème est NP-complet est d'utiliser une *réduction*. Supposons que l'on veuille montrer qu'un problème P_1 est NP-complet. On doit d'abord vérifier que le problème est dans NP. Pour ce faire, il suffit de donner un algorithme de vérification. On s'assure ensuite que $Q \preceq P_1$ pour tout problème Q dans NP. Pour ce faire, il suffit de trouver un problème P_2 qui soit NP-complet et tel que $P_2 \preceq P_1$. Par transitivité, on trouve alors que $Q \preceq P_1$ pour tout problème NP-complet Q. Pour démontrer que $P_2 \preceq P_1$ on procède en général comme suit :

1. on donne une transformation f des instances de P_2 en instances de P_1,
2. on s'assure que cette transformation s'effectue en temps polynomial,
3. on s'assure que pour une instance x de P_2, $f(x)$ est une instance positive de P_1 si et seulement si x est une instance positive de P_2.

On a alors démontré que $P_2 \preceq P_1$ car si on suppose que l'on peut résoudre P_1 en temps constant, on peut alors résoudre P_2 en temps polynomial en appliquant la transformation f à notre instance puis en vérifiant que l'on obtient une instance positive de P_1.

On peut, par exemple, montrer que le problème IS (Induced Subtree) consistant à déterminer si un graphe contient un sous-arbre induit à plus de i sommets est NP-complet. Pour ce faire, on utilise la réduction proposée dans (Erdős *et al.*, 1986) à partir du problème NP-complet INDEPSET (Independant Set). Ce dernier consiste à déterminer si un graphe contient un *ensemble indépendant* à k sommets

(c'est-à-dire un ensemble de k sommets dont aucune paire n'est adjacente). Pour une preuve de la NP-complétude de INDEPSET voir (Garey et Johnson, 1979).

Proposition 1.17 (Erdős, Saks et Sòs, 1986). *Le problème* IS *est NP-complet.*

Démonstration. On vérifie d'abord que IS est bien dans NP. Pour ce faire, on prend comme certificat un sous-ensemble des sommets et l'algorithme de vérification consiste à vérifier que ce sous-ensemble n'induit pas de cycle et est connexe. Cette vérification s'effectue bien en temps polynomial puisqu'il suffit de faire un parcours de graphe en se restreignant aux sommets du sous-ensemble.

Pour une instance de INDEPSET, qui est une paire (G, i) avec G un graphe et i un entier, on associe une instance $f(G, i)$ du problème IS. Pour ce faire on construit le graphe H en ajoutant un sommet spécial adjacent à tous les sommets de G. On attache ensuite une chaîne de longueur $|G|$ au sommet spécial. Cette transformation est illustrée à la figure 1.6. On définit alors $f(G, i) = (H, |G|+i+1)$. La transformation f s'effectue bien en temps polynomial puisqu'on ajoute $|G|+1$ sommets et $2|G|$ arêtes.

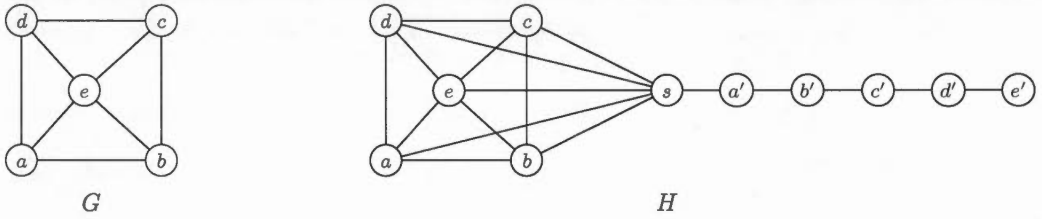


Figure 1.6 Transformation polynomiale d'une instance de INDEPSET en instance de IS.

De plus, si (G, i) est une instance positive de INDEPSET, c'est-à-dire si G possède un ensemble indépendant de taille i, alors H possède un sous-arbre induit de taille $|G| + i + 1$. Pour le trouver, on prend tous les sommets de la chaîne, et les

i sommets de l'ensemble indépendant. Finalement, si H contient un sous-arbre induit à $|G| + i + 1$ sommets alors au moins i de ces sommets doivent être des sommets de G. Aussi, comme le sous-arbre contient au moins un sommet de la chaîne, le sommet spécial doit être inclus dans le sous-arbre. Ainsi, aucun des i sommets choisis dans G ne peut être adjacent sans qu'il n'y ait de cycle. On a donc que ces i sommets forment un ensemble indépendant. □

1.4.4 Générateur et complexité amortie

L'analyse d'algorithmes s'effectue un peu différemment lorsqu'on s'intéresse aux *générateurs*, des algorithmes permettant de parcourir tous les éléments d'un ensemble. Une stratégie utilisée pour ce type d'algorithme est d'utiliser la récursivité. Pour une stratégie récursive, on sépare en plus petits problèmes. Par exemple pour engendrer les sous-ensembles de S on utilise le fait que $\mathcal{P}(S)$ est l'union disjointe de $\mathcal{P}(S\backslash\{x\})$ et $\{\{x\} \cup S' : S' \in \mathcal{P}(S\backslash\{x\})\}$. La stratégie récursive est celle qui sera utilisée au chapitre 2.

On peut aussi commencer avec un élément de l'ensemble que l'on veut générer et utiliser une transformation bien choisie qui permet de passer d'un élément à l'autre. On utilise une telle stratégie pour générer tous les mots sur $\{a, b\}$ de longueur n. On commence avec $w_1 \dots w_n = a \dots a$. À partir d'un mot, pour trouver le mot suivant, on part de la droite et on transforme tout les b en a jusqu'à ce qu'on tombe sur un a. On termine en remplaçant ce a par un b. La figure 1.1 liste les mots que l'on obtient en appliquant successivement cette procédure pour $n = 4$. À partir de l'exemple, on peut facilement se convaincre qu'en procédant ainsi on génère tous les mots d'une longueur fixée.

L'analyse de la complexité totale d'un algorithme de génération est souvent peu intéressante, d'autant plus qu'il est courant que le nombre d'éléments dans l'ensemble soit exponentiel. Ainsi, on analyse plus souvent le délai et l'initialisation

	$w_1w_2w_3w_4$	nombre d'opérations		$w_1w_2w_3w_4$	nombre d'opérations
0	$aaaa$	1	8	$baaa$	1
1	$aaab$	2	9	$baab$	2
2	$aaba$	1	10	$baba$	1
3	$aabb$	3	11	$babb$	3
4	$abaa$	1	12	$bbaa$	1
5	$abab$	2	13	$bbab$	2
6	$abba$	1	14	$bbba$	1
7	$abbb$	4	15	$bbbb$	

Tableau 1.1 Génération des mots binaires de 4 lettres. La colonne « nombre d'opérations » indique le nombre de lettres qui sont modifiées pour passer au mot suivant. En bleu, les lettres qui sont modifiées pour passer au mot suivant.

du générateur. Le *délai* d'un générateur est le temps requis pour trouver l'élément suivant alors que *l'initialisation* est le temps requis pour obtenir un premier élément. Si l'on tente d'analyser le temps nécessaire pour obtenir le mot suivant dans notre générateur de mots, on observe que peu import la longueur, il y a toujours un moment où toutes les lettres doivent être changées (du 7 au 8 dans notre exemple). Conséquemment, dans le pire cas, pour obtenir le mot suivant on doit changer toutes les lettres. Ainsi, le délai est dans $\mathcal{O}(n)$ où n est la longueur des mots générés. Toutefois, pour la plupart des mots, on ne change que peu de lettres. Pour en tenir compte dans notre analyse du délai, on introduit l'analyse de complexité amortie.

Définition 1.18. Soit G un générateur dont la somme des délais est dans $\mathcal{O}(f(n))$. On dit alors que le *délai amorti* est en $\mathcal{O}(f(n)/m)$ où m est le nombre d'éléments engendrés par le générateur.

L'intuition derrière cette définition est qu'on effectue la moyenne des délais pour obtenir chacun des éléments. Cette façon de faire l'analyse de complexité amortie est appelée la méthode agrégée. Calculons le nombre total de changements de lettre

pour notre exemple de générateur de mots. On remarque que w_1 n'est modifié qu'une seule fois, w_2 est modifié trois fois, w_3 sept fois et ainsi de suite. En général, w_i est modifié $2^i - 1$ fois. Ainsi, pour engendrer tous les mots de longueur n on effectue $\sum_{i=1}^{n}(2^i - 1) = 2^{n+1} - 2 - n$ changements de lettre. Le délai amorti du générateur est donc dans $\mathcal{O}\left(\frac{2^{n+1}-2-n}{2^n}\right) = \mathcal{O}(1)$.

CHAPITRE II

CALCUL DE LA FONCTION FEUILLE D'UN GRAPHE

Dans ce chapitre, on donne un algorithme général pour le calcul de la fonction feuille d'un graphe simple quelconque. On commence par montrer que décider si un graphe possède un sous-arbre induit à i sommets et ℓ feuilles est un problème NP-complet. Par la suite, on construit une structure de données nommée *configuration*, que l'on utilise pour construire un algorithme de type séparation et évaluation progressive pour le calcul de la fonction feuille.

2.1 NP-complétude

On étudie ici la complexité du problème de décision LIS (Leafed Induced Subtree) portant sur les sous-arbres induits et leur nombre de feuilles.

Définition 2.1. Le problème LIS est un problème de décision qui, pour un graphe G et deux entiers i et ℓ, consiste à décider s'il existe un sous-arbre induit de G qui a exactement i sommets et ℓ feuilles.

Rappelons que le problème IS consiste à décider si un graphe possède un sous-arbre induit à plus de i sommets. On sait qu'il est NP-complet par la proposition 1.17. Pour démonter que LIS est NP-complet, on procède donc à une réduction à partir de ce dernier.

Théorème 2.2. *Le problème* LIS *est NP-complet.*

Démonstration. On vérifie facilement que le problème est dans NP. En effet, si l'on prend comme certificat un sous-ensemble de sommets de G, on peut vérifier en temps polynomial si celui-ci induit un sous-arbre à i sommets et ℓ feuilles.

Soit $G = (V, E)$ un graphe simple. On construit H à partir de G en ajoutant pour chaque sommet v dans V un sommet v' et l'arête $\{v, v'\}$. Explicitement,

$$H = (V \cup V', E \cup \{\{v, v'\} : v \in V\}),$$

où $V' = \{v' : v \in V\}$. Un exemple de cette transformation est illustré à la figure 2.1. Cette transformation s'effectue en temps et en espace polynomial puisqu'on ne fait qu'ajouter $|V|$ sommets et $|V|$ arêtes.

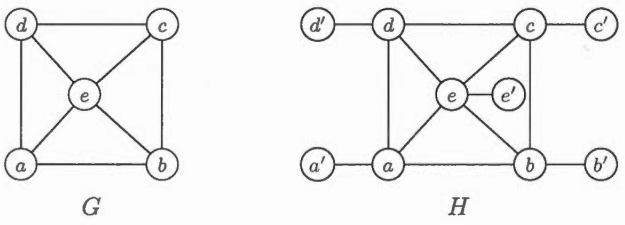


Figure 2.1 Transformation polynomiale d'une instance de IS en instance de LIS.

Soit $f(G, i) = (H, 2(i+1), i+1)$ notre transformation. On montre que G possède un sous-arbre induit à plus de i sommets si et seulement si H contient un sous-arbre induit à $2(i+1)$ sommets et $i+1$ feuilles.

$\boxed{\Longrightarrow}$ Si G possède un sous-arbre induit de plus de i sommets, G a, entre autres, un sous-arbre induit de $i+1$ sommets. Soit $\{v_1, \ldots, v_{i+1}\} \subseteq V$, induisant un sous-arbre dans G. Alors, l'ensemble de sommets $\{v_1, v_1', \ldots, v_{i+1}, v_{i+1}'\}$ induit un sous-arbre dans H. En effet, les sommets $v_1, \ldots, v_{i+1}$ induisent un sous-arbre dans H, car ils en induisent un dans G. L'ajout des sommets $v_1', \ldots, v_{i+1}'$ n'induit pas de cycle. En effet, aucun cycle de H ne contient un sommet de V' puisqu'ils sont tous de degré 1. Donc, les sommets $v_1, v_1', \ldots, v_{i+1}, v_{i+1}'$ induisent un sous-arbre

qui, en outre, a exactement $i+1$ feuilles. Effectivement, chaque sommet de V' doit être de degré 1 dans $H[\{v_1, v_1', \ldots, v_{i+1}, v_{i+1}'\}]$, car il est de degré 1 dans H. À l'opposé, chaque sommet $v_j \in V$ doit être de degré plus grand ou égal à 2, car v_j est connecté à sa copie v_j', ainsi qu'à un autre sommet de V, puisque les $v_1, \ldots, v_{i+1}$ induisent un sous-arbre de H (et forment donc un sous-graphe induit connexe). Par conséquent, chaque sommet de V' doit être une feuille alors que chaque sommet de V ne peut en être une. Conséquemment, $H[\{v_1, v_1', \ldots, v_{i+1}, v_{i+1}'\}]$ est un sous-arbre induit de $2(i+1)$ sommets et $i+1$ feuilles.

$\boxed{\Longleftarrow}$ Supposons que H possède un sous-arbre induit à $2(i+1)$ sommets et $i+1$ feuilles. Soit $U \subseteq V \cup V'$ le sous-ensemble induisant ce dernier. On s'assure que $|U \cap V| \geq i+1$. Si $v' \in V'$ est dans U, alors il est nécessaire que le sommet v de H soit dans U pour que le sous-graphe induit soit connexe. En effet, c'est le seul sommet adjacent à v' dans H. Comme chaque sommet de V' partage l'unique arête qui lui est incidente avec un sommet distinct de V, l'ensemble U contient un sommet différent de V pour chaque sommet distinct de V' qu'il contient. Ainsi, $|U \cap V| \geq |U \cap V'|$. Or, comme $|U \cap V| + |U \cap V'| = 2(i+1)$, on obtient que $|U \cap V| \geq (i+1)$. Comme $H[U]$ est acyclique, on a que $H[U \cap V]$ l'est aussi. Ce dernier graphe est aussi connexe, car les sommets de $V' \cap U$ sont des feuilles de $H[U]$. Pour cette raison, puisque $G = H[V]$, le graphe $G[U \cap V]$ est identique à $H[U \cap V]$ et, conséquemment, il est un sous-arbre induit de G à plus de i sommets.

En conclusion, on a donc que IS $\preceq$ LIS et donc LIS est NP-complet par la proposition 1.17. □

Comme le problème LIS est NP-complet, il est difficile d'espérer mieux pour le calcul de la fonction feuille qu'un algorithme exponentiel. Dans la suite de ce chapitre, on développe donc un algorithme de type séparation et évaluation pour le calcul de la fonction feuille.

2.2 Configuration d'un graphe

On décrit maintenant une structure de donnée appelée *configuration* de sous-arbre induit d'un graphe G qui sert à parcourir tous les sous-arbres induits de G. Une configuration est un sous-arbre induit enrichi d'informations qui permettent de construire d'autres sous-arbres induits par inclusion, exclusion et retour sur trace. On attribue des couleurs aux sommets de G, inspirées des feux de circulation. Les sommets coloriés en vert décrivent forcément un sous-arbre induit par (a). On appel ce dernier *sous-arbre vert* ou simplement *arbre vert.*

Définition 2.3. Soit $G = (V, E)$ un graphe simple et $\Gamma = \{vert, jaune, rouge, bleu\}$ un ensemble de couleurs. Une *configuration* C sur G est le couple donné par un coloriage $c : V \to \Gamma$ et une pile H de coloriage appelée *historique* de C.

Tous les coloriages $c : V \to \Gamma$ doivent satisfaire les conditions suivantes :

(a) Le sous-graphe induit par $c^{-1}(vert)$ est un arbre ;

(b) Si $c(u) = vert$ et $\{u, v\}$ est dans E, alors $c(v) \in \{vert, jaune, rouge\}$;

(c) Si $c(u) = jaune$, alors $|c^{-1}(vert) \cap N(u)| = 1$ où $N(u)$ est l'ensemble des voisins de u.

La *configuration initiale* sur G est la paire (c_{bleu}, H_0) où $c_{bleu}(v) = bleu$ pour tout v dans G et H_0 est la pile vide.

Finalement, on dit que deux configurations sur le même graphe sont égales, si elles ont la même coloration et le même historique (les mêmes colorations empilées dans le même ordre dans H).

Dans une configuration, le rouge indique les sommets qui ne peuvent pas être ajoutés au sous-arbre vert. Un sommet peut être rouge pour deux raisons. Cela peut être parce qu'il est adjacent à deux sommets verts et que conséquemment,

son inclusion créerait un cycle. Sinon, c'est qu'il est explicitement exclu pour des raisons de génération par l'utilisation de l'opération EXCLURESOMMET. Cette seconde possibilité est discutée plus loin dans cette section. Les sommets jaunes sont adjacents à un seul sommet de l'arbre vert. Pour cette raison, ils représentent les sommets que l'on pourrait colorier en vert, pour augmenter le nombre de sommets de l'arbre vert. Finalement, les sommets bleus sont ceux qui ne sont pas « visibles », c'est-à-dire qu'ils sont à une distance d'au moins deux de tous les sommets verts.

Les configurations sont des structures que l'on peut transformer en utilisant plusieurs opérations. On verra que l'on peut modifier la coloration de différents sommets en respectant certaines conditions. Pour des raisons de génération et pour permettre un retour sur trace efficace, il est utile de connaître toutes les colorations qui ont mené à la configuration actuelle. Aussi, ces colorations sont enregistrées dans la pile H.

La figure 2.2 (a) illustre un exemple de configuration. Les sommets $0, 1, 2, 3, 4, 5$ et 6 induisent un sous-arbre dont les arêtes sont mises en évidence en vert. Les sommets 14 et 15 doivent être rouges puisqu'ils sont adjacents à plus d'un sommet vert. Les sommets $7, 8, 9, 10$ et 11 pourraient être jaunes ou rouges puisqu'ils sont voisins d'exactement un sommet du sous-arbre vert. Finalement, les sommets $12, 13$ et 16 pourraient être bleus ou rouges, puisqu'ils ne partagent pas d'arête avec un sommet vert.

Comme mentionné précédemment, on peut obtenir de nouvelles configurations en appliquant différentes opérations à une configuration. On peut d'abord initialiser une configuration à partir d'un graphe. L'initialisation sur un graphe G consiste à créer la configuration initiale sur G, c'est-à-dire la configuration sur G avec la coloration à bleu partout et un historique vide. Pour les autres opérations

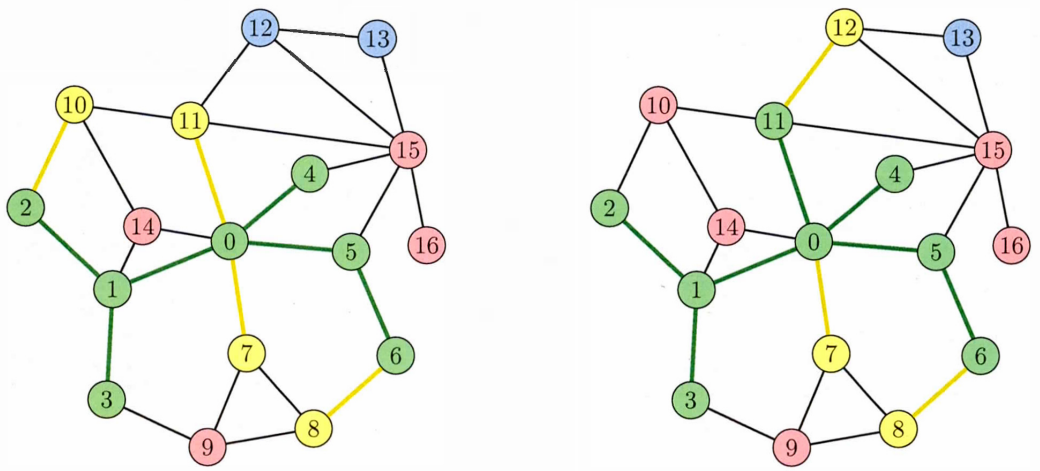


Figure 2.2 Configuration de sous-arbre induit. Les arêtes vertes mettent en évidence le sous-arbre vert et les arêtes jaunes, les extensions possibles de ce dernier. (a) Une configuration C. (b) La configuration C.INCLURESOMMET(11).

possibles, on utilise la définition suivante :

Définition 2.4. Soit $G = (V, E)$ un graphe et $C = (c, H)$ une configuration sur G. On dit qu'un sommet $v \in V$ est *coloriable* pour C si $c(v) =$ *jaune* ou si $c(v) =$ *bleu* et $c^{-1}(vert) = \emptyset$.

En général, un sommet coloriable est donc un sommet qu'il est possible de colorier en vert, tout en continuant de respecter la condition que les sommets verts induisent un sous-arbre de G. En effet, si le sous-arbre vert est non vide, un sommet coloriable est adjacent à exactement un sommet de l'arbre et s'il est vide, alors n'importe quel sommet peut être choisi. Dans les deux cas, on évite les sommets rouges puisqu'on les considère comme exclus. Pour une configuration $C = (c, H)$, on a les opérations suivantes :

— C.EXTENSIONPOSSIBLE() qui retourne n'importe quel sommet de G qui est

coloriable. S'il n'existe pas de tel sommet, l'opération retourne NUL. Cette opération est non déterministe.

— C.INCLURESOMMET(v) qui empile d'abord une copie de c sur le dessus de H. Il change ensuite la couleur de v à *vert* et met à jour les couleurs des voisins de v pour conserver une coloration valide. Le sommet v doit être coloriable.

— C.EXLCURESOMMET(v) qui empile d'abord une copie de c au sommet de H et change ensuite la couleur du sommet v à *rouge*. Il faut que v soit coloriable.

— C.DEFAIRE que l'on décrit à la section 2.2.1.

On peut illustrer le fonctionnement de ces opérations sur la configuration C de la figure 2.2 (a). C.EXTENSIONPOSSIBLE() retourne un des éléments de $\{7, 8, 10, 11\}$, car ce sont tous les sommets qui sont jaunes. Soit C', la configuration obtenue en appliquant INCLURESOMMET(11) à C. On doit alors faire une mise à jour des couleurs de la façon suivante : $c(11) \leftarrow$ *vert*, $c(10) \leftarrow$ *rouge* et $c(12) \leftarrow$ *jaune*. C' est illustrée à la figure 2.2 (b).

On dit qu'on applique une *extension* à la configuration si on utilise une opération d'inclusion ou d'exclusion. De façon analogue, on dira qu'une configuration C' peut être obtenue d'une configuration C par *extension*, si on peut l'obtenir en n'utilisant que les opérations INCLURESOMMET et EXCLURESOMMET.

2.2.1 Opération Défaire

Finalement, une configuration offre une dernière opération : C.DEFAIRE(). Les opérations DEFAIRE, INCLURESOMMET et EXCLURESOMMET forment l'ensemble des opérations qui modifient la configuration. On les nomme donc *modification*. Comme son nom l'indique, nous voulons que DEFAIRE annule les extensions effec-

tuées sur la structure. On veut, en particulier, qu'une inclusion ou une exclusion suivie de DEFAIRE ne modifie pas la coloration. On définit donc cette dernière à partir de cette propriété. On montre par la suite, qu'il existe une opération satisfaisant cette propriété et on démontre certaines des propriétés de cette opération.

Définition 2.5. L'opération DEFAIRE() est n'importe quelle opération qui satisfait les deux conditions suivantes :

1. Il ne peut être appliqué à une configuration C que si la pile H est non vide ;
2. Pour toute configuration C et tout sommet coloriable v de C, alors on a

$$C = (C.\text{INCLURESOMMET}(v).\text{DEFAIRE}()), \quad \text{et}$$
$$C = (C.\text{EXCLURESOMMET}(v).\text{DEFAIRE}()).$$

On montre à l'aide du lemme suivant qu'il existe une telle opération.

Lemme 2.6. *Si* DEFAIRE *dépile la tête de H et remplace c par celle-ci alors* DEFAIRE *satisfait la définition 2.5.*

Démonstration. On note d'abord que l'opération ne s'applique que si la pile est non vide, puisqu'on doit dépiler la tête de celle-ci. De plus, il est clair que l'opération satisfait la seconde condition. En effet, la coloration n'est pas modifiée puisque la coloration initiale est empilée, puis dépilée pour redevenir la coloration de la configuration. De plus, la pile n'est pas modifiée puisqu'on ne fait qu'y empiler un élément que l'on dépile ensuite. □

Comme l'illustre le lemme précédent, l'intuition derrière DEFAIRE est qu'il s'agit d'une opération qui permet d'annuler successivement les dernières modifications empilées dans l'historique.

Pour bien comprendre l'effet de DEFAIRE, on étudiera l'effet d'une suite de modifications sur une configuration. Soit $o_1, \ldots, o_n$ une suite de modifications, c'est-à-dire une suite d'opération INCLURESOMMET($\cdot$), EXCLURESOMMET($\cdot$) et DEFAIRE(). On dénote $C.(o_1 \ldots o_n)$ la configuration obtenue en appliquant successivement à C les opérations o_1, o_2 jusqu'à o_n. On note d'abord que

$$C.(o_1 \cdots o_n) = (C.(o_1 \cdots o_k)).(o_{k+1} \cdots o_n)$$

pour $1 \leq k \leq n$. On dénote par () la suite d'opération ne contenant aucune opération. Ainsi $C.() = C$.

2.2.2 Suite de modifications admissibles

Remarquons que toute séquence d'opérations ne peut pas toujours être appliquée sur une configuration donnée. Par exemple, la suite d'opérations consistant à inclure le sommet 11, puis exclure le sommet 10, ne peut être appliquée à la configuration de la figure 2.2 (a), car 10 n'est pas coloriable dans C.INCLURESOMMET(11) illustrée à la figure 2.2 (b). En général, pour inclure ou exclure un sommet v, il faut qu'il soit coloriable et pour pouvoir appliquer DEFAIRE, il faut que la pile H soit non vide.

Définition 2.7. Soit $C = (c, H)$ une configuration. Alors INCLURESOMMET(v) ou EXCLURESOMMET(v) est une *opération admissible* pour C, si v est coloriable dans C. De plus, DEFAIRE() est une *opération admissible* pour C, si H est non vide.

On dit qu'une suite de modifications $o_1, o_2, \ldots, o_n$ est *admissible* pour une configuration C, si pour $1 \leq i \leq n$, o_i est une opération admissible pour $C.(o_1 \ldots o_{i-1})$.

Inspiré par la terminologie de la combinatoire des mots, on dit qu'un *facteur* d'une suite d'opérations $o_1, o_2, \ldots, o_n$ est une suite de la forme $o_i, \ldots, o_j$ avec

$1 \leq i \leq j \leq n$. On dit aussi qu'un *préfixe* est un facteur contenant o_1 et un *suffixe* est un facteur contenant o_n.

Soit $o_1, o_2, \ldots, o_n$ une suite de modifications effectuées sur une configuration de départ C. On veut mettre en parallèle les suites de modifications et les mots sur l'alphabet $\{[,]\}$. Pour ce faire, on utilise la fonction p qui associe une lettre à chaque modification. On définit p comme suit :

$$p(o) = \begin{cases} [& \text{si } o \text{ est une inclusion ou une exclusion,} \\ ] & \text{si } o \text{ est l'opération DEFAIRE.} \end{cases}$$

La fonction p nous permet de transformer une suite d'opérations en un mot $w = p(o_1)p(o_2)\ldots p(o_n)$.

Définition 2.8. Soit $o_1, \ldots, o_n$ une suite admissible pour une configuration C. On dit que $o_1, \ldots, o_n$ est *bien parenthésée* si $p(o_1)p(o_2)\ldots p(o_n)$ est un mot bien parenthésé (voir la définition 1.9).

Lemme 2.9. *Soit C une configuration et $o_1, \ldots, o_n$ une suite de modifications admissibles pour C. Si $o_1, \ldots, o_n$ est bien parenthésée alors $C.(o_1 \ldots o_n) = C$.*

Démonstration. On procède par récurrence sur la longueur de la suite d'opérations. Comme les mots bien parenthésés sont toujours de longueur paire, on procède par bonds de deux.

Si $n = 0$, le résultat est trivial. Soit $o_1, \ldots, o_n$ une suite de modifications admissibles et bien parenthésées. On sait alors que $w = p(o_1)\ldots p(o_n)$ contient un facteur $[]$. Ainsi, il existe un indice k, tel que o_k est INCLURESOMMET ou EXCLURESOMMET et o_{k+1} est DEFAIRE. Par définition de DEFAIRE, on a donc que

$$C.(o_1 \ldots o_{k-1}) = (C.(o_1 \ldots o_{k-1})).(o_k o_{k+1}) = C.(o_1 \ldots .o_{k+1}).$$

On en déduit donc, pour $k+2 \leq i \leq n$, que

$$\begin{aligned} C.(o_1 \dots o_i) &= (C.(o_1 \dots o_{k+1})).(o_{k+2} \dots o_i) \\ &= (C.(o_1 \dots o_{k-1})).(o_{k+2} \dots o_i) \\ &= C.(o_1 \dots o_{k-1} o_{k+2} \dots o_i). \end{aligned}$$

En particulier, on a que

$$C.(o_1 \dots o_n) = C.(o_1 \dots o_{k-1} o_{k+2} \dots o_n).$$

De plus, $o_1, \dots, o_{k-1}, o_{k+2}, \dots, o_n$ est admissible pour C. En effet, pour i dans $\{1, 2 \dots, k-1\}$ il est clair que o_i est admissible pour $C.(o_1 \dots o_{i-1})$ par l'admissibilité de $o_1, \dots, o_n$. Pour $i = k+2, \dots, n$, on a que o_i est admissible pour $C.(o_1 \dots o_{k-1} o_{k+2} \dots o_{i-1})$, car il l'est pour $C.(o_1 \dots o_{i-1})$ et

$$C.(o_1 \dots o_{i-1}) = C.(o_1 \dots o_{k-1} o_{k+2} \dots o_i)$$

par le raisonnement précédent. Comme $o_1 \dots o_{k-1} o_{k+2} \dots o_n$ est une suite admissible bien parenthésée de longueur $n-2$, on a par récurrence que $C = C.(o_1 \dots o_n)$.

□

Corollaire 2.10. *Soit C une configuration et $o_1, \dots, o_n$ une suite de modifications admissibles. Si $o_i, \dots, o_j$ est une suite bien parenthésée, alors $o_1, \dots, o_{i-1}, o_{j+1}, \dots, o_n$ est une suite admissible pour C et*

$$C.(o_1, \dots, o_n) = C.(o_1, \dots, o_{i-1}, o_{j+1}, \dots, o_n)$$

Démonstration. Par le lemme 2.9, on sait que

$$C.(o_1 \dots o_j) = C.(o_1 \dots o_{i-1})$$

Par un argument similaire à celui de la preuve précédente, on peut montrer que $o_1, \dots, o_{i-1}, o_{j+1}, \dots, o_n$ est une suite admissible pour C. Finalement, on a que

$$C.(o_1 \dots o_n) = (C.(o_1 \dots o_j)).(o_{j+1} \dots o_n)$$

$$= (C.(o_1 \dots o_{i-1})).(o_{j+1} \dots o_n)$$
$$= C.(o_1 \dots o_{i-1} o_{j+1} \dots o_n)$$

□

On conclut cette section en montrant que toute configuration obtenue en appliquant les trois opérations de modification à la configuration initiale peut être obtenue en n'utilisant que des extensions.

Lemme 2.11. *Soit G un graphe et C_0 la configuration initiale sur ce dernier. Soit $o_1, \dots, o_n$ une suite admissible pour C_0 et $C = C_0.(o_1 \dots o_n)$. Alors, il existe $1 \leq i_1 \leq \dots \leq i_k \leq n$ tel que $C = C_0.(o_{i_1} \dots o_{i_k})$ et les o_{i_j} soient des extensions.*

Démonstration. On remarque d'abord qu'au moins une opération DEFAIRE est précédée par une extension. En effet, puisque DEFAIRE() ne peut pas être appliqué lorsque H est vide, o_1 doit être une extension. Ainsi, la première opération DEFAIRE de la suite est forcément précédée d'une extension.

Si on a une suite d'opérations contenant k fois DEFAIRE, alors par le Corollaire 2.10 C peut être obtenue par une suite de modifications contenant seulement $k-1$ DEFAIRE. En effet, on sait qu'il existe une opération DEFAIRE qui est précédée d'une extension. Ces deux opérations forment un facteur bien parenthésé. Par le Corollaire 2.10, on peut les retirer de la suite sans effet. On peut conséquemment obtenir C par une suite admissible ne contenant que $k-1$ opérations DEFAIRE. On procède ainsi par suppressions successives d'opérations DEFAIRE jusqu'à ce qu'il ne reste plus que des extensions dans la suite.

□

2.3 Implémentation

Pour utiliser la structure, nous avons besoin d'une implémentation efficace des différentes opérations. On utilise une structure de données CONFIGURATION pour implémenter la structure de configuration décrite ci-dessus. On montre aussi que l'implémentation de tous ses opérations a une complexité linéaire ou moindre. La structure de données CONFIGURATION possède les attributs suivants :

- un graphe G implémenté avec des listes d'adjacence (c'est un attribut constant qui ne change jamais),
- un tableau associatif c ayant comme clés les sommets de G,
- un entier ℓ,
- une pile de sommets His,
- un tableau associatif i ayant comme clés les sommets de G.

La coloration et l'historique d'une configuration sont simulés par les attributs de la structure de données CONFIGURATION de la manière suivante : on retrouve la coloration de la configuration dans le tableau associatif c. On verra, dans l'implémentation des opérations que l'historique H est simulé à partir des deux attributs His et i, en utilisant seulement un espace linéaire au lieu de quadratique. En particulier, la hauteur de la pile His indique la hauteur de la pile H puisque chacune des opérations d'extension y empile un élément. L'attribut i permet aussi d'ajuster ℓ pour qu'il indique en tout temps le nombre de feuilles du sous-arbre vert. Notons que pour cette implémentation, il est plus simple de considérer un sommet de degré 0 comme une feuille. L'ajustement pouvant facilement être fait à posteriori, on considére dans cette section qu'une *feuille* est un sommet de degré 0 ou 1 dans un arbre.

L'implémentation proposée offre toutes les opérations décrites dans la section pré-

cédente, c'est-à-dire une initialisation, ainsi que les opérations ExtensionPossible, InclureSommet, ExclureSommet et Defaire.

Soit G un graphe et C_0 l'instance de Configuration représentant la configuration initiale sur G. Le tableau c de C_0 a toutes ses valeurs à *bleu*. Pour C_0, le tableau i est initialisé à NUL, la pile His est vide et $\ell = 0$. On veut montrer que pour toute suite admissible de modifications $o_1, \ldots, o_n$, la configuration $C_0.(o_1 \ldots o_n)$ respecte les invariants suivants :

(i) c décrit une coloration valide (voir la définition 2.3),

(ii) ℓ est le nombre de feuilles du sous-arbre induit par $c^{-1}(vert)$,

(iii) $i[v]$ est le degré de v dans l'arbre vert si $c(v) = vert$, $i[v] =$ NUL si $c(v) \in \{jaune, bleu\}$ et si $c(v) = rouge$ alors $i[v] = u$ implique $c(u) = vert$.

On s'intéresse maintenant à l'implémentation des différentes opérations offertes par cette structure. On montre, par la suite, qu'elles préservent les invariants et on analyse leur complexité. On commence par l'initialisation :

Lemme 2.12. *Soit G un graphe à n sommets. L'initialisation, c'est-à-dire la création de C_0 sur G s'effectue en $\mathcal{O}(n)$ et celle-ci respecte les conditions (i) à (iii).*

Démonstration. On initialise $c[v]$ à *bleu* pour tous sommets v de G. Ainsi, c représente une coloration valide et (i) est satisfait. De même, pour tout $v \in G$, on initialise $i[v]$ à NUL. La condition (iii) est donc vérifiée, car tous les sommets sont bleus. On fixe $\ell = 0$ et comme le sous-arbre vert est vide, la condition (ii) est satisfaite. Finalement, His est la pile vide. Pour chaque attribut, l'initialisation se fait en $\mathcal{O}(1)$ ou $\mathcal{O}(n)$ et un simple pointeur vers le graphe donné est nécessaire. La complexité temporelle totale de l'initialisation est donc dans $\mathcal{O}(n)$. □

On s'intéresse maintenant aux opérations d'exclusion et d'inclusion. Tout d'abord, l'exclusion s'effectue en temps constant. Pour ce faire, on utilise l'algorithme 2.13

Algorithme 2.13 Exclusion d'un sommet de C

Précondition: v est coloriable

```
1: procédure ExclureSommet(v : sommet)
2:     c[v] ← rouge
3:     His.Empiler(v)
4: fin procédure
```

Lemme 2.14. *Soit C une* Configuration *respectant (i) à (iii) et v un sommet coloriable. Alors C.ExclureSommet(v) tel que décrit par l'algorithme 2.13 satisfait les conditions (i) à (iii) et le temps d'exécution est dans $\mathcal{O}(1)$.*

Démonstration. Soit c la coloration de C et c' la coloration de la configuration C.ExclureSommet(v). Par hypothèse, on sait que c est valide. On s'assure d'abord que c' vérifie les trois conditions d'une coloration (voir la définition 2.3). Comme les sommets verts ne changent pas, on sait que (a) est satisfait par c'. Pour le (b), si c ne colorie pas de sommet en vert, alors c' n'en colorie pas non plus. Aussi, si c colorie des sommets en vert, alors, comme v est coloriable, il passe de jaune à rouge. La condition (b) reste donc vérifiée dans les deux cas. Finalement, la condition (c) reste vraie, car aucun nouveau sommet n'est jaune et les sommets verts restent les mêmes. Ainsi, la coloration de C.ExclureSommet(v) est valide. Conséquemment, cette configuration satisfait (i).

Comme c et c' décrivent le même sous-arbre vert, (ii) est satisfait puisque ℓ n'est pas modifié. Finalement, i et le sous-arbre vert ne sont pas modifiés et aucun nouveau sommet n'est colorié en bleu ou jaune. Ainsi, (iii) reste vérifié pour les sommets bleus, jaunes et verts. Comme $i[v] = \text{NUL}$ et que v est le seul nouveau sommet rouge, (iii) est, en fait, vérifié pour toutes les couleurs de sommets.

On voit directement que le temps d'exécution est dans $\mathcal{O}(1)$. □

L'ajout d'un sommet est légèrement plus complexe puisqu'il faut ajuster la couleur

de plusieurs sommets pour conserver une coloration valide. Pour ce faire, on utilise l'algorithme 2.15. On prouve un résultat analogue à celui du lemme 2.14.

Algorithme 2.15 Ajout d'un sommet au sous-arbre vert

Précondition: v est coloriable

```
1: procédure INCLURESOMMET(v : sommet)
2:     c[v] ← vert
3:     ℓ ← ℓ + 1
4:     i[v] ← 1
5:     His.EMPILER(v)
6:     pour u ∈ G.VOISINS(v) faire
7:         si c[u] = vert alors
8:             si i[u] = 1 alors
9:                 ℓ ← ℓ − 1
10:            fin si
11:            i[u] ← i[u] + 1
12:        sinon si c[u] = jaune alors
13:            c[u] ← rouge
14:            i[u] ← v
15:        sinon si c[u] = bleu alors
16:            c[u] ← jaune
17:        fin si
18:    fin pour
19: fin procédure
```

Lemme 2.16. *Soit C une* CONFIGURATION *sur un graphe G respectant (i) à (iii) et v un sommet coloriable. Alors C.*INCLURESOMMET(v) *tel que décrit par l'algorithme 2.15 satisfait les conditions (i) à (iii) et le temps d'exécution dans* $\mathcal{O}(\deg(v))$.

Démonstration. Soit c la coloration de C et c' la coloration de la configuration C.INCLURESOMMET(v). Par hypothèse, on sait que c est valide. On vérifie d'abord que c' satisfait les trois conditions d'une coloration (voir la définition 2.3). Si $c^{-1}(vert) = \emptyset$, alors $c'^{-1}(vert) = \{v\}$ induit forcément un sous-arbre. Sinon, comme v est coloriable et c est valide, $c(v) = jaune$. Ainsi, $c'^{-1}(vert) = c^{-1}(vert) \cup \{v\}$

avec v adjacent à un unique sommet de $c^{-1}(\mathit{vert})$ car v est coloriable. On obtient effectivement que c' satisfait (a).

Les sommets verts de c sont aussi des sommets verts de c' et aucun sommet n'est colorié en bleu lors du passage de c à c'. Tous les voisins des sommets verts de c sont donc verts, jaunes ou rouges dans c'. Le seul nouveau sommet vert dans c' est v et l'algorithme 2.15 colorie tous ses voisins bleus en jaune. Ainsi, la condition (b) est préservée.

Pour le (c), les sommets jaunes qui ne sont pas adjacents à v restent jaunes et adjacents à un unique sommet vert. De plus, les sommets jaunes de c qui sont adjacents à v sont, pour c', adjacents à deux sommets verts et coloriés en rouge. Finalement, les nouveaux sommets jaunes sont les sommets qui ont comme seul voisin vert v puisqu'ils étaient bleus dans c. Par conséquent, c' vérifie (c) et l'invariant (i) est préservé.

Pour le nombre de feuilles, on distingue trois cas, dépendemment de si le sommet v est adjacent à une feuille de l'arbre vert (figure 2.3 (a)), un sommet interne de l'arbre vert (figure 2.3 (b)), ou aucun sommet vert (si l'arbre vert est vide). Dans le premier cas, le nombre de feuilles ne change pas, alors que dans les deux autres cas, le nombre de feuilles augmente de 1. L'invariant (ii) est donc préservé puisque ℓ est toujours incrémenté de 1, mais qu'il est décrémenté (Ligne 9) si v est adjacent à un sommet vert qui était une feuille.

Dans i, les sommets verts dont le degré dans l'arbre vert est modifié sont mis à jour. En effet, les seuls sommets dont le degré est modifié sont v et le voisin de v dans l'arbre. On voit aussi qu'aucun sommet rouge ou vert ne change de couleur. Ainsi, le tableau i indique toujours NUL pour les sommets bleus et jaunes. Finalement, les nouveaux sommets rouges prennent v, qui est maintenant vert, comme valeur dans i. Donc, l'invariant (iii) est préservé. □

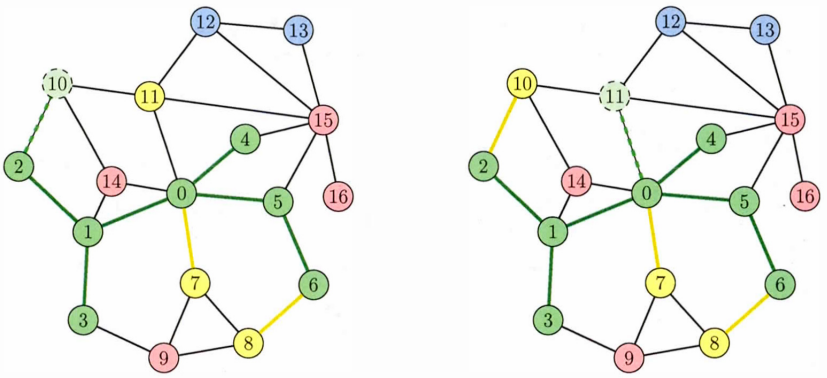


Figure 2.3 Inclusion de sommets et modification du nombre de feuilles. (a) On n'ajoute pas de feuille en incluant le sommet 10. (b) On ajoute une feuille en incluant le sommet 11.

Pour l'opération DEFAIRE, on utilise la procédure décrite par l'algorithme 2.17. On déduit du lemme suivant que l'implémentation de DEFAIRE satisfait la définition 2.5.

Lemme 2.18. *Soit C une* CONFIGURATION *respectant (i) à (iii) et v un sommet coloriable. Alors, avec l'implémentation de* DEFAIRE *présentée à l'algorithme 2.17, on a*

$$C = (C.(\text{INCLURESOMMET}(v))).\text{DEFAIRE}(), \text{ et}$$
$$C = (C.(\text{EXCLURESOMMET}(v))).\text{DEFAIRE}().$$

Démonstration. Pour vérifier l'égalité des deux configurations, on doit montrer que les attributs ont tous la même valeur avant et après l'application de la composition des deux opérations.

Les deux opérations d'extension modifient His en y empilant un sommet alors que la procédure DEFAIRE dépile de His. Ainsi, dans les deux cas, His n'est pas

Algorithme 2.17 Annulation de la dernière modification

Précondition: His est non vide

```
1: procédure DEFAIRE
2:     v ← His.DEPILER()
3:     si c[v] = vert alors
4:         ℓ ← ℓ − 1
5:         pour u ∈ G.VOISINS(v) faire
6:             si c[u] = vert alors
7:                 i[u] ← i[u] − 1
8:                 si i[u] = 1 alors
9:                     ℓ ← ℓ + 1
10:                fin si
11:            sinon si c[u] = jaune alors
12:                c[u] ← bleu
13:            sinon si c[u] = rouge et i[u] = v alors
14:                c[u] ← jaune
15:                i[u] ← NUL
16:            fin si
17:        fin pour
18:    fin si
19:    si ℓ > 0 alors
20:        c[v] ← jaune
21:    sinon
22:        c[v] ← bleu
23:    fin si
24:    i[v] ← NUL
25: fin procédure
```

modifié.

Seule la procédure INCLURESOMMET peut modifier ℓ. On constate que DEFAIRE modifie la valeur de ℓ si et seulement si v est vert (c'est-à-dire si l'extension est une inclusion) et qu'il procède exactement à l'inverse de INCLURESOMMET(). Ainsi, ℓ n'est pas modifié.

On remarque que les valeurs des deux tableaux associatifs ne sont modifiées que pour v et ses voisins. Dans C, v est bleu si le sous-arbre est vide et jaune si non. Conséquemment, $i[v] = \text{NUL}$. Ainsi, comme la condition $\ell > 0$ indique si le sous-

arbre est vide, la couleur de v n'est pas modifiée par la composition d'opérations. De plus, $i[v]$ vaut toujours NUL après avoir appliqué les deux opérations. Ainsi, les valeurs de $i[v]$ et $c[v]$ ne sont modifiées par aucune des deux compositions.

On note que les valeurs de i et c pour les voisins de v sont modifiées si et seulement si la première opération est une inclusion. Soit u un voisin de v. On procède selon la couleur de u dans C. Si u est rouge alors l'inclusion ne modifie ni $i[u]$, ni $c[u]$. Comme $i[u]$ est différent de v, car v n'est pas vert dans C, DEFAIRE ne modifie pas non plus les valeurs de c et i pour u. Si u est bleu, l'inclusion fixe $i[u]$ et change $c[u]$ à jaune. Par la suite, DEFAIRE rechange $c[u]$ à bleu et ne modifie pas $i[u]$. Si u est jaune, $i[u] = \text{NUL}$ dans C par (iii). L'inclusion change alors $c[u]$ à rouge et $i[u]$ à v. Ainsi, DEFAIRE() remet $c[u]$ à jaune et $i[u]$ à NUL. Finalement, si u est vert, la couleur de u n'est pas modifiée et la valeur de $i[u]$ est incrémentée puis décrémentée. On conclut que pour tous les voisins de v, les valeurs $c[u]$ et $i[u]$ ne sont pas modifiées.

Comme tous les attributs sont préservés par la composition, on a bien l'égalité des configurations dans les deux cas. □

Notons qu'on empile sur His lors de chacune des extensions, comme on empile une coloration sur H dans la définition des extensions. Ainsi, la condition "la pile H est non vide" est équivalente à "His est non vide". Comme DEFAIRE ne peut être appliqué que si His est non-vide, on a bien que l'implémentation de DEFAIRE satisfait la définition 2.5. Notons aussi que DEFAIRE s'exécute en $\mathcal{O}(\deg(v))$ où v est le degré du sommet dépilé de His. En général, la complexité de cette opération est donc en $\mathcal{O}(\delta)$ si δ est le degré maximal dans G.

Proposition 2.19. *Les conditions (i) à (iii) sont des invariants de l'implémentation de la structure de données* CONFIGURATION.

Démonstration. Par le lemme 2.12, à son initialisation, la CONFIGURATION respecte les trois conditions. Par les lemmes 2.14 et 2.16, si on utilise seulement les opérations EXCLURESOMMET et INCLURESOMMET, les conditions restent vérifiées. Ainsi, une suite de modifications ne contenant que des extensions préserve toujours les invariants.

Il reste à vérifier que les invariants sont préservés, si on ajoute des opérations DEFAIRE. Par le lemme 2.11, toute configuration obtenue à partir de la configuration initiale peut être obtenue en appliquant uniquement des extensions à cette dernière. Ainsi, les conditions (i) à (iii) sont des invariants de l'implémentation. □

La dernière opérations est EXTENSIONPOSSIBLE. L'utiliser ne modifie pas les attributs. En effet, pour déterminer si $c^{-1}(\mathit{vert})$ est vide, il suffit de regarder si $\ell = 0$. Par la suite, il suffit de parcourir les sommets de G jusqu'à en trouver un qui soit bleu si, le sous-arbre vert est vide ou sinon, un sommet qui soit jaune. Si tous les sommets sont parcourus sans en trouver un qui soit de la bonne couleur, l'opération retourne NUL. La complexité temporelle de EXTENSIONPOSSIBLE est donc en $\mathcal{O}(n)$.

2.4 Générateur de sous-arbres induits

La première utilisation que l'on fait de la structure de configuration est la construction d'un générateur de sous-arbres induits d'un graphe G. Pour ce faire, on utilise l'algorithme 2.20.

Théorème 2.21. *Soit G un graphe avec $|G| = n$. La structure de données configuration peut être utilisée pour générer les sous-arbres induits de G avec une initialisation en $\mathcal{O}(n)$ et un délai amorti en $\mathcal{O}(n)$.*

Algorithme 2.20 Générateur des sous-arbres induits d'un graphe G

```
 1: Soit C la configuration initiale sur le graphe G
 2: EXPLORERCONFIGURATION()
 3: fonction EXPLORERCONFIGURATION()
 4:     extension ← C.EXTENSIONPOSSIBLE()
 5:     si extension = NUL alors
 6:         Générer le sous-arbre vert
 7:     sinon
 8:         C.INCLURESOMMET(extension)
 9:         EXPLORERCONFIGURATION()
10:         C.DEFAIRE()
11:         C.EXCLURESOMMET(extension)
12:         EXPLORERCONFIGURATION()
13:         C.DEFAIRE()
14:     fin si
15: fin fonction
```

Démonstration. On utilise l'algorithme 2.20 pour générer les sous-arbres en utilisant la stratégie récursive. Il faut voir l'exécution de ce générateur comme un arbre. Pour un état donné de la configuration, que l'on voit comme un nœud de notre arbre, cet état a deux enfants, celui qui inclut l'extension proposée et celui qui l'exclut (voir figure 2.4 pour un exemple). À partir de la configuration C, on explore d'abord l'état avec l'inclusion, puis l'on fait un retour sur trace et on explore l'état avec l'exclusion. Le lemme 2.9 assure que C est identique avant l'inclusion de la Ligne 8 et avant l'exclusion de la Ligne 11.

Soit $T = \{v_1, \ldots, v_k\}$ un sous-ensemble de sommets de G induisant un sous-arbre. Pour obtenir le sous-arbre vert induit par T dans C, la seule façon possible est d'ajouter le sommet proposé, s'il est dans T et de l'exclure sinon. Chaque sous-arbre est donc atteint au plus une fois.

Il reste à voir que tous les sommets de T sont éventuellement proposés si l'on suit la stratégie décrite ci-dessus. Si aucun sommet n'a encore été ajouté, n'importe quel sommet bleu peut être proposé par EXTENSIONPOSSIBLE(), alors un de ceux

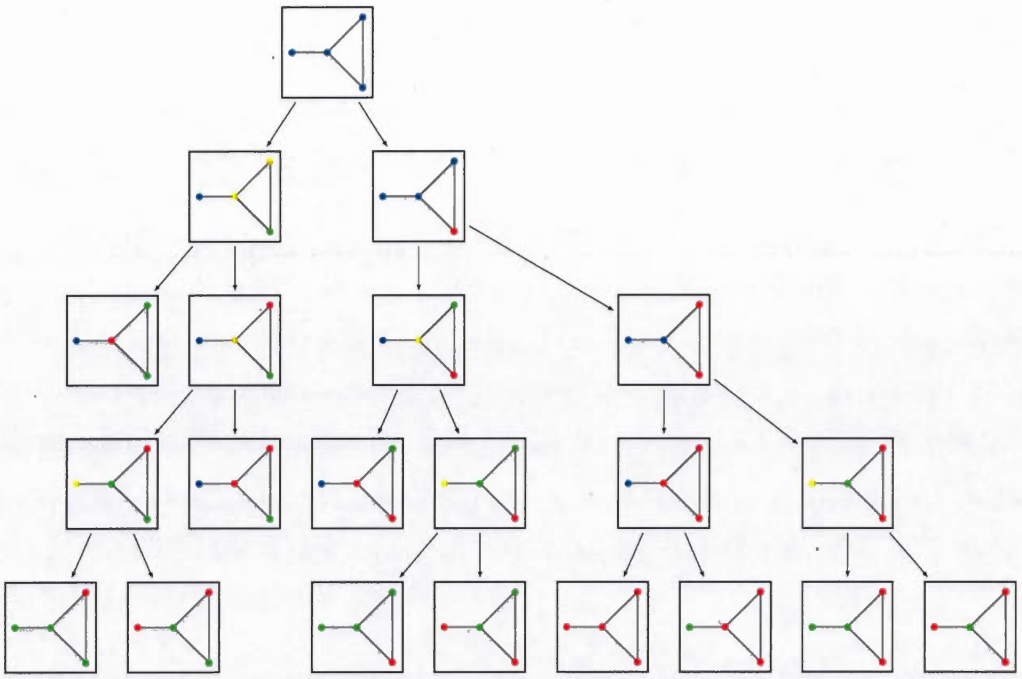


Figure 2.4 Arbre d'exploration du générateur de sous-arbres induit

de T est éventuellement proposé.

Si tous les sommets de T ne sont pas déjà verts dans C, on montre qu'un sommet non vert de T est éventuellement proposé. Comme T induit un sous-arbre dans G, au moins un sommet non vert de T doit être connecté à l'arbre vert. Il doit aussi être adjacent à un seul sommet vert, sinon le sous-graphe induit par T n'est pas acyclique. Comme ce sommet ne peut pas avoir été exclu, il doit être jaune. Par conséquent, il sera éventuellement proposé.

On conclut la preuve avec l'analyse de complexité. L'initialisation du générateur se fait en $\mathcal{O}(n)$ par le lemme 2.12 puisqu'on n'a qu'à initialiser une configuration pour G.

Pour le délai, on utilise la méthode d'analyse agrégée, décrite dans la section 1.4.1, en comptant le temps total d'exécution et en divisant par le nombre de sous-arbres

induits visités. Reprenons l'arbre d'exploration que nous avons décrit au début de la preuve. On nommera ce dernier S. Les nœuds de S représentent les configurations explorées, puis les arêtes, les transitions effectuées entre les configurations. Ainsi, l'ensemble des opérations INCLURESOMMET et EXLCURESOMMET effectuées par le générateur est en bijection avec les arêtes de S. De même, chaque opération DEFAIRE est associée à une inclusion ou une exclusion. Donc, l'ensemble des opérations DEFAIRE effectuées par le générateur est en bijection avec les arêtes de S. Finalement, une opération EXTENSIONPOSSIBLE est exécutée pour chaque nœud de S. Comme on a vu que la complexité des opérations était dans $\mathcal{O}(\delta)$, pour δ le degré maximal dans le graphe, le temps total d'exécution est dans

$$\begin{aligned} 2\mathcal{O}(\delta) \cdot \#\{\text{arête de } S\} + \mathcal{O}(n) \cdot \#\{\text{nœud de } S\} & \\ &= \mathcal{O}\left(\delta \cdot \#\{\text{arête de } S\} + n \cdot \#\{\text{nœud de } S\}\right) \\ &= \mathcal{O}\left(n \cdot (2 \cdot \#\{\text{arête de } S\} + 1)\right) \\ &= \mathcal{O}(n \cdot \#\{\text{arête de } S\}), \end{aligned}$$

car dans un arbre, il y a toujours exactement un nœud de plus que le nombre d'arêtes.

De plus, notons que l'on peut mettre en bijection les sous-arbres induits de G avec les feuilles de S, car le générateur retourne le nombre de feuilles d'un sous-arbre de G lorsqu'il n'y a plus d'extension possible, c'est-à-dire lorsqu'on a une feuille de S. Ainsi, le nombre sous-arbres visités est $\#\{\text{feuille de } S\}$.

On remarque que S est un arbre binaire entier, c'est-à-dire que chaque configuration a 2 ou 0 enfants, selon qu'une extension est possible ou non. Or, on peut vérifier qu'un arbre binaire entier satisfait la relation $\ell = \frac{a}{2} + 1$ avec a, son nombre d'arêtes, et ℓ, son nombre de feuilles. On a donc que la complexité amortie pour

le délai est

$$\mathcal{O}\left(\frac{n \cdot \#\{\text{arête de S}\}}{\#\{\text{feuille de S}\}}\right) = \mathcal{O}(n).$$

□

2.5 Potentiel de feuilles

Si l'on veut calculer la fonction feuille L_G en utilisant le générateur de la section précédente, il est évident que certaines configurations peuvent être ignorées, si elles ne peuvent pas être étendues en une configuration « très feuillue ». Dans cette section, on s'intéresse donc, pour une configuration donnée, à borner le nombre de feuilles qui peut être atteint par une extension de celle-ci. Ainsi, pour une configuration C avec n sommets verts, on définit la fonction C.PotentielDeFeuilles(n') qui calcule une borne sur le nombre de feuilles qui peut être obtenu par une extension de C en une configuration à n' sommets verts. Tout d'abord, on fixe C.PotentielDeFeuilles(n') $= -\infty$ pour n' plus grand que la taille de la composante connexe K qui contient le sous-arbre (pour le calcul de la composante connexe, on traite les sommets rouges comme exclus du graphe).

Ensuite, pour $n' < |K|$, on imagine un scénario optimiste, où tous les sommets bleus et jaunes accessibles qui sont assez proches peuvent être ajoutés sans créer de cycle et ce, peu importe l'ordre dans lequel ont les choisit. Pour utiliser cette idée, on partitionne d'abord les sommets jaunes et bleus accessibles, ainsi que les feuilles de l'arbre vert, par rapport à leur distance aux sommets internes de l'arbre vert. Lors du calcul de la distance, les sommets rouges sont traités comme s'ils avaient été retirés du graphe. La figure 2.5 montre un exemple de partition des sommets d'une configuration. Les sommets 1,2 et 3 sont les sommets internes de l'arbre vert. Les autres sommets sont étagés en fonction de leur distance à ces derniers. Le sommet 16 est bien à distance 3, car les sommets rouges sont ignorés.

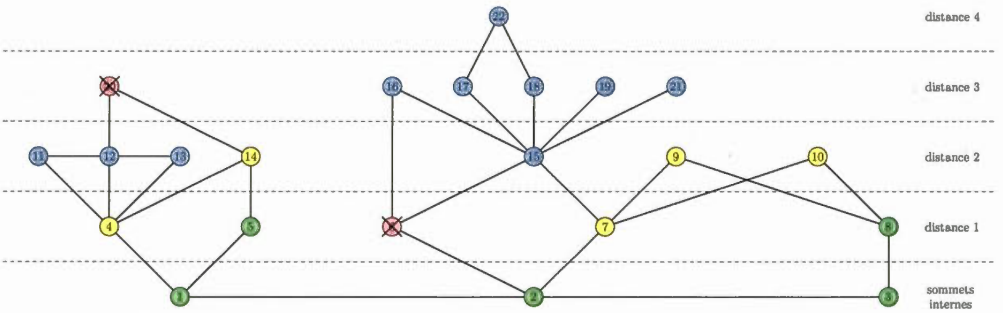


Figure 2.5 Partition des sommets d'une configuration pour le calcul du potentiel de feuilles

L'algorithme 2.22 calcule le potentiel de feuilles de C. La première partie de l'algorithme consiste à compléter la configuration. Précisément, on dit qu'une configuration est *complète* si chaque sommet jaune est adjacent à une feuille de l'arbre vert. Ainsi, aucun ajout de sommet ne peut créer une feuille. En regardant s'il y a des sommets jaunes à distance 1, on vérifie si la configuration est complète et, si elle ne l'est, pas on augmente n et ℓ comme si elle l'était (Lignes 5-9). Par la suite, on choisit un sommet v parmi tous les sommets *disponibles* (c'est-à-dire qui n'ont pas déjà été choisis) à distance d. On suppose qu'il est vert et on augmente alors n et ℓ comme si chaque voisin jaune ou bleu de v était une feuille ajoutée à la configuration courante (Lignes 14-18). On répète ce processus jusqu'à ce que n atteigne la taille objectif n'.

Dans l'exemple de la figure 2.5, pour compléter la configuration, il faudrait inclure les sommets 4 et 7 au sous-arbre vert. Ainsi, l'étape de complétion augmente n et ℓ de 2 et $(n, \ell) = (7, 4)$. Par la suite, les premiers sommets choisis par la fonction sont dans l'ordre 4 (degré 5), 15 (degré 6) puis 7 (degré 4). Les valeurs de (n, ℓ) deviennent alors successivement $(11, 7)$, $(16, 11)$ puis $(19, 13)$. Ainsi, pour $n' = 19$, le potentiel de feuilles serait 13 alors que pour $n' = 11$, il vaudra 7.

Algorithme 2.22 Calcul du potentiel de feuilles pour n'

```
1: fonction PotentielDeFeuilles(C : configuration, n' : entier) : entier
2:     n ← nombre de sommets verts de C
3:     ℓ ← nombre de feuilles du sous-arbre de C
4:     y ← nombre de sommets jaunes adjacents à un sommet interne de l'arbre
5:     si n + y ≥ n' alors
6:         (n, ℓ) ← (n', ℓ + (n' − n))
7:     sinon
8:         (n, ℓ) ← (n + y, ℓ + y)
9:     fin si
10:    d ← 1
11:    tant que n < n' et qu'il y a des sommets disponibles à distance au plus d
       faire
12:        Soit v un sommet disponible de plus haut degré
13:                                  ▷ Le degré ne compte pas les sommets rouges
14:        si n + deg(v) − 1 ≤ n' alors
15:            (n, ℓ) ← (n + deg(v) − 1, ℓ + deg(v) − 2)
16:        sinon
17:            (n, ℓ) ← (n', ℓ + (n' − n) − 1)
18:        fin si
19:        Retirer v des sommets disponibles
20:        d ← d + 1
21:    fin tant que
22:    retourner ℓ
23: fin fonction
```

On remarque d'abord que n et ℓ dans l'algorithme 2.22 satisfont l'invariant de boucle suivant :

Lemme 2.23. *Les variables n et ℓ de l'algorithme 2.22 satisfont l'invariant de boucle $n - \ell = |I| + k$ où I est l'ensemble des sommets internes du sous-arbre de C et k est le nombre de tours de boucle effectués.*

Démonstration. On initialise n et ℓ de telle sorte de $n - \ell = |I|$. Durant l'étape de complétion, on a

$$n' - (\ell + (n' - n)) = n - \ell = |I|, \text{ et}$$

$$n + y - (\ell + y) = n - \ell = |I|.$$

L'invariant est donc préservé durant cette phase. Finalement, si après k tours de boucles, $n - \ell = |I| + k$, alors après le tour suivant l'invariant est préservé car

$$n + \deg(v) - 1 - (\ell + \deg(v) - 2) = n - l + 1 = |I| + (k+1), \text{ et}$$
$$n' - (\ell + (n' - n) - 1) = n - \ell + 1 = |I| + (k+1).$$

□

Il découle directement du lemme 2.23 que si l'algorithme 2.22 s'arrête après k' itérations de la boucle, alors la valeur retournée est $n' - (|I| + k')$ puisque $n = n'$.

On prouve maintenant que l'algorithme 2.22 calcule une borne sur le nombre de feuilles qui peut être réalisé par extension. Mentionnons que pour obtenir une borne non triviale, on se restreint aux sommets disponibles à distance au plus d et on augmente la valeur de d à chaque itération.

Proposition 2.24. *Soit C une configuration d'un graphe simple G avec $n \geq 3$ sommets verts et soit n' un entier tel que $n \leq n' \leq |V|$. Alors, toute extension de C à une configuration à n' sommets a au plus C.PotentielDeFeuilles(n') feuilles, où le potentiel de feuilles est la fonction décrite par l'algorithme 2.22.*

Démonstration. On démontre le cas où $n' \leq |K|$. L'autre cas est direct.

Soit ℓ le nombre de feuilles du sous-arbre vert de C, I l'ensemble des sommets internes de ce sous-arbre et

$$Y = \{v \in G : v \text{ est un sommet jaune de } C \text{ à distance 1 de } I\}.$$

Soit $p' = C$.PotentielDeFeuilles(n'). Si $n' - n \leq |Y|$, alors $p' = \ell + n - n'$ et il est clair qu'ajouter $n - n'$ sommets ne peut pas ajouter plus que $n' - n$ feuilles.

Sinon, on procède par contradiction en supposant que l'on peut étendre C à une configuration C' avec n' sommets verts et ℓ' feuilles avec $\ell' > p'$. Soit $v_1, v_2, \ldots, v_k$ la suite de sommets qui deviennent, dans cet ordre, des sommets internes durant les extensions successives pour passer de C à C'. On a alors $\ell' = n' - (|I| - k)$. Soit $v'_1, v'_2, \ldots, v'_{k'}$ les sommets choisis par la procédure C.PotentielDeFeuilles(n'). Par la remarque suivant le lemme 2.23, on a que $p' = n' - (|I| + k')$. Comme on suppose que $\ell' > p'$, on en déduit que $k < k'$.

Sans perte de généralité, on suppose que si v_i et v_j sont à la même distance de I et $\deg(v_j) \leq \deg(v_i)$ alors $i \leq j$ (sinon on échange simplement les paires qui ne satisfont pas cette condition). De plus, on sait que v_i est au plus à distance i de I. Par conséquent,

$$\deg(v_1) \leq \deg(v'_1),\ \deg(v_2) \leq \deg(v'_2),\ \ldots, \deg(v_k) \leq \deg(v'_k).$$

Pour chaque nouveau sommet interne v_i, seuls ses voisins peuvent être ajoutés sans créer un nouveau sommet interne. De façon analogue, transformer v_i en sommet interne implique qu'au plus $\deg(v_i) - 2$ feuilles sont gagnées. En considérant les feuilles potentielles trouvées dans Y, on conclut que

$$\ell' \leq \ell + |Y| + \sum_{i=1}^{k} (\deg(v_i) - 2) \leq \ell + |Y| + \sum_{i=1}^{k'-1} (\deg(v'_i) - 2) \leq p'$$

ce qui est une contradiction. Ainsi la configuration C' ne peut pas exister. □

Notons que l'on peut calculer la fonction PotentielDeFeuilles pour tous les n' en une seule passe, en retenant les valeurs pour chaque n, au fur et à mesure que l'on incrémente sa valeur. De plus, pour une implémentation efficace, on utilise une file à priorité pour obtenir le sommet de plus grand degré disponible rapidement.

2.6 Calcul de la fonction feuille

On conclut ce chapitre en présentant l'algorithme 2.25, qui calcule la fonction feuille L pour un graphe arbitraire. L'idée inspirant cet algorithme consiste à générer tous les sous-arbres induits à l'aide des configurations comme à la section 2.4 en gardant en mémoire, dans une fonction L, les maximums de feuilles atteints pour chaque taille. Contrairement au générateur, on n'explore pas tous les nœuds. En fait, on cesse d'explorer une branche si l'on sait qu'aucune extension ne peut avoir plus de feuilles que ce qui est déjà enregistré dans la fonction L. Il découle de la proposition 2.24 qu'une configuration C à n sommets verts et r sommets rouges ne peut être étendue à une configuration dans laquelle le sous-arbre a plus de feuilles que les meilleures valeurs trouvées jusqu'à maintenant et enregistrées dans L lorsque

$$C.\textsc{PotentielDeFeuilles}(n') \leq L(n') \text{ pour tous } n \leq n' \leq |V|. \tag{2.1}$$

À titre d'exemple, on considère la configuration illustrée à la figure 2.6. Supposons que la fonction L calculée jusqu'à ce qu'on arrive à cette configuration est représentée à la deuxième ligne du tableau 2.1. De l'autre côté, les valeurs du potentiel de feuilles calculées avec l'algorithme 2.22 pour la configuration sont représentées à la troisième ligne de ce dernier. Par la proposition 2.24, on voit alors clairement qu'il est inutile de regarder les extensions de cette configuration, car aucune de celle-ci ne peut avoir plus de feuilles que la fonction L courante. On arrête donc l'exploration de cette branche.

Pour calculer la fonction feuille, on utilise alors l'algorithme 2.25.

En utilisant le théorème 2.21, ainsi que la proposition 2.24 et discussion précédente, on peut en déduire le résultat suivant :

Tableau 2.1 Potentiel de feuilles et fonction L courante.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$L(n)$	0	0	2	2	2	3	4	5	5	6	7	7	8	9
Potentiel					2	3	3	4	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$

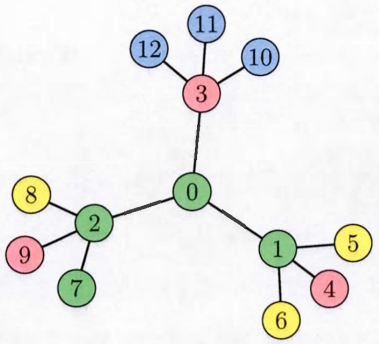


Figure 2.6 Une configuration C de l'arbre ternaire complet de hauteur 2.

Théorème 2.26. *Soit G un graphe simple. Alors, l'algorithme 2.25 retourne la fonction feuille L_G de G.*

2.7 Analyse de performance

Pour analyser la performance de l'algorithme, on utilise la *densité* d'un graphe. On définit la densité d'un graphe à n sommets et a arêtes par $\frac{2a}{n(n+1)}$. Il s'agit en fait du rapport du nombre d'arêtes sur le nombre maximal d'arêtes possible. Ainsi, un graphe de densité 1 est un graphe complet. On dit qu'un graphe est *dense*, si sa densité est élevée et qu'il est *clairsemé*, si sa densité est basse.

De façon empirique, on observe les éléments suivants : premièrement, il semble que la performance soit significativement meilleure sur les graphes denses. Plus précisément, pour un nombre fixe de sommets, le calcul de la fonction feuille est plus rapide sur un graphe dense que sur un graphe clairsemé. On peut le constater

Algorithme 2.25 Calcul de la fonction feuille

```
1: fonction FONCTIONFEUILLE(G : graphe) : liste
2:     fonction EXPLORERCONFIGURATION()
3:         u ← C.EXTENSIONPOSSIBLE()
4:         si u = NUL alors
5:             i ← nombre de sommets verts de C
6:             ℓ ← le nombre de feuilles de C
7:             L[i] ← max(L[i], ℓ)
8:         sinon si la condition (2.1) n'est pas satisfaite alors
9:             C.INCLURESOMMET(u)
10:            EXPLORERCONFIGURATION()
11:            C.DEFAIRE()
12:            C.EXCLURESOMMET(u)
13:            EXPLORERCONFIGURATION()
14:            C.DEFAIRE()
15:        fin si
16:    fin fonction
17:    Soit C la configuration initiale de G
18:    L[0] ← 0
19:    L[i] ← −∞ pour i = 1, 2, ..., |G|
20:    EXPLORERCONFIGURATION()
21:    retourner L
22: fin fonction
```

à la figure 2.7(a-b). Ceci n'est pas surprenant puisque, pour un sous-ensemble de sommets d'un graphe dense, la probabilité qu'il induise un cycle est plus élevée que dans un graphe clairsemé. On a donc moins de sous-arbres à visiter pour les graphes denses. Par exemple, des données expérimentales montrent que le nombre de sous-arbres visités, dans un graphe à 30 sommets et de densité 0.1, reste dix fois plus élevé que dans un graphe de densité 0.9 à 80 sommets.

Deuxièmement, le potentiel de feuille réduit toujours le nombre de sous-arbres visités, peu importe la densité. Toutefois, la différence est plus marquée sur les graphes clairsemés (voir figure 2.7(c-d)). On peut probablement expliquer cela par le fait que lorsque la densité décroit, le nombre d'étages dans la partition utilisée pour le calcul du potentiel de feuille croit. Conséquemment, lorsque le potentiel

de feuille est calculé sur un graphe peu dense, la borne est plus précise.

Finalement, l'expérimentation tend amène à conjecturer que la complexité globale de l'algorithme est dans $\mathcal{O}(\alpha^n)$ avec $\alpha < 2$. Malheureusement, aucune démonstration d'une telle borne n'a été trouvée.

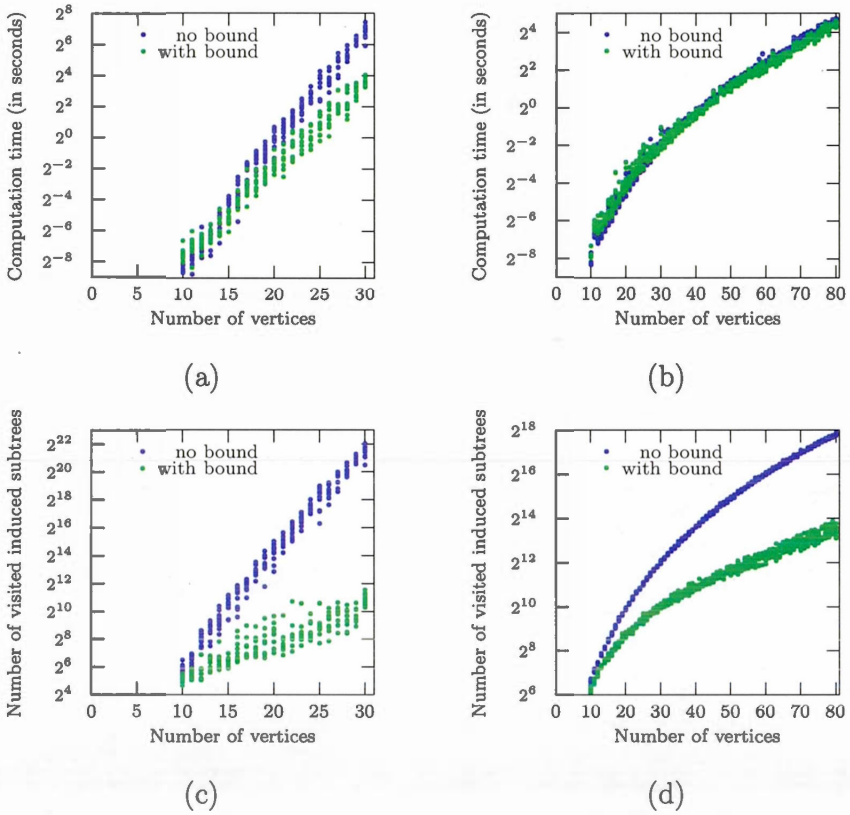


Figure 2.7 Le temps d'exécution de l'algorithme 2.25 sur 10 graphes générés aléatoirement de densité (a) 0.2 et (b) 0.8 avec ou sans utilisation du potentiel de feuille. Le nombre sous-arbres visités durant l'exécution est illustré en (c) pour la densité 0.2 et en (d) pour la densité 0.8. Algorithme implémenté à l'aide de SageMath et calcul effectués sur un Mac Pro équipé d'un processeur Intel(R) Xeon(R) CPU E5-2697 et de 64 Go de mémoire vive. Figure tirée de (Blondin Massé *et al.*, 2018c)

CHAPITRE III

RÉALISATION DE LA FONCTION FEUILLE ET GRAPHES CHENILLES

Dans le chapitre précédent, l'attention était mise sur le calcul de la fonction feuille. Dans celui-ci, on cherche à répondre à la question « inverse » du dernier problème. On tente de déterminer si, étant donnée une fonction f, il existe un graphe dont f est la fonction feuille. Pour ce faire, on introduit d'abord le mot feuille d'un graphe qui est la dérivée discrète de sa fonction feuille, on décrit l'alphabet des mots feuilles d'un graphe et d'un arbre puis, on introduit les mots préfixes normaux qui permettent de caractériser les fonctions feuilles pouvant être réalisées par les graphes chenilles. Tous les graphes considérés dans ce chapitre sont connexes

Définition 3.1. Soit ℓ une suite finie d'éléments de l'ensemble $\mathbb{N} \cup \{-\infty\}$. On dit que ℓ est une *suite feuillue*, s'il existe un graphe connexe G tel que la suite des valeurs de sa fonction feuille L_G soit égale à ℓ, c'est-à-dire que $\ell = (L_G(i))_{i=0,1,2,\ldots,|G|}$.

Par exemple, la suite $(0, 0, 2, 2, 3, 4, 4, -\infty)$ est une suite feuillue, car le graphe de la figure 3.1 la réalise. Au contraire, la suite $(0, 0, 2, 3, 3, 4, -\infty)$ n'est pas une suite feuillue. En effet, pour avoir un graphe G qui réalise la suite, il faudrait que $L_G(3)$ soit 3 or, il est impossible d'avoir un arbre avec 3 sommets et 3 feuilles.

On tente de résoudre le problème suivant :

Problème 3.2 (Réalisation de la fonction feuille)**.** Étant donnée une suite $\ell =$

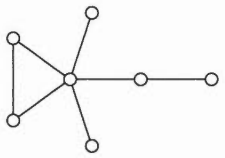


Figure 3.1 Graphe réalisant la suite $(0, 0, 2, 2, 3, 4, 4, -\infty)$.

$(\ell_0, \ell_1, \ldots, \ell_n)$ d'éléments de $\mathbb{N} \cup \{-\infty\}$, cette suite est-elle feuillue ? Autrement dit, existe-t-il un graphe connexe G à n sommets tel que L_G, la fonction feuille de G, prenne ℓ_i comme valeur en i ?

Un outil pratique pour l'étude du problème 3.2 est le mot de différence aussi appelé la dérivée discrète.

Définition 3.3. Soit L une fonction $L : \{0, 1, 2, \ldots n\} \to \mathbb{N} \cup \{-\infty\}$, on définit la *dérivée discrète* de ΔL comme le mot sur l'alphabet $\mathbb{Z} \cup \{\omega\}$ avec sa i^{e} lettre donnée par

$$\Delta L(i) = L(i+3) - L(i+2),$$

pour $i = 1, 2, \ldots, n-3$. On définit $L(i+3) - L(i+2) = \omega$ si $L(i+2) = -\infty$ ou $L(i+3) = -\infty$.

Un mot w de $(\mathbb{Z} \cup \{\omega\})^*$ est un *mot feuille*, s'il existe un graphe connexe G tel que $\Delta L_G = w$. On dit alors que w est le mot feuille de G.

Pour motiver le décalage de 3 dans la définition précédente, on remarque facilement qu'il y a, à isomorphisme près, un unique arbre à $0, 1, 2$ ou 3 sommets. Ces arbres sont respectivement le graphe vide, le graphe à un seul sommet et les chaînes à deux et trois sommets. Ainsi, pour un graphe connexe G à trois sommets ou plus et non isomorphe au graphe complet, on a $L_G(0) = L_G(1) = 0$ et $L_G(2) = L_G(3) = 2$. Pour la plupart des graphes connexes, la suite $(L_G(i) - L_G(i-1))_{i=1,2,\ldots,|G|}$ débute donc par $0, 2, 0$. Pour éviter l'information redondante dans les mots feuilles,

on choisit d'utiliser une dérivée discrète tronquée de ses trois premières lettres par rapport la définition habituelle dans la définition ci-dessus.

Pour un exemple de mots feuilles, on considère G le graphe illustré à la figure 3.2. Le préfixe 020 de sa dérivée discrète est omit et le mot feuille de G est 110101.

i	0	1	2	3	4	5	6	7	8	9
$L_G(i)$	0	0	2	2	3	4	4	5	5	6
$L_G(i+1) - L_G(i)$	0	2	0	1	1	0	1	0	1	

ΔL_G

Figure 3.2 Un graphe G, sa fonction feuille L_G, sa dérivée discrète et son mot feuille ΔL_G.

Comme première piste de solution au problème 3.2, on démontre le lemme suivant :

Lemme 3.4. *L'ensemble $S = \{1, 0, -1, -2, \ldots, \omega\}$ est le plus petit ensemble tel que $\Delta L_G \in S^*$ pour tout graphe G à trois sommets ou plus.*

Démonstration. On montre d'abord que pour n'importe quel graphe G, son mot feuille ne peut contenir un entier $k > 1$. Supposons qu'il existe un graphe G et un entier i tel que $L_G(i+1) - L_G(i) = k$. Alors, G possède un sous-arbre induit T à $i+1$ sommet et $L_G(i) + k$ feuilles. Si l'on retire une feuille à T, on obtient alors T' un sous-arbre induit à i sommets. Si $i \leq 2$ alors la différence $L_G(i+1) - L_G(i)$ n'apparaît pas dans le mot feuille. Si $i > 2$, toutes les feuilles de T, sauf celle que l'on a retirée, restent des feuilles de T'. Donc, G contient un sous-arbre induit de taille i à $L_G(i) + k - 1$ feuilles. Ceci contredit la définition de fonction feuille, car $k - 1 > 0$.

Pour la minimalité de l'ensemble S, rappelons que la fonction feuille du graphe roue W_n à $n+1$ sommets est :

$$L_{W_n}(i) = \begin{cases} 0 & \text{si } i = 0, 1 \\ 2 & \text{si } i = 2 \\ i-1 & \text{si } 3 \leq i \leq \lfloor \frac{n}{2} \rfloor + 1 \\ 2 & \text{si } \lfloor \frac{n}{2} \rfloor + 2 \leq i \leq n-1 \\ -\infty & \text{si } i = n, n+1 \end{cases}$$

On voit facilement que les valeurs $0, 1, \omega$ sont possibles pour $L(i+3) - L(i+2)$. Pour tout $k \geq 1$, on retrouve la lettre $-k$ dans le mot feuille de W_{2k+4} en calculant

$$L(k+4) - L(k+3) = 2 - (k+2) = -k.$$

□

La figure 3.3 montre un cas où l'on retrouve la lettre -3 dans le mot feuille. En effet, pour le graphe W_{10}, on a $L_{W_{10}}(7) - L_{W_{10}}(6) = 2 - 5 = -3$.

Le lemme précédent permet par exemple de démontrer que la suite

$$(0, 0, 2, 2, 3, 3, 3, 4, 6, 6, -\infty, -\infty)$$

n'est pas une suite feuillue, car le mot feuille d'un graphe la réalisant contiendrait la lettre 2.

(a) (b)

Figure 3.3 Le graphe roue W_{10} avec deux sous-arbres pleinement feuillus de taille (a) 6 et (b) 7 en bleu.

On peut aussi déterminer, à partir de son mot feuille, si un graphe est un arbre.

Lemme 3.5. *Soit G un graphe de taille au moins 4. Alors* $\mathrm{Alph}(\Delta L_G) \subseteq \{0,1\}$ *si et seulement si G est un arbre.*

Démonstration. $\Longrightarrow$: Si G est un arbre, alors sa fonction feuille est croissante. En effet, si $L_G(i) = \ell$ alors G contient un sous-arbre induit T avec i sommets et ℓ feuilles. Soit v un sommet de $G \backslash T$ adjacent à T dans G. Soit T', le sous-graphe induit par $T \cup \{v\}$. T' est forcément un arbre, car G est acyclique. De plus, T' a au moins ℓ feuilles. En effet, v est forcément une nouvelle feuille de T' et au plus une feuille de T devient un nœud interne dans T', lors de l'adjonction d'un sommet. Ainsi $L_G(i+1) \geq \ell \geq L_G(i)$, et la fonction feuille de G est croissante. Comme $L_G(0) = 0$, la fonction ne prend donc jamais $-\infty$ comme valeur et son mot feuille ne contient donc pas ω. Le mot feuille de G ne peut pas non plus contenir d'entier négatif, car sa fonction feuille est croissante. Ainsi, par le lemme 3.4, $\mathrm{Alph}(\Delta L_G) \subseteq \{0,1\}$.

$\Longleftarrow$: Soit $n = |G|$. Si $\mathrm{Alph}(\Delta L_G) \subseteq \{0,1\}$ alors $L(n) - L(n-1) \neq \omega$. On a donc que $L(n) \neq -\infty$. On en déduit donc que G contient un sous-arbre induit à n sommets. Or le seul sous-graphe induit de G à n sommets est G, ce dernier doit donc forcément être un arbre. □

À partir du mot feuille, on peut donc facilement déterminer si deux arbres ont la même fonction feuille en utilisant le lemme ci-dessous dont la preuve découle directement de la définition 3.3.

Lemme 3.6. *Deux fonctions $L_1, L_2 : \{0,1,\ldots,n\} \to \mathbb{N}$ sont identiques si et seulement si $(L_1(0), L_1(1), L_1(2), L_1(3)) = (L_2(0), L_2(1), L_2(2), L_2(3))$ et $\Delta L_1 = \Delta L_2$.*

En utilisant la définition du mot feuille, on déduit aussi directement que pour un

arbre T et toute paire d'indices $3 \leq i \leq j \leq n$

$$L_T(j) - L_T(i) = |\text{suff}_{j-i}(\text{pref}_{j-3}(\Delta L_T))|_1. \tag{3.1}$$

En d'autres mots, la différence entre la fonction feuille en deux points est le nombre de 1 dans le facteur « correspondant » du mot feuille du graphe. La formule peut être vérifiée sur l'arbre G de la figure 3.2. Par exemple, pour $j = 7$ et $i = 4$,

$$L_T(7) - L_T(4) = 2 = |101|_1 = |\text{suff}_3(\text{pref}_4(\Delta L_T))|_1.$$

Le problème de réalisation de la fonction feuille étant assez difficile dû la grande liberté possible dans les graphes, et sa solution restant inconnue à ce jour, on se concentre dans le reste de ce chapitre sur la réalisation de la fonction feuille pour les graphes chenilles.

Problème 3.7 (Réalisation de la fonction feuille pour les chenilles)**.** Étant donnée une suite $\ell = (\ell_0, \ell_1, \ldots, \ell_n)$ d'éléments de $\mathbb{N}$, existe-t-il un graphe chenille C tel que L_C, la fonction feuille de C, prenne ℓ_i comme valeur en i ?

Pour résoudre le problème 3.7, on utilise des outils de la combinatoire des mots. Dans les prochaines sections, on introduit donc les mots préfixes normaux. On développe ensuite des outils sur les chenilles pour finalement établir des liens entre les mots feuilles des chenilles et les mots préfixes normaux.

3.1 Mots préfixes normaux

Les définitions de base sur les mots ayant été introduites dans la section 1.3, on se concentre dans cette section sur les mots préfixes normaux.

Pour l'étude des chenilles, on utilise les mots binaires sur l'alphabet $\{0, 1\}$. Les définitions et résultats présentés ci-dessous (jusqu'au théorème 3.11) sont issus de (Burcsi *et al.*, 2017). Sur les mots binaires, on définit la fonction « maximum de 1 ».

Définition 3.8. Soit w un mot de $\{0,1\}^*$. On définit, pour chaque $0 \leq k \leq |w|$,

$$F_1(w,k) = \max\{|v|_1 : v \in \mathrm{Fact}_k(w)\},$$

qui indique le nombre maximal de 1 dans un facteur de longueur de k de w. On dénote aussi $F_1(w)$ la fonction $k \mapsto F_1(w,k)$.

Par exemple, pour $w = 11010010$, $(F_1(w,k))_{k=0,1,\ldots,|w|} = (0,1,2,2,3,3,3,4,4)$. La fonction F_1 permet d'établir une relation d'équivalence sur les mots binaires :

Définition 3.9. Deux mots binaires de même longueur u et w sont dits F_1-*équivalents* si $F_1(w,k) = F_1(u,k)$ pour tous $0 \leq k \leq |w|$. Lorsque u et w sont F_1-équivalents, on note $w \equiv u$.

Par exemple, sous la relation $\equiv$, la classe d'équivalence du mot 110100110 est

$$\{110100110, 110010110, 011010011, 011001011\}.$$

Notons que deux mots u et w sont équivalents, si et seulement si, $F_1(w) = F_1(u)$ et que les classes d'équivalence sont fermées sous l'opération d'image miroir sur les mots. On remarque aussi que dans la classe d'équivalence ci-dessus, le mot 110100110 joue un rôle particulier, car la valeur de $F_1(w,k)$ est toujours réalisée par le préfixe de longueur k. Les mots ayant cette propriété sont appelés mots préfixes normaux.

Définition 3.10. Soit $w \in \{0,1\}^*$. On dit que w est préfixe normal si $F_1(w,k) = |\mathrm{pref}_k(w)|_1$ pour tout $0 \leq k \leq |w|$.

Grosso modo, les mots préfixes normaux sont ceux qui sont les plus « denses » en 1 dans leurs préfixes. Par exemple, le mot 11000110011 n'est pas préfixe normal, car $|110011|_1 = 4$ mais $|\mathrm{pref}_6(11000110011)|_1 = 3$. Par contre, le mot 11001100011

est bien préfixe normal. Les mots préfixes normaux de longueur 5 sont : 11111, 11110, 11101, 11100, 11011, 11010, 11001, 11000, 10101, 10100, 10010, 10001, 10000, 00000.

Un premier intérêt pour les mots préfixes normaux est qu'ils forment un système de représentants pour les classes de la relation d'équivalence $\equiv$.

Théorème 3.11. *Pour tout $w \in \{0,1\}^*$, il existe un unique mot préfixe normal tel que $w \equiv w'$.*

Un tel représentant est appelé *forme préfixe normale* du mot.

Remarque 3.12. L'introduction des mots préfixes normaux est à l'origine motivée par la résolution du problème de correspondance abélienne de motif dans les mots binaires (*Binary Jumbled Pattern Matching*, BJPM). Ce problème consiste à décider s'il existe un facteur d'un mot binaire qui contient exactement un certain nombre de 0 et de 1. Une version indexée de ce problème (*Indexed Binary Jumbled Pattern Matching*) autorise la création d'un index de taille linéaire en la longueur du mot pour répondre à cette question. Il s'avère en fait que le calcul de la forme préfixe normale d'un mot est très utile pour la création d'un tel index.

Une question naturelle est celle de générer les mots préfixes normaux de taille fixée. À ce jour, le meilleur algorithme connu génère tous les mots préfixes normaux de taille n avec un délai dans $\mathcal{O}(n)$ (Cicalese *et al.*, 2018). Un autre algorithme présenté dans (Burcsi *et al.*, 2014) a, quant à lui, un délai amorti dans $\mathcal{O}(n)$, mais il est conjecturé que ce dernier soit dans $\mathcal{O}(\log n)$. Les problèmes de dénombrement, à la fois des mots préfixes normaux et ceux liés aux classes d'équivalence F_1, restent ouverts.

On termine cette section avec un résultat sur les mots qui ne sont pas préfixes normaux. Ce résultat sera utile pour démonter les résultats principaux à la fin de

ce chapitre.

Lemme 3.13. *Soit w un mot binaire sur $\{0,1\}$ qui n'est pas préfixe normal, alors il existe deux mots abéliens équivalents u et u', tels que $u0 \in \mathrm{Pref}(w)$ et $1u' \in \mathrm{Fact}(w)$.*

Démonstration. Comme w n'est pas préfixe normal, il existe un préfixe p et un facteur f, tel que $|p|_1 < |f|_1$ et $|p| = |f|$. Sans perte de généralité, supposons que p et f soit de longueur minimale. Soit u, u', deux mots et a, b, deux lettres tels que $p = ua$ et $f = bu'$. Par minimalité de p et f, on sait que $|u|_1 \geq |u'|_1$. Comme on a que $|p|_1 < |f|_1$, on en déduit que $|u|_1 + |a|_1 < |u'|_1 + |b|_1$. Ainsi,

$$|a|_1 < |u'|_1 - |u|_1 + |b|_1 \leq |b|_1.$$

Comme a et b sont dans $\{0,1\}$, on doit donc avoir $a = 0$ et $b = 1$. En remplaçant les valeurs de a et b, on trouve que $|u|_1 = |u'|_1$ et donc u et u' doivent être abéliens équivalents. □

3.2 Graphes chenilles et sous-arbres induits pleinement feuillus

Rappelons qu'un graphe est une chenille, si ce dernier est un arbre tel que lorsqu'on lui retire toutes ses feuilles, il ne reste qu'une chaîne appelée colonne (voir la définition 1.3 et la figure 1.3).

3.2.1 Monoïde des chenilles orientées

Par commodité, on considére les graphes chenilles comme des objets orientés. Pour ce faire, on associe à la colonne d'une chenille, un sens. Toute chenille peut alors être représentée par un n-uplet d'entiers, nommé *séquence des degrés*, représentant la suite des degrés des sommets sur sa colonne. Par exemple, la chenille de la figure 3.4 peut être lue de gauche à droite pour obtenir la séquence des degrés

$(3, 2, 5, 2, 6, 2, 3)$ ou de droite à gauche pour obtenir $(3, 2, 6, 2, 5, 2, 3)$. Par convention, lorsqu'on illustre une chenille, on suppose qu'elle est orientée de gauche à droite.

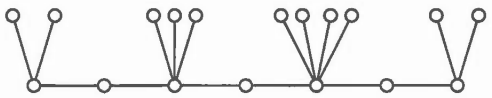


Figure 3.4 Chenille non orientée

En exprimant une chenille comme un n-uplet, on obtient facilement son nombre de sommets ainsi que son nombre de feuilles. En effet, pour une chenille $C = (c_1, \ldots, c_n)$ on a que

$$|C| = \sum_{i=1}^{n} c_i - n + 2 \tag{3.2}$$

$$|C|_{\text{feuilles}} = \sum_{i=1}^{n} c_1 - 2n + 2 \tag{3.3}$$

Sur les chenilles orientées, on peut définir une opération binaire qui permet de joindre deux chenilles en une seule. On nomme cette opération la *greffe*.

Définition 3.14. Pour $C = (c_1, \ldots, c_m)$ et $C' = (c'_1, \ldots, c'_n)$ des chenilles orientées, on définit la *greffe* de C et C' dénotée $C \diamond C'$ par

$$C \diamond C' = (c_1, \ldots, c_{m-1}, c_m + c'_1 - 2, c'_2, \ldots, c'_n)$$

Bien que cette définition semble un peu ad hoc, elle est motivée par le fait que la plus petite chenille, la chenille (2), soit l'élément neutre de la greffe, c'est-à-dire que lorsque l'on greffe (2) à C on retrouve C. Ce choix est motivé par la construction de la définition 3.15 permettant d'obtenir l'isomorphisme du lemme 3.16. La figure 3.5 illustre la greffe de deux chenilles orientées. Les deux arêtes rouges sont fusionnées ainsi que la paire d'arêtes bleues.

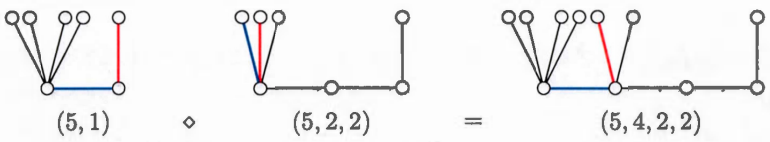


Figure 3.5 Greffe des chenilles $(5,1)$ et $(5,2,2)$.

L'opération de greffe est compatible avec les fonctions $|\cdot|$ et $|\cdot|_{\leaf}$ dans le sens suivant :

$$|C \diamond C'| = |C| + |C'| - 3 \tag{3.4}$$

$$|C \diamond C'|_{\leaf} = |C|_{\leaf} + |C'|_{\leaf} - 2 \tag{3.5}$$

On peut construire une chenille à partir d'un mot binaire sur $0,1$ en utilisant la fonction RC.

Définition 3.15. Soit $w \in \{0,1\}^*$ un mot binaire. Si $w = \varepsilon$ alors $\mathrm{RC}(w) = (2)$ et si $w = ua$ avec u un mot et a une lettre alors

$$\mathrm{RC}(ua) = \begin{cases} \mathrm{RC}(u) \diamond (2,2) & \text{si } a = 0 \\ \mathrm{RC}(u) \diamond (3) & \text{si } a = 1 \end{cases}$$

$\mathrm{RC}(w)$ est appelée *chenille de lecture* (*reading caterpillar*) de w.

Essentiellement, pour construire la chenille de lecture, on lit le mot de gauche à droite. Si on lit un 1, on attache une feuille supplémentaire au dernier sommet de la colonne. Si on lit un 0, on étend la colonne en transformant une feuille du dernier sommet en sommet interne en lui ajoutant une feuille. La figure 3.6 illustre ces deux cas. Pour le mot $u = 1101011$ dont la chenille de lecture est $(3,3,3)$, on retrouve à la figure 3.6 (a) la construction de $\mathrm{RC}(u0)$ et à la figure 3.6 (b) celle de $\mathrm{RC}(u1)$.

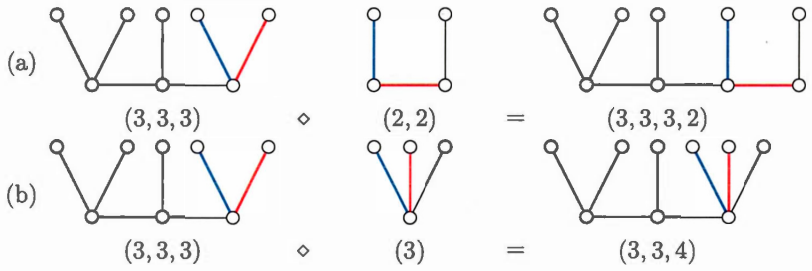


Figure 3.6 Construction de la chenille de lecture (a) de $u0$ et (b) de $u1$.

On voit facilement qu'il est aussi possible d'exprimer la chenille de lecture par des opérations arithmétiques simples sur les n-uplets. En effet, si $\mathrm{RC}(u) = (c_1, \ldots, c_k)$, alors

$$\mathrm{RC}(u0) = (c_1, \ldots, c_k, 2) \tag{3.6}$$

$$\mathrm{RC}(u1) = (c_1, \ldots, c_{k-1}, c_k + 1) \tag{3.7}$$

On a déjà remarqué que la chenille (2) est un élément neutre pour la greffe. On peut aussi facilement vérifier que $\diamond$ est associative. À partir des Équations (3.6) et (3.7), on peut montrer que RC est inversible avec l'inverse suivante :

$$\mathrm{RC}^{-1}(c_1, \ldots, c_k) = \begin{cases} \varepsilon & \text{si } (c_1, \ldots, c_k) = (2) \\ \mathrm{RC}^{-1}(c_1, \ldots, c_{k-1})0 & \text{si } c_k = 2 \\ \mathrm{RC}^{-1}(c_1, \ldots, c_k - 1)1 & \text{si } c_k > 2 \end{cases}$$

On note CW la fonction RC^{-1}. Pour une chenille C, on nomme *mot de chenille* le mot $\mathrm{CW}(C)$. La figure 3.7 donne un exemple de cette bijection. Finalement, comme RC et son inverse sont compatibles avec les opérations, on obtient le lemme suivant :

Lemme 3.16. *L'ensemble des chenilles orientées munies de l'opération de greffe forme un monoïde isomorphe au monoïde* $\{0, 1\}^*$ *muni de la concaténation.*

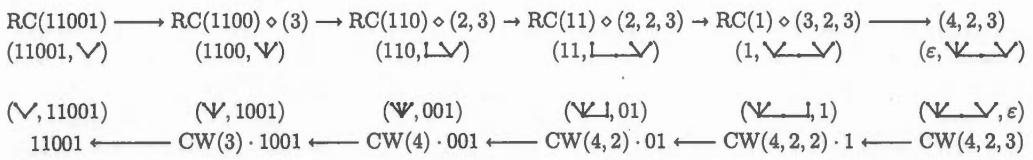


Figure 3.7 Bijection entre les mots binaires et les chenilles orientées.

3.2.2 Mots et chenilles

Dans cette section, on transpose des définitions bien connues dans l'univers des mots, en concepts sur les chenilles. On commence en adaptant les notions de préfixe et suffixe.

Définition 3.17. Soit C une chenille orientée, pour tout $3 \leq i \leq |C|$, on appelle *chenille gauche* et *chenille droite* les chenilles orientées :

$$\text{Left}_i(C) = \text{RC}(\text{pref}_{i-3}(\text{CW}(C))) \text{ et} \tag{3.8}$$

$$\text{Right}_i(C) = \text{RC}(\text{suff}_{i-3}(\text{CW}(C))) \tag{3.9}$$

Visuellement, pour trouver Left_i (resp. Right_i) d'une chenille, on garde les i nœuds les plus à gauche (resp. à droite) de la chenille. Pour la chenille $C = (3, 2, 5, 2, 6, 2, 3)$, la figure 3.8, montre la chenille $\text{Left}_7(C) = (3, 2, 3)$ en bleu et la chenille $\text{Right}_4(C) = (3)$ en rouge.

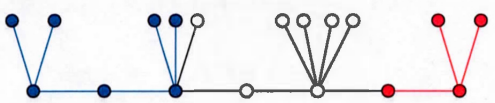


Figure 3.8 Chenille $C = (3, 2, 5, 2, 6, 2, 3)$ avec $\text{Left}_7(C)$ en bleu et $\text{Right}_4(C)$ en rouge.

Dans le lemme suivant, on démontre plusieurs correspondances entre les notations sur les mots et les notations sur les chenilles.

Lemme 3.18. *Soit w un mot binaire sur $\{0,1\}$, $a \in \{0,1\}$ et $3 \leq i \leq |w|+3$. Alors*

(i) $\mathrm{Left}_i(\mathrm{RC}(w)) = \mathrm{RC}(\mathrm{pref}_{i-3}(w))$;

(ii) $\mathrm{Right}_i(\mathrm{RC}(w)) = \mathrm{RC}(\mathrm{suff}_{i-3}(w))$;

(iii) $|\mathrm{RC}(w)| = |w| + 3$;

(iv) $|\mathrm{RC}(wa)|_{\text{@}} = |\mathrm{RC}(w)|_{\text{@}} + a$;

(v) $|\mathrm{RC}(w)|_{\text{@}} = |w|_1 + 2$;

(vi) $|\mathrm{Left}_i(\mathrm{RC}(w))|_{\text{@}} = |\mathrm{pref}_{i-3}(w)|_1 + 2$;

(vii) $|\mathrm{Right}_i(\mathrm{RC}(w))|_{\text{@}} = |\mathrm{suff}_{i-3}(w)|_1 + 2$.

Démonstration.

(i) $\mathrm{Left}_i(\mathrm{RC}(w)) = \mathrm{RC}(\mathrm{pref}_{i-3}(\mathrm{CW}(\mathrm{RC}(w)))) = \mathrm{RC}(\mathrm{pref}_{i-3}(w))$

La preuve des autres énoncés découlent aussi directement des définitions et observations précédentes. □

On termine cette section avec un lemme permettant de factoriser une chenille en deux sous-chenilles dont la greffe redonne la chenille d'origine.

Lemme 3.19. *Pour toute chenille C et tout entier $i \in \{3, 4, \ldots, |C|\}$,*

$$C = \mathrm{Left}_i(C) \diamond \mathrm{Right}_{|C|+3-i}(C).$$

Démonstration. Soit C une chenille. On déduit du lemme 3.18 (iii) que $|\mathrm{CW}(C)|$ est égal à $|C| - 3$. Ainsi $\mathrm{pref}_{i-3}(\mathrm{CW}(C)) \cdot \mathrm{suff}_{|C|-i}(\mathrm{CW}(C)) = \mathrm{CW}(C)$. On a donc

$$\begin{aligned}\mathrm{Left}_i(C) \diamond \mathrm{Right}_{|C|+3-i}(C) &= \mathrm{RC}(\mathrm{pref}_{i-3}(CW(C))) \diamond \mathrm{RC}(\mathrm{suff}_{|C|-i}(\mathrm{CW}(C))) \\ &= \mathrm{RC}(\mathrm{pref}_{i-3}(CW(C)) \cdot \mathrm{suff}_{|C|-i}(\mathrm{CW}(C)))\end{aligned}$$

$$= \mathrm{RC}(\mathrm{CW}(C)) = C$$

□

Ainsi, pour toute taille i, on peut trouver une « factorisation » d'une chenille en deux chenilles, dont une est de taille i. Pour l'instant, un des obstacles à la généralisation à des familles de graphe plus grande des résultats de la section 3.3 semble venir du fait qu'il n'existe pas une telle factorisation pour les arbres en général. Il ne semble pas, par exemple, y avoir de manière naturelle de factoriser l'arbre ternaire complet de hauteur 2 (voir figure 3.9) en un arbre de taille 5 et un arbre de taille 8. On voit d'ailleurs que le mot feuille de cet arbre est 1101011011 qui n'est pas préfixe normal puisque le suffixe 11011 contient plus de 1 que le préfixe de longueur 5.

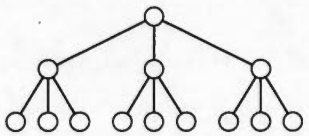


Figure 3.9 Arbre ternaire complet de hauteur 2.

3.2.3 Sous-chenille

Pour calculer la fonction feuille d'une chenille, on doit considérer tous les sous-arbres induits de celle-ci. Or, tout sous-arbre d'une chenille doit forcément être une chenille. On peut, de plus, facilement caractériser une chenille qui est sous-arbre (on dit aussi *sous-chenille*) d'une autre. En effet, pour deux chenilles $C = (c_1, \ldots, c_n)$ et $C' = (c'_1, \ldots, c'_k)$, on a que C' est sous-chenille de C (on note $C' \preceq C$) si et seulement s'il existe un certain décalage $0 \leq i \leq n-k$, tel que $c'_j \leq c_{i+j}$ pour tout $j \in \{1, 2, \ldots, k\}$. En particulier, la chenille de lecture d'un facteur d'un mot w est toujours sous-chenille de la chenille de lecture de w.

Lemme 3.20. *Soit w un mot binaire et u un facteur de w. Alors* $\mathrm{RC}(u)$ *est une sous-chenille de* $\mathrm{RC}(w)$.

Démonstration. Si u est un facteur de w alors il existe des mots x, y tels que $w = xuy$. Conséquemment $\mathrm{RC}(w) = \mathrm{RC}(x) \diamond \mathrm{RC}(u) \diamond \mathrm{RC}(y)$. Posons

$$\mathrm{RC}(x) = (\alpha_1, \ldots, \alpha_\ell),$$
$$\mathrm{RC}(u) = (\beta_1, \ldots, \beta_m) \text{ et}$$
$$\mathrm{RC}(y) = (\gamma_1, \ldots, \gamma_n).$$

On a alors que

$$\mathrm{RC}(w) = (\alpha_1, \ldots, \alpha_{\ell-1}, \alpha_\ell + \beta_1 - 2, \beta_2, \ldots, \beta_{m-1}, \beta_m + \gamma_1 - 2+, \gamma_2, \ldots, \gamma_n).$$

Comme α_ℓ et γ_1 sont des degrés de sommets internes, ils valent au moins 2. Ainsi, en utilisant un décalage de $l - 1$, on trouve bien que $\mathrm{RC}(u)$ est sous-chenille de $\mathrm{RC}(w)$. □

Il découle directement du lemme 3.20 que $\mathrm{Left}_i(C)$ et $\mathrm{Right}_i(C)$ sont des sous-chenilles de C. La relation « être sous-chenille » donne une relation d'ordre sur l'ensemble des chenilles orientées. La figure 3.10 illustre les premiers niveaux du diagramme de Hasse de cette relation d'ordre. On remarque facilement que cet ensemble partiellement ordonné est rangé par le nombre de sommets des chenilles et qu'il ne s'agit pas d'un treillis puisque les éléments $(2, 1)$ et $(1, 0, 1)$ sont couverts à la fois par $(1, 1, 1)$ et $(2, 0, 1)$.

Lemme 3.21. *Si w est un mot binaire, alors pour tout entier $i \in \{3, \ldots, |w|+3\}$, w contient un facteur u de longueur $i - 3$ tel que* $\mathrm{RC}(u)$ *soit une sous-chenille pleinement feuillue.*

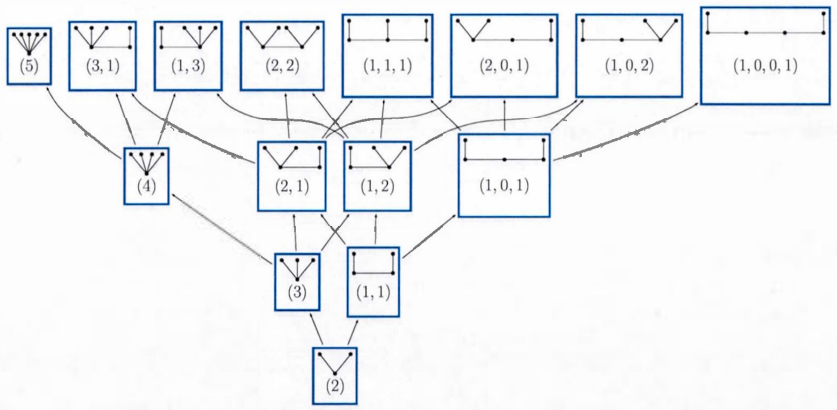


Figure 3.10 Ensemble partiellement ordonné des chenilles orientées.

Démonstration. Posons $\mathrm{RC}(w) = (r_1, \ldots, r_n)$. Soit $S = (s_1, \ldots, s_k)$ une sous-chenille pleinement feuillue de taille i de $\mathrm{RC}(w)$ qui n'est pas forcément la chenille de lecture d'un facteur de w. Comme S est une sous-chenille de $\mathrm{RC}(w)$, on a, pour un certain décalage ℓ, que $s_m \leq r_{\ell+m}$ pour tout $1 \leq m \leq k$.

Posons $d = \sum_{m=2}^{k-1} r_{\ell+m} - s_m$. On montre par contradiction que $s_1 + s_k - 4 \geq d$. Supposons que $s_1 + s_k - 4 < d$. L'idée est de voir que l'on peut alors construire une sous-chenille de $\mathrm{RC}(w)$ de même taille que S mais avec plus de feuilles. On commence par enlever $s_k + s_1 - 3$ sommets en considérant la chenille $(s_2, s_3, \ldots, s_{k-1}, 2)$. Notons que la taille de cette chenille est $|S| - s_1 - s_k + 3$ et son nombre de feuilles est $|S|_{\mathscr{D}} - s_k - s_1 + 2$. Par hypothèse, on peut alors ajouter $s_1 + s_k - 3$ feuilles en les attachant aux sommets 2 à $k-1$. Précisément, on choisit un entier $z \in \{2, \ldots k-1\}$ maximal tel que $s_1 + s_k - 3 \geq \sum_{m=2}^{z} r_{\ell+m} - s_m$. On fixe $\xi = s_1 + s_k - 3 - \sum_{m=2}^{z} r_{l+m} - s_m$. On a alors que la chenille

$$S' = (r_{\ell+2}, r_{\ell+3}, \ldots, r_{\ell+z}, s_{z+1} + \xi + s_{z+2}, \ldots, s_{k-1}, 2)$$

est une sous-chenille de $\mathrm{RC}(w)$. De plus, S' satisfait $|S'| = |S|$ et $|S'|_{\mathscr{D}} = |S|_{\mathscr{D}} + 1$ ce qui contredit le fait que S soit pleinement feuillue.

Par conséquent, on sait que $s_1 + s_k - 4 \geq d$. Ainsi, $S' = (s'_1, r_{\ell+2}, \ldots, r_{l+k-1}, s'_k)$ avec $s'_1 = s_1 - \min(s_1 - 2, d)$ et $s'_k = s_1 + s_k - d - s'_1$ est une sous-chenille de $\mathrm{RC}(w)$. De plus, elle a le même nombre de sommets et de feuilles que S. Elle est donc pleinement feuillue.

On voit facilement, pour

$$A = (r_1, \ldots, r_l, r_{l+1} - s'_1 + 2) \qquad \text{et}$$
$$B = (r_{l+k} - s'_k + 2, r_{l+k+1}, \ldots, r_n),$$

que $\mathrm{RC}(w) = A \diamond S' \diamond B$ et que avec A et B des sous-chenilles de $\mathrm{RC}(w)$. Ainsi, $w = \mathrm{CW}(A)\mathrm{CW}(S')\mathrm{CW}(B)$ et donc $CW(S')$ est facteur de w. On a donc bien qu'il existe une sous-chenille pleinement feuillue de taille i de $\mathrm{RC}(w)$ qui est la chenille de lecture d'un facteur de w. □

On remarque que toute sous-chenille pleinement feuillue n'est pas forcément la chenille de lecture d'un facteur. Par exemple, la chenille de lecture de $w = 00110101100$ illustrée à la figure 3.11 contient une sous-chenille pleinement feuillue qui n'est pas la chenille de lecture d'un facteur. Sa fonction feuille vaut 5 pour la taille 8 et la sous-chenille pleinement feuillue en rouge est la chenille de lecture de 11001 qui n'est pas facteur de w. Par contre, la chenille bleue est la chenille de lecture de 11010 qui est facteur de w. Il y a donc une sous-chenille pleinement feuillue qui est la chenille de lecture d'un facteur de w. La transformation de S vers S' décrite dans la preuve précédente, est celle qui transforme la chenille rouge en la chenille bleue.

3.3 Résultat pour les graphes chenilles

On présente dans cette section deux nouveaux résultats liant graphes chenilles et mots préfixes normaux. On montre d'abord que les fonctions feuilles dont la dérivée discrète est un mot préfixe normal, sont exactement les fonctions feuilles qui

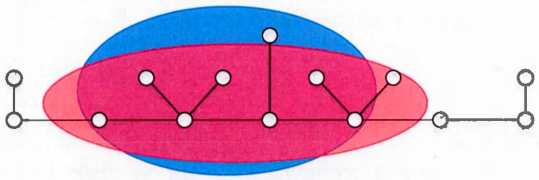


Figure 3.11 Deux sous-chenilles pleinement feuillues (en rouge et en bleu) de RC(00110101100).

peuvent être réalisées par les graphes chenilles. Puis, on montre que les chenilles qui ont la même fonction feuille sont celles dont les mots sont F_1-équivalent.

Théorème 3.22. *Soit* $L : \{0, 1, 2, \ldots, n\} \to \mathbb{N}$ *une fonction telle que* $L(0) = L(1) = 0$ *et* $L(2) = L(3) = 2$. *Alors il existe une chenille* C *telle que* $L = L_C$ *si et seulement si* ΔL *est un mot préfixe normal.*

Démonstration. $\Longleftarrow$: Soit w un mot préfixe normal et $3 \leq i \leq |w| + 3$. On montre que :

(i) $L_{\mathrm{RC}(w)}(i) = |\mathrm{Left}_i(\mathrm{RC}(w))|_{\text{◎}}$ pour $3 \leq i \leq |w| + 3$;

(ii) $\Delta L_{\mathrm{RC}(w)} = w$.

(i) Il est clair que $L_{\mathrm{RC}(w)}(i) \geq |\mathrm{Left}_i(\mathrm{RC}(w))|_{\text{◎}}$ puisque $\mathrm{Left}_i(\mathrm{RC}(w))$ est une sous-chenille de taille i de $\mathrm{RC}(w)$. Il reste à prouver que $L_{\mathrm{RC}(w)}(i) \leq |\mathrm{Left}_i(\mathrm{RC}(w))|_{\text{◎}}$.

Par le lemme 3.21, w contient un facteur u de longueur $i - 3$ tel que $\mathrm{RC}(u)$ soit une sous-chenille pleinement feuillue de $\mathrm{RC}(w)$. Conséquemment,

$$\begin{aligned}
L_{\mathrm{RC}(w)}(i) &= |\mathrm{RC}(u)|_{\text{◎}} \\
&= |u|_1 + 2 && \text{(par le lemme 3.18 (v))} \\
&\leq F_1(w, i-3) + 2 && \text{(par la définition 3.8)} \\
&= |\mathrm{pref}_{i-3}(w)|_1 + 2 && \text{(car } w \text{ est préfixe normal)} \\
&= |\mathrm{Left}_i(\mathrm{RC}(w))|_{\text{◎}}. && \text{(par le lemme 3.18 (vi))}
\end{aligned}$$

(ii) Pour $1 \leq i \leq |w|$, on a

$$\begin{aligned}
&\Delta L_{\mathrm{RC}(w)}(i) \\
&\quad = L_{\mathrm{RC}(w)}(i+3) - L_{\mathrm{RC}(w)}(i+2) && \text{(par la définition 3.3)} \\
&\quad = |\mathrm{Left}_{i+3}(\mathrm{RC}(w))|_{\text{🍎}} - |\mathrm{Left}_{i+2}(\mathrm{RC}(w))|_{\text{🍎}} && \text{(par (i))} \\
&\quad = (|\mathrm{pref}_i(w)|_1 + 2) - (|\mathrm{pref}_{i-1}(w)|_1 + 2) && \text{(par le lemme 3.18 ((vi)))} \\
&\quad = w_i,
\end{aligned}$$

comme voulu.

Étant donnée une fonction L, telle que $w = \Delta L$ soit préfixe normal, on sait par (ii) que la chenille $\mathrm{RC}(w)$ satisfait $\Delta L_{\mathrm{RC}(w)} = w = \Delta L$. On sait aussi que $(L_{\mathrm{RC}(w)}(i))_{i=0,1,2,3} = (0,0,2,2)$. Conséquemment, par le lemme 3.6, $L_{\mathrm{RC}(w)} = L$.

$\Longrightarrow$: On procède par contradiction. Soit C, une chenille et L_C, sa fonction feuille. Supposons que ΔL_C ne soit pas préfixe normal. Par le lemme 3.13, il existe deux mots, p et f, de même longueur, tels que $|p|_1 = |f|_1$, $p0$ soit un préfixe de ΔL_C et $1f$ soit facteur de ΔL_C. Soit $i-3$ l'indice de la dernière lettre d'une occurrence du facteur $1f$ dans ΔL_C. On sait alors que $i-3$ est tel que $1f = \mathrm{suff}_{|1f|}(\mathrm{pref}_{i-3}(\Delta L_C))$. Soit S, une sous-chenille de C, telle que $|S|_{\text{🍎}} = L_C(i)$ et $|S| = i$.

Fixons $A = \mathrm{Left}_{i-|1f|}(S)$ et $B = \mathrm{Right}_{|1f|+3}(S)$. Alors, par le lemme 3.19, on sait que $S = A \diamond B$. Ainsi,

$$\begin{aligned}
|B|_{\text{🍎}} &= |S|_{\text{🍎}} - |A|_{\text{🍎}} + 2 && \text{(par l'équation (3.5))} \\
&= L_C(i) - |A|_{\text{🍎}} + 2 && \text{(par hypothèse)} \\
&\geq L_C(i) - L_C(i - |1f|) + 2 && \text{(par définition de } L_C\text{)} \\
&= |\mathrm{suff}_{|1f|}(\mathrm{pref}_{i-3}(\Delta L_C))|_1 + 2 && \text{(par l'équation (3.1))}
\end{aligned}$$

$$
\begin{aligned}
&= |1f|_1 + 2 && \text{(par hypothèse)} \\
&= |f|_1 + 3 \\
&= |p|_1 + 3 \\
&> |p|_1 + 2 \\
&= |p0|_1 + 2 \\
&= L_C(|p0| + 3) && \text{(par l'équation (3.1) avec } i = 3)
\end{aligned}
$$

ce qui est absurde puisque $|B| = |p0| + 3$ et $B \preceq S$. □

Théorème 3.23. *Soit deux mots binaires w et w'. Alors les chenilles de lecture de w et w' ont la même fonction feuille si et seulement si w est F_1-équivalent à w'.*

Démonstration. $\Longrightarrow$: On procède par la contraposée. Supposons qu'il existe un certain $i \in \mathbb{N}$ tel que $F_1(w, i) \neq F_1(w', i)$. Sans perte de généralité, supposons $F_1(w, i) > F_1(w', i)$. Alors, w contient un facteur u de longueur i tel que pour tout facteur u' de longueur i dans w', $|u|_1 > |u'|_1$. Par le lemme 3.20, $\mathrm{RC}(u)$ est donc une sous-chenille de $\mathrm{RC}(w)$ avec strictement plus de feuilles que n'importe quelle sous-chenille de $\mathrm{RC}(w')$ qui est égale à $\mathrm{RC}(u')$ pour u' un facteur de longueur i de w'. On conclut que $L_{\mathrm{RC}(w)}(i) > L_{\mathrm{RC}(w')}(i)$ par le lemme 3.21.

$\Longleftarrow$: On procède par la contraposée. Posons $n = |w|$ et $w = w_1 \dots w_n$. Supposons que $L_{\mathrm{RC}(w)} \neq L_{\mathrm{RC}(w')}$. Sans perte de généralité, il existe donc $4 \leq i \leq n$ tel que $L_{\mathrm{RC}(w)}(i) > L_{\mathrm{RC}(w')}(i)$. Soit S une sous-chenille pleinement feuillue de $\mathrm{RC}(w)$ tel que $|S| = i$ et $|S|_{\mathscr{D}} = L_{\mathrm{RC}(w)}(i)$. Par le lemme 3.21, on peut supposer que S est tel qu'il existe un entier k satisfaisant

$$\mathrm{RC}(w) = \mathrm{Left}_k(\mathrm{RC}(w)) \diamond S \diamond \mathrm{Right}_{n+9-k-i}(\mathrm{RC}(w)).$$

Ainsi, par l'équation (3.5) et le lemme 3.18,

$$\begin{aligned}
&L_{\mathrm{RC}(w)}(i) = |S|_{\text{@}} \\
&= |\mathrm{RC}(w)|_{\text{@}} - |\mathrm{Left}_k(\mathrm{RC}(w))|_{\text{@}} - |\mathrm{Right}_{n+9-k-i}(\mathrm{RC}(w))|_{\text{@}} + 4 \\
&= |w|_1 + 2 - |\mathrm{pref}_{k-3}(w)|_1 - 2 - |\mathrm{suff}_{n+6-k-i}(w)|_1 - 2 + 4 \\
&= |w_{k-2}w_{k-1}\cdots w_{k+i-6}|_1 + 2.
\end{aligned}$$

Il a donc un facteur $u = w_{k-2}w_{k-1}\cdots w_{k+i-6}$ de longueur $i-3$ de w qui contient $L_{\mathrm{RC}(w)}(i) - 2$ fois la lettre 1. Conséquemment,

$$\begin{aligned}
|u|_1 &= L_{\mathrm{RC}(w)}(i) - 2 \\
&> L_{\mathrm{RC}(w')}(i) - 2 && \text{(par hypothèse)} \\
&\geq |u'|_1 && \text{(par le lemme 3.20)}
\end{aligned}$$

pour tout u' facteur de w' tel que $|u'| = i-3$. Ainsi, $F_1(w, i-3) > F_1(w', i-3)$ et $w \not\equiv w'$. □

CONCLUSION

En bref, ce travail met de l'avant les sous-arbres induits pleinement feuillus. Après le chapitre 1, présentant les notions préliminaires, le chapitre 2 attaque le problème de calcul de la fonction feuille à l'aide des configurations de graphes. Le chapitre 3 est centré sur le problème de réalisation de la fonction feuille avec une caractérisation complète des fonctions feuilles réalisées par les graphes chenilles.

Il reste plusieurs questions ouvertes ou qui pourraient être approfondies :

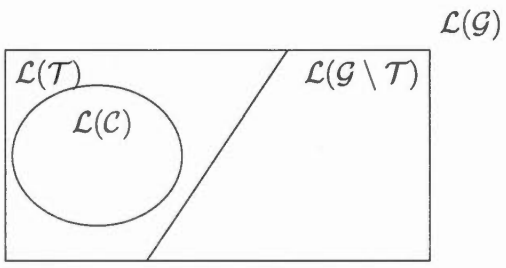
— On a établi une borne pour le nombre de feuilles possible pour les extensions des configurations. Toutefois, le scénario optimiste ne tient pas compte des nouveaux cycles qui pourraient se créer. Comme discuté dans la section 2.7, cette borne est donc d'une efficacité limitée sur les graphes denses. Il serait intéressant de trouver une meilleure borne pour ces cas. Dans un graphe dense, l'utilisation des cliques ce trouvant dans ce dernier pourrait être utilisé. Par exemple, lors de la construction d'un scénario d'extension optimiste, on pourrait ne pas ajouter plus de deux sommets par clique.

— En spécialisant l'algorithme 2.25 pour exploiter certaines symétries particuières des graphes hypercubiques, on arrive à calculer la fonction feuille jusqu'à la dimension 6. Pour ce faire, on élimine certains sous-arbres induits, s'ils peuvent être envoyés sur un autre sous-arbre induit déjà visité par un automorphisme de G. On pourrait accélérer le calcul de la fonction feuille en utilisant un générateur qui visite un seul représentant de chaque classe de sous-arbres induits équivalents à automorphisme près. Pour l'instant, aucun générateur de ce type n'est suffisamment rapide pour que l'on tire avantage

de la réduction du nombre de sous-arbres visités. L'étude du groupe d'automorphisme du graphe et de son action sur les sous-arbres induits semble toutefois une direction intéressante.

— On dénote $\mathcal{T}$ l'ensemble des arbres, $\mathcal{C}$ l'ensemble des chenilles et $\mathcal{G}$ l'ensemble des graphes. Le langage d'un ensemble de graphes E, noté $\mathcal{L}(E)$, est l'ensemble des mots feuilles de ces graphes. Le lemme 3.5 montre que $\mathcal{L}(\mathcal{T}) \cap \mathcal{L}(\mathcal{G} \backslash \mathcal{T}) = \emptyset$. On peut aussi voir que $\mathcal{L}(\mathcal{C}) \subsetneq \mathcal{L}(\mathcal{T})$ puisque le mot feuille de l'arbre ternaire complet de hauteur 2 (voir figure 3.9) n'est pas préfixe normal. Les relations connues entre les différents langages sont schématisées dans la figure ci-dessous. Le problème de déterminer les langages des arbres ou des graphes reste ouvert. Il existe aussi plusieurs arbres qui ne sont pas des chenilles bien que leur mot feuille soit préfixe normal. Le problème de caractériser complètement la classe de graphes dont les mots feuilles sont préfixes normaux reste donc ouvert.

On dit qu'un mot binaire est k-préfixe normal si tout facteur contient au plus k 1 de plus que le préfixe de même longueur. Cette définition relaxée de la préfixe normalité mène nautrellement au problème de déterminer les arbres dont les mots feuilles sont k-préfixe normaux. Ce problème semble une bonne porte d'entrée pour celui du langage des arbres.



RÉFÉRENCES

Abbott, H. L. et Katchalski, M. (1988). On the snake in the box problem. *J. Combin. Theory Ser. B*, *45*(1), 13–24. http://dx.doi.org/10.1016/0095-8956(88)90051-2

Abdenbi, M., Blondin Massé, A. et Goupil, A. (2018). On the maximal number of leaves in induced subtrees of series-parallel graphs. Dans *Proceedings of the 11th International Conference on Random and Exhaustive Generation of Combinatorial Structures, GASCom 2018, Athens, Greece, June 18-20, 2018.*, 24–31.

Blondin Massé, A., de Carufel, J. et Goupil, A. (2018a). Non saturated polyhexes and polyiamonds. Dans *Proceedings of the 11th International Conference on Random and Exhaustive Generation of Combinatorial Structures, GASCom 2018, Athens, Greece, June 18-20, 2018.*, 116–123.

Blondin Massé, A., de Carufel, J. et Goupil, A. (2018b). Saturated fully leafed tree-like polyforms and polycubes. *arXiv preprint arXiv :1803.09181.*

Blondin Massé, A., de Carufel, J., Goupil, A., Lapointe, M., Nadeau, É. et Vandomme, É. (2018c). Fully leafed induced subtrees. Dans C. Iliopoulos, H. W. Leong, et W.-K. Sung (dir.). *Combinatorial Algorithms*, 90–101., Cham. Springer International Publishing.

Blondin Massé, A., de Carufel, J., Goupil, A., Lapointe, M., Nadeau, E. et Vandomme, E. (2018). Leaf realization problem, caterpillar graphs and prefix normal words. *Theoret. Comput. Sci.*, *732*, 1–13. http://dx.doi.org/10.1016/j.tcs.2018.04.019

Blondin Massé, A., de Carufel, J., Goupil, A. et Samson, M. (2018). Fully leafed tree-like polyominoes and polycubes. Dans L. Brankovic, J. Ryan, et W. F. Smyth (dir.). *Combinatorial Algorithms*, 206–218., Cham. Springer International Publishing.

Blondin Massé, A. et Nadeau, E. (2017). Fully leafed induced subtrees. https://github.com/enadeau/fully-leafed-induced-subtrees. GitHub Repository.

Bollobás, B. (1998). *Modern Graph Theory*, volume 184 de *Graduate Texts in Mathematics*. Springer Verlag. http://dx.doi.org/10.1007/978-1-4612-0619-4

Boukerche, A., Cheng, X. et Linus, J. (2005). A performance evaluation of a novel energy-aware data-centric routing algorithm in wireless sensor networks. *Wireless Networks*, *11*(5), 619–635. http://dx.doi.org/10.1007/s11276-005-3517-6

Burcsi, P., Fici, G., Lipták, Z., Ruskey, F. et Sawada, J. (2014). On combinatorial generation of prefix normal words. Dans *Combinatorial pattern matching*, volume 8486 de *Lecture Notes in Comput. Sci.*, 60–69. Springer, Cham. http://dx.doi.org/10.1007/978-3-319-07566-2_7

Burcsi, P., Fici, G., Lipták, Z., Ruskey, F. et Sawada, J. (2017). On prefix normal words and prefix normal forms. *Theoret. Comput. Sci.*, *659*, 1–13. http://dx.doi.org/10.1016/j.tcs.2016.10.015

Cicalese, F., Lipták, Z. et Rossi, M. (2018). Bubble-flip—a new generation algorithm for prefix normal words. *Theoret. Comput. Sci.*, *743*, 38 – 52. http://dx.doi.org/10.1016/j.tcs.2018.06.021

Cook, S. A. (1971). The complexity of theorem-proving procedures. Dans *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, 151–158., New York, NY, USA. ACM. http://dx.doi.org/10.1145/800157.805047

Cormen, T. H., Leiserson, C. E., Rivest, R. L. et Stein, C. (2009). *Introduction to algorithms* (third éd.). MIT Press, Cambridge, MA.

Erdős, P., Saks, M. et Sós, V. T. (1986). Maximum induced trees in graphs. *J. Combin. Theory Ser. B*, *41*(1), 61–79. http://dx.doi.org/10.1016/0095-8956(86)90028-6

Erdős, P. et Gallai, T. (1961). Gráfok előírt fokú pontokkal (Graphs with prescribed degrees of vertices). *Mat. Lapok.*, *11*, 264–274.

Garey, M. R. et Johnson, D. S. (1979). *Computers and intractability*. W. H. Freeman and Co., San Francisco, Calif. A guide to the theory of NP-completeness, A Series of Books in the Mathematical Sciences.

Goupil, A., Pellerin, M.-E. et de Wouters d'oplinter, J. (2018). Partially directed snake polyominoes. *Discrete Applied Mathematics*, *236*, 223 – 234. http://dx.doi.org/10.1016/j.dam.2017.09.003

Hakimi, S. L. (1962). On realizability of a set of integers as degrees of the vertices of a linear graph. I. *J. Soc. Indust. Appl. Math.*, *10*(3), 496–506.

Harary, F. et Schwenk, A. J. (1973). The number of caterpillars. *Discrete Math.*, *6*, 359–365. http://dx.doi.org/10.1016/0012-365X(73)90067-8

Havel, V. (1955). Poznámka o existenci konečných grafů (A remark on the existence of finite graphs). *Časopis Pěst. Mat.*, *080*(4), 477–480.

Jaffe, A. M. (2006). The millennium grand challenge in mathematics. *Notices of the AMS*, *53*(6).

Kleitman, D. J. et West, D. B. (1991). Spanning trees with many leaves. *SIAM J. Discrete Math.*, *4*(1), 99–106. http://dx.doi.org/10.1137/0404010

Lothaire, M. (1997). *Combinatorics on words.* Cambridge Mathematical Library. Cambridge University Press, Cambridge. http://dx.doi.org/10.1017/CBO9780511566097

Nef cone from the associahedron

The claim of this section is to prove the following result suggested by Chapoton to Hugh in Luminy.

Claim 1. *Let Σ be the g-vector fan associated to $\mathbb{A}_c$. Then, the nef cone corresponding to the toric variety X_Σ is simplicial.*

1. Background about toric varieties

1.1. Affine toric varieties and toric varieties.

Definition 1. [1, 1.0.2] The *n-dimensional torus* T is the Zariski open subset $(\mathbb{C}^*)^n$ seen in $\mathbb{C}^n$. We embed T in $\mathbb{C}^{n+1}$ so that it becomes an affine variety. Its coordinate ring is $\mathbb{C}[X_1, X_1^{-1}, ..., X_n, X_n^{-1}]$.

I think it is really in $\mathbb{C}^n$.

Definition 2. A *torus* T is an affine variety which is isomorphic to $(\mathbb{C}^*)^n$. Since $(\mathbb{C}^*)^n$ is a group, T inherits a group structure from the isomorphism.

Definition 3. A *character* of a torus T is a morphism χ :T$\to \mathbb{C}^*$ that is also a group homorphism.

Remark 1. Let $m = (m_1, \cdots, m_n) \in \mathbb{Z}^n$. If $T = (\mathbb{C}^*)^n$, then we get a character as follow :

$$\begin{array}{rccc} \chi^m & : & T & \to & \mathbb{C}^* \\ & & (t_1, \cdots, t_n) & \mapsto & t_1^{m_1} \dots t_n^{m_n} \end{array}$$

It turns out that all the characters of $(\mathbb{C}^*)^n$ arise this way. Thus, $Hom((\mathbb{C}^*)^n, \mathbb{C}^*)$ is isomorphic to $\mathbb{Z}^n$ as a group. Now, if T is an arbitrary torus then we see $Hom(T, \mathbb{C}^*)$ as $\mathbb{Z}^n$, and if we take $m \in Hom(T, \mathbb{C}^*)$ then it is usual to say that m is the character χ^m.

Definition 4. [1, 1.1.3] An *affine toric variety* V is an affine variety containing a torus $T \simeq (\mathbb{C}^*)^n$ as an open subset such the action of T on itself extends to an algebraic action on V. We can show that an affine toric variety is irreducible so we don't need this assumption in the definition.

Definition 5. [1, 3.1.1] A *toric variety* X is an irreducible variety containing a torus $T \simeq (\mathbb{C}^*)^n$ as an open subset such that the action of T on itself extends to an algebraic action on X.

1.2. **Construction of affine toric varieties with a cone.** Let N a lattice with the basis $\{e_1, \cdots, e_n\}$ and let $M = \mathbb{Z}e_1^* \oplus \cdots \oplus \mathbb{Z}e_n^*$ be its dual lattice. Let σ a cone in $N_{\mathbb{R}} := N \otimes \mathbb{R}$. We will denote $\sigma^\vee = \{u \in M_{\mathbb{R}} \mid \langle u, v\rangle \geq 0 \ \forall v \in \sigma\}$. From now on we will look at only the "nice cones", i.e. the convex cones in $N_{\mathbb{R}}$ generated with some vectors of N.

Let $\sigma \subset N_{\mathbb{R}}$ and let the semi-group $S_\sigma = \sigma^\vee \cap M$. From this semi-group it is possible to get an algebra $\mathbb{C}[S_\sigma]$. This algebra is built as follow. It is a $\mathbb{C}$ vector space with $\{\chi^u\}_{u \in S_\sigma}$ as basis and the multiplication is given by $\chi^u \chi^v = \chi^{u+v}$. This is well explained page 16-17-18 of [1]. It turns out that $\mathbb{C}[S_\sigma]$ is an algebra of finite type, thus taking $Spec(\mathbb{C}[S_\sigma])$ we get an affine variety U_σ. We can also see U_σ as follow. Since $\mathbb{C}[S_\sigma]$ is a finitely generated algebra it is equal to $\mathbb{C}[X_1, ..., X_p]/I$ where $I = (F_1, ..., F_r)$. Then $U_\sigma = V(F_1, ..., F_r)$ and it is an affine toric variety.

1.3. **Construction of toric varieties with a fan.** The definition of fans is given in the definition 3.1.2. Let Σ a fan. The idea to build a toric variety is to take all the maximum cones in Σ and to glue them along some open subspaces. More precisely, if $\Sigma = \{0, \sigma_1, \sigma_2, \tau = \sigma_1 \cap \sigma_2$, and others cones$\}$ with σ_1 and σ_2 the only two maximal cones in Σ sharing the common face τ, then U_τ is seen as an open subset of U_{σ_1} and U_{σ_2} and thus there exist an isomorphism ϕ between these 2 copies of U_τ. Finally $X_\Sigma = U_{\sigma_1} \sqcup U_{\sigma_2} / \sim_\phi$. And this construction gives a toric variety.

1.4. **Main results useful for us.** There a several results connecting the properties of the cones/fans and the toric varieties. It might be useful to have them because when we work with the divisors there are always many conditions in the statements. And as Hugh said, in our setting we will probably have all the good assumptions to use all these results.

Theoreme 1. *Let Σ a fan. Then X_Σ is normal and separated.*

Proposition 1. *The g-vector fan associated to $\mathbb{A}_c$ is complete and then X_Σ is compact for the usual topology.*

2. Background about divisors

3. Cartier divisors of toric varieties

4. Toric variety and divisor associated to a polytope : X_P, D_P

Let P a lattice polytope in $M_{\mathbb{R}}$ and X_P the toric variety associated to P (this is also possible to do it, to get a toric variety from a polytope). An explicit description of P is given by :

$$P = \{m \in M_{\mathbb{R}} \mid \langle m, u_F\rangle \geq -a_F \ for\ all\ facets\ F\ of\ P\}$$

where $a_F \in \mathbb{Z}$ and $u_F \in N$ is the minimal generator of the ray $\rho_F \in \Sigma_P$ associated to the facet F, with Σ_P the fan associated to X_P such that $X_{\Sigma_P} = X_P$. The rays of this fan are generated by the u_F's. Besides the vertices of P correspond

(via the classical bijection Theorem 3.2.6) to the maximal cones in $\Sigma_P(n)$ and the facets of P to the rays in $\Sigma_P(1)$.

Let D_F the prime divisor associated to the facet (or the ray cone(u_F)).We look at now the following divisor :

$$D_P = \sum_{F \ facet} a_F D_F$$

This divisor is T-invariant because we could rewrite it as $D_P = \sum_{\rho \in \Sigma_P(1)} a_\rho D_\rho$ where $a_\rho := a_F$ and then we use the exercise 4.1.1. It turns out that D_P is a Cartier divisor (proposition 4.2.10) and has for local data :

$$\{(m_{\sigma_v}, \chi^{-m_{\sigma_v}})\}_{m_{\sigma_v} \in \Sigma_P(n)}$$

where v is a vertice of P and m_{σ_v} is such that the point (d) of theoreme 4.2.8.

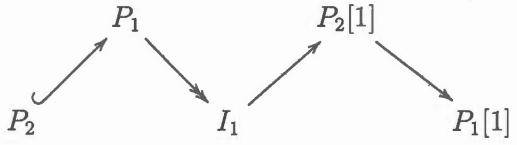
The collection $\{m_{\sigma_v}\}_{\sigma_v \in \Sigma_P(n)}$ is uniquely determined because the cones σ_v are maximal. Finally we have $v = m_{\sigma_v}$ and thus the local data of the Cartier divisor D_P is nothing but the vertices of P in M.

If we look at now the class $[D_P] \in \mathrm{Pic}(X_P) = \mathrm{CDiv}(X_P)/\mathrm{Div}_0(X_P)$ then we get all the translates of P, that is each divisor in this class is the divisor (with this construction above) of a translate of P (exercice 4.2.5). Then P depends only of the Picard group.

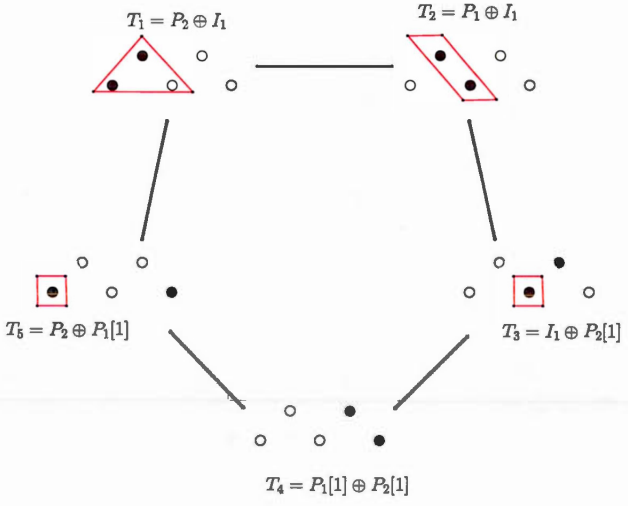
5. Polytope associated to a divisor : P_D

6. An example : $\mathbb{A}_2$

Let's consider $\mathbb{A}_2$ with the orientation $1 \to 2$. The Auslander-Reiten quiver of this graph is :



Let c_{11}, c_{21} and $c_{22} \in \mathbb{R}^+$ and $M_c = P_2^{c_{11}} \oplus P_1^{c_{21}} \oplus I_1^{c_{22}}$. here is the exchange graph of the cluster algebra of type $\mathbb{A}_2$ (the red part is the torsion class of T_i) :



By the Auslander-Reiten quiver plus the $c'_{ij}s$ we get a system of equations. By lemma 4 we have $\pi(q_{T_i}) = Dim(t_{T_i}M_c)$.

T_1 : $t_{T_1}M_c = M_c$ and thus $Dim(t_{T_1}M_c) = c_{11}dim(P_2)+c_{21}dim(P_1)+c_{22}dim(I_1) = c_{11}(0,1) + c_{21}(1,1) + c_{22}(1,0) = (c_{21} + c_{22}, c_{11} + c_{21})$

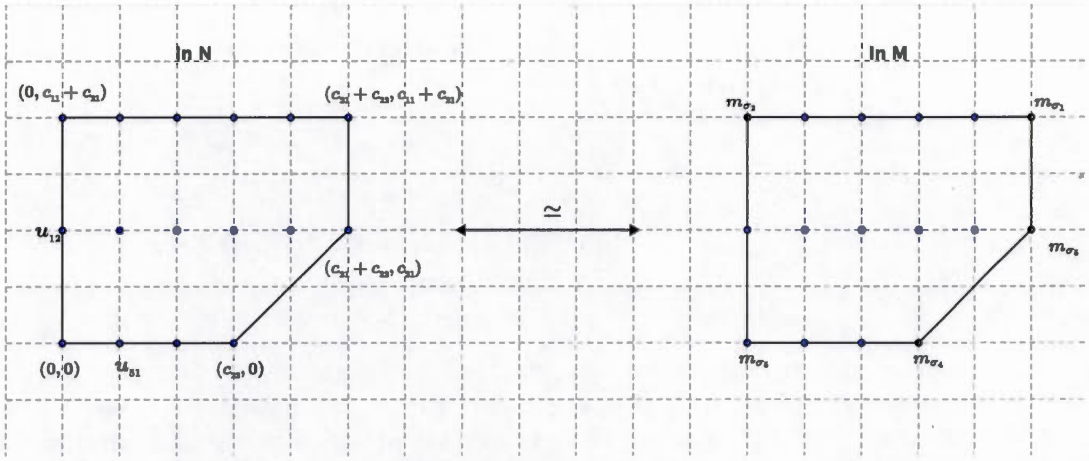
T_2 : $t_{T_2}M_c = P_1^{c_{21}} \oplus I_1^{c_{22}}$, then $Dim(t_{T_2}M_c) = (c_{21} + c_{22}, c_{21})$

T_3 : $t_{T_3}M_c = I_1^{c_{22}}$, then $Dim(t_{T_3}M_c) = (c_{22}, 0)$

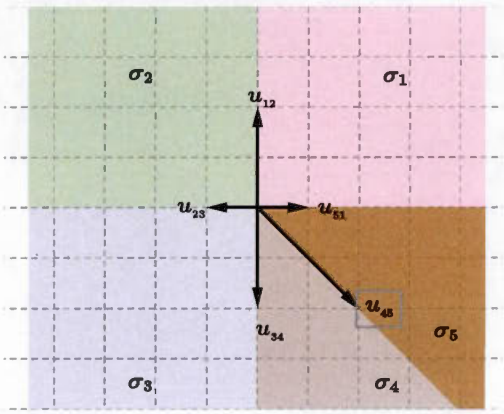
T_4 : $t_{T_4} M_c = \emptyset$, then $Dim(t_{T_4} M_c) = (0, 0)$

T_5 : $t_{T_5} M_c = P_2^{c_{11}+c_{21}}$, then $Dim(t_{T_5} M_c) = (0, c_{11} + c_{21})$

Then we can draw the associahedron seen in $N = \mathbb{Z}\langle u_{12}, u_{51} \rangle$ and $M = N^\vee$.
The associahedron belongs in M, not in N.



And the fan giving our toric variety is :
Clarifying that this is in N.



The u_{ij}'s give us the normal fan of $\mathbb{A}_c$ which is also the g-vector fan. Besides we do see here that each ray is generated by a u_{ij}. We already know that $D_{\mathbb{A}_c} = \{m_{\sigma_1}, ..., m_{\sigma_5}\}$ is a Cartier divisor because of the section 4 but we could

check it here with the compatibility conditions. Indeed for example we must have $\frac{\chi^{-m_{\sigma_1}}}{\chi^{-m_{\sigma_2}}} \in \Gamma(U_{\sigma_1} \cap U_{\sigma_2})^* = \{\chi^m\}_{m \in (\sigma_1 \cap \sigma_2)^\perp \cap N} = \{\chi^m\}_{m \in \mathbb{R}(1,0) \cap N}$. Thus we have to have $m_{\sigma_1} - m_{\sigma_2} = \lambda(1,0)$ with $\lambda \in \mathbb{R}$ such that this point belongs to N. And this is the case cause $m_{\sigma_2} - m_{\sigma_1} = -(c_{21} + c_{22}, 0)$ and the vertices of $\mathbb{A}_c$ are points of the lattice N.

Application of theorem 6.2.8

1) Let $\tau_{12} = \sigma_1 \cap \sigma_2$ and $C_{12} = V(\tau_{12})$. We have $\tau_{12} \cap N =< u_{12} >$ and then $N(\tau_{12}) = N/(\tau_{12} \cap N)$ is a one-dimensional lattice. We pick up the minimal generator of this new lattice such that it maps with an element in $\sigma_1 \cap N$. Thus we have $N/(\tau_{12} \cap N) =< \overline{u_{51}} >$. Finally :

$$\begin{aligned} D.C_{12} &= \langle m_{\sigma_2} - m_{\sigma_1}, u_{51} \rangle \\ &= \langle 2u_{12} - (2u_{12} + 5u_{51}), u_{51} \rangle \\ &= \langle -5u_{51}, u_{51} \rangle \\ &= -5 \end{aligned}$$

2) Let $\tau_{23} = \sigma_2 \cap \sigma_3$ and $C_{23} = V(\tau_{23})$. We have $\tau_{23} \cap N =< u_{23} >$ and then $N(\tau_{23}) = N/(\tau_{23} \cap N)$ is a one-dimensional lattice. We pick up the minimal generator of this new lattice such that it maps with an element in $\sigma_2 \cap N$. Thus we have $N/(\tau_{23} \cap N) =< \overline{u_{12}} >$. Finally :

$$\begin{aligned} D.C_{23} &= \langle m_{\sigma_3} - m_{\sigma_2}, u_{12} \rangle \\ &= \langle -2u_{12}, u_{12} \rangle \\ &= -2 \end{aligned}$$

3) Let $\tau_{34} = \sigma_3 \cap \sigma_4$ and $C_{34} = V(\tau_{34})$. We have $\tau_{34} \cap N =< u_{34} >$ and then $N(\tau_{34}) = N/(\tau_{34} \cap N)$ is a one-dimensional lattice. We pick up the minimal generator of this new lattice such that it maps with an element in $\sigma_3 \cap N$. Thus we have $N/(\tau_{34} \cap N) =< \overline{u_{23}} >$. Finally :

$$\begin{aligned} D.C_{34} &= \langle m_{\sigma_4} - m_{\sigma_3}, u_{23} \rangle \\ &= \langle 3u_{51}, -u_{51} \rangle \\ &= -3 \end{aligned}$$

4) Let $\tau_{45} = \sigma_4 \cap \sigma_5$ and $C_{45} = V(\tau_{45})$. We have $\tau_{45} \cap N =< u_{45} >$ and then $N(\tau_{45}) = N/(\tau_{45} \cap N)$ is a one-dimensional lattice. We pick up the minimal generator of this new lattice such that it maps with an element in $\sigma_4 \cap N$.

Thus we have $N/(\tau_{45} \cap N) =< \overline{u_{34}} >$. Finally :

$$\begin{aligned} D.C_{45} &= \langle m_{\sigma_5} - m_{\sigma_4}, u_{34} \rangle \\ &= \langle 5u_{51} + u_{12} - 3u_{51}, -u_{12} \rangle \\ &= \langle 2u_{51} + u_{12}, -u_{12} \rangle \\ &= -2\langle u_{51}, u_{12} \rangle - \langle u_{12}, u_{12} \rangle \\ &= -1 \end{aligned}$$

5) Let $\tau_{51} = \sigma_5 \cap \sigma_1$ and $C_{51} = V(\tau_{51})$. We have $\tau_{51} \cap N =< u_{51} >$ and then $N(\tau_{51}) = N/(\tau_{51} \cap N)$ is a one-dimensional lattice. We pick up the minimal generator of this new lattice such that it maps with an element in $\sigma_5 \cap N$. Thus we have $N/(\tau_{51} \cap N) =< \overline{u_{45}} >$. Finally :

$$\begin{aligned} D.C_{51} &= \langle m_{\sigma_1} - m_{\sigma_5}, u_{45} \rangle \\ &= \langle 2u_{12} + 5u_{51} - (5u_{51} + u_{12}), 2u_{51} - u_{12} \rangle \\ &= \langle u_{12}, 2u_{51} - u_{12} \rangle \\ &= -1 \end{aligned}$$

Remark 2. For us it's enough to check this intersection only for the curves associated to the walls between the maximal cones. We could say that D is nef if all these numbers were positive. So it must have a sign issue somewhere.

7. Sketch of possible text for inclusion

A fan is a collection of cones such that ...

Write Σ for the g-vector fan. Associated to a fan there is a corresponding toric variety; we denote the toric variety corresponding to Σ by X_Σ.

A Cartier divisor D on X_Σ is given by a collection of lattice points m_σ for $\sigma \in \Sigma^{(n)}$, and satisfying a compatibility condition...

Define P_D.

We can get a Cartier divisor D, which is nef, from a polytope P_D as follows :

— The outer facet normals of the polytope corresponds to the rays of σ.

— Lattice points are given by the vertices of polytopes.

We claim that these two conditions arise from our construction of $\mathbb{A}_c$. Given a polytope we can tell the local data m_σ.

We also claim that for any choice of $c \in \mathbb{Z}$ we have a Cartier divisor D. But, when we assume $c \in \mathbb{Z}_{\geq 0}$ then D is nef.

We recall the following results from [1].

Theoreme 2. [1, Theorem 6.1.10 and Theorem 6.2.12] *Assume Σ is complete. TFAE :*

(a) D is basepoint free.

(b) $\{m_\sigma\}$ is the set of vertices of P_D

(c) D is nef.

Proposition 2. *The divisor associated to the polytope $\mathbb{A}_c$ is Cartier.*

Démonstration. From the representation theory we can say that the g-vector fan Σ associated to $\mathbb{A}_c$ is unimodular, therefore the toric variety X_Σ is smooth. Thus each Weil divisor on X_Σ is actually a Cartier divisor and particularly $D_{\mathbb{A}_c}$. □

****

A Cartier divisor corresponds to a collection $m_\sigma \in M$ for $\sigma \in \Sigma(n)$, satisfying the compatibility condition that for any ray $\rho \in \Sigma(1)$, and $\sigma \in \Sigma(n)$, with σ containing ρ, $\langle m_\sigma, u_\rho \rangle$ doesn't depend on the choice of σ. We define $a_\rho = -\langle m_\sigma, u_\rho \rangle$. The Cartier divisor can also be written as $\sum_{\rho \in \Sigma(1)} a_\rho D_\rho$.

Given any collection $c_\alpha \in \mathbb{Z}^N$, we can define $\mathbb{E}_c$, and m_σ can be defined by taking the point in $\mathbb{E}_c$ where the coordinate hyperplanes corresponding to the rays that contain σ meet and projecting to the last n coordinates. Note that when we find the point where the coordinate hyperplanes meet, we are solving a system of $N+n$ equations in $N+n$ variables.

There is a lattice isomorphism from $\mathbb{E}_c$ to $\mathbb{Z}^n$ by projecting onto the final coordinates. (Clearly this is a surjective linear map since we can take any $(x_1, \ldots, x_n)$ in $\mathbb{Z}^n$ can a be extended to a point in $\mathbb{E}_c$.)

m_σ is defined by intersecting n hyperplanes in $\mathbb{E}_c$. We can think of those hyperplanes (intersected with $\mathbb{E}_c$) in $\mathbb{Z}^n$.

Those hyperplanes are normal to the g-vectors. (One way to say this really concretely is that the equation defining each hyperplane is $g_i \cdot x = a_i$. ie $G \cdot (x_1, \ldots, x_n) = (a_1, \ldots, a_n)$. But G has determinant ± 1. Using the mesh relations

we can express any coordinate y_i as some linear combination of the final coordinates plus an integer constant. That constant is exactly a_i so a_i is an integer.

We claim that this is a bijection from $\mathbb{Z}^N$ to Pic (i.e., Cartier divisors quotiented out by the action of $\mathbb{Z}^n$ acting by addition on all the m_σ).

In order to have a map from $\mathbb{Z}^N$ to Pic, we need to know that the m_σ are in $\mathbb{Z}^n$, and we need to know that they satisfy the compatibility relations, namely that for all σ containing a ray ρ, $\langle m_\sigma, u_\rho \rangle$ doesn't depend on σ. But what this is saying is that there is a hyperplane perpendicular to u_ρ containing all the m_σ, and there is : this is just the projection of the intersection of the coordinate hyperplane with $\mathbb{E}_c$.

Therefore the m_σ coming from a point c_α in $\mathbb{Z}^N$ do define a class in Pic.

Given a class in Pic, we can write its collection of m_σs, and we can subtract the constant collection $m_\sigma = c$ where c is the point $m_{\sigma_{\text{final}}}$ where we write σ_{final} for the σ corresponding to the cluster $\mathcal{I} \setminus \mathcal{I}^+$.

We now take this divisor and we can write it as $\sum a_\rho D_\rho$ with $a_\rho \in \mathbb{Z}$.

We express the hyperplane on which the m_σ lie for σ containing ρ as $\langle m_\sigma, u_\rho \rangle = a_\rho$. Consider the equation $y_\rho = \langle (y_1, \ldots, y_n), u_\rho \rangle - a_\rho$. These equations cut out a locus which we would like to be the locus $\mathbb{E}_c$. If we can find c so that $\mathbb{E}_c$ matches these equations, I claim we are done.

We can do that because we know the deformed mesh relations give us an equation of the form $y_\rho = \langle (y_1, \ldots, y_n), u_\rho \rangle - d$ for d some integer. But we can make that integer whatever we like by changing the value in the mesh to the right of y_ρ (working from right to left).

We then claim that the points in the positive orthant of $\mathbb{Z}^N$ actually correspond to nef divisors.

Theorem 6.2.8 : D is nef if for each $\tau \in \Sigma(n-1)$, we have $D \cdot V(\tau) \geq 0$. The same theorem expresses the condition in terms of local equations. Say $\tau = \sigma_1 \cap \sigma_2$, with σ_1 and σ_2 being n-dimensional cones in Σ. Then $D \cdot V(\tau) = \langle m_{\sigma_1} - m_{\sigma_2}, u \rangle$, where the local equations for D are given by m_σ. We write $N(\tau)$ for the sublattice of N generated by τ. We define u to be an element in σ_2 which maps to a generator of $N/N(\tau)$ (which is isomorphic to $\mathbb{Z}$).

From the construction in our paper, the m_σ are the vertices of the polytope defined by the a_ρ, and therefore the corresponding divisor is nef.

Suppose some $c_{ij} < 0$. I expect that the intersection of the corresponding divisor with $V(\tau)$ is negative, where $V(\tau)$ is any cone corresponding to the exchange between the two cluster variables on either end of the mesh with c_{ij} in the middle. Note in particular that if we have clusters σ_1 and σ_2 which differ only in that one has one end of the mesh and the other has the other end, then both clusters contain the middle of the mesh as well. The coordinates of m_{σ_1} with respect to the coordinates appearing in the mesh are zero and one negative coordinate at one end of the mesh, and same for m_{σ_2}, but the negative coordinate is that the other end of the mesh. I then claim this is enough information to conclude that the intersection with $V(\tau)$ is negative.

This identifies the nef cone with the positive orthant in $\mathbb{Z}^N$.

Références

[1] DD. Cox and J. Little and H. Schenck. *Toric varieties.*