UNIVERSITÉ DU QUÉBEC À MONTRÉAL

RAISONNEMENT SPATIAL POUR LA PRISE DE DÉCISION STRATÉGIQUE DANS UN JEU DE STRATÉGIE EN TEMPS RÉEL

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

JOHAN KAYSER

FÉVRIER 2017

REMERCIEMENTS

Dans un premier temps, je tiens à remercier mon directeur de recherche, le professeur Éric Beaudry, avec qui tout s'est bien déroulé et qui a toujours eu réponse à mes questions et considéré mes besoins tout au long de la maîtrise.

J'aimerais aussi remercier, pour leur aide et leurs conseils, Jean Massardi et Mathieu Gravel, mes collègues de laboratoire avec qui j'ai eu plaisir à travailler, ainsi que les autres personnes que j'ai côtoyées au laboratoire et lors des cours pendant ces deux années.

Je suis reconnaissant envers l'EXIA et l'UQAM de m'avoir offert l'opportunité de venir à Montréal au sein d'un programme de double cursus, pour réaliser cette maîtrise.

Je souhaite également remercier la Faculté des sciences de l'UQAM et le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) pour leur soutien financier, qui m'a permis de me consacrer pleinement à mes études.

Enfin, j'ai aussi une pensée pour mes proches, qui ont gardé le contact et m'ont soutenu durant cette période et l'ensemble de mes études.

Merci à eux.

TABLE DES MATIÈRES

LISTE DES TABLEAUX vii

LISTE DES FIGURES ix

RÉSUMÉ xi

INTRODUCTION 1

CHAPITRE I
L'INTELLIGENCE ARTIFICIELLE APPLIQUÉE AUX JEUX 5

1.1 Les jeux 6

1.2 Algorithmes de prise de décision 8

1.2.1 Algorithme minimax 8

1.2.2 Algorithme d'exploration minimax avec élagage alpha-bêta . . 11

1.2.3 Méthodes de Monte-Carlo 16

1.3 État de l'art des programmes de jeu 21

1.4 IA pour les jeux dans la recherche académique et IA dans les jeux vidéo 24

1.4.1 Situation actuelle 24

1.4.2 Vers une convergence ? 25

CHAPITRE II
JEUX DE STRATÉGIE EN TEMPS RÉEL : UN DÉFI POUR L'INTELLIGENCE ARTIFICIELLE 27

2.1 Le jeu StarCraft : Brood War 28

2.1.1 Description du jeu 28

2.1.2 Aperçu des stratégies 33

2.1.3 Challenges pour l'intelligence artificielle 37

2.2 État de l'art de l'IA appliquée à StarCraft 39

2.2.1 Approches au niveau stratégie 40

2.2.2 Approches au niveau tactique 43

2.2.3 Approches au niveau de la réaction 45
2.3 Communauté d'IA pour StarCraft 47
2.3.1 Interface de programmation 48
2.3.2 Analyse de la carte 50
2.3.3 UAlbertaBot : étude d'un bot 55
CHAPITRE III
EMPLOI DU RAISONNEMENT SPATIAL DANS LE JEU STARCRAFT 61
3.1 État de l'art du raisonnement spatial appliqué à StarCraft 62
3.1.1 Approche tactique : le micromanagement 63
3.1.2 Approche stratégique : le macromanagement 66
3.1.3 Limites et constat 68
3.2 Motivation et objectifs du mémoire 70
CHAPITRE IV
INTÉGRATION DU RAISONNEMENT SPATIAL AU CŒUR DE LA STRATÉGIE ET DE LA PRISE DE DÉCISIONS 73
4.1 SRbotOne : Design et stratégie 73
4.1.1 Stratégie d'ouverture 76
4.1.2 Choix de la stratégie principale 81
4.1.3 Choix de la région à sécuriser 86
4.1.4 Adaptations stratégiques et déroulement 91
4.2 SRbotOne : Détails de développement 98
4.2.1 Système de gestion des bases 98
4.2.2 Système de gestion des fronts 99
4.2.3 Comparaison avec UAlbertaBot 101
CHAPITRE V
EXPÉRIMENTATIONS ET ANALYSE DES RÉSULTATS 103
5.1 Système de fichiers de journaux 104
5.2 SRbotOne *vs* l'IA originale de StarCraft 105

5.2.1 Résultats 106
5.3 SRbotOne *vs* d'autres bots 108
5.3.1 Règlement du tournoi 110
5.3.2 Résultats 111
5.4 Compétition CIG 2016 116
5.4.1 Résultats 117
5.5 Analyse globale des résultats 121
5.5.1 Principales lacunes et pistes d'évolution 124
5.5.2 Comparaison avec UAlbertaBot 126
CONCLUSION 131
RÉFÉRENCES 135

LISTE DES TABLEAUX

Tableau Page

4.1 Informations de production des unités de début de partie. 77

4.2 Estimation de la réserve de ressources en début de partie et déduction des dates de construction des bâtiments. 79

5.1 Caractéristiques de la machine de test principale. 106

5.2 Résultats de l'opposition à l'IA de StarCraft. 107

5.3 Résultats de l'opposition à d'autres bots (tournoi) 112

5.4 Résultats de l'opposition à d'autres bots (fichiers de journaux). . . 114

5.5 Résultats du tour qualificatif de la compétition CIG 2016 118

5.6 Résultats du tour qualificatif de la compétition CIG 2016 (fichiers de journaux). 119

5.7 Statistiques des cartes de l'expérimentation 1 (1/2) 127

5.8 Statistiques des cartes de l'expérimentation 1 (2/2) 128

5.9 Statistiques des cartes de l'expérimentation 2 128

5.10 Statistiques des cartes de l'expérimentation 3 129

LISTE DES FIGURES

Figure		Page
1.1	Arbre de jeu pour l'algorithme minimax	9
1.2	Pseudo-code pour l'algorithme minimax	10
1.3	Étapes pour l'algorithme d'exploration alpha-bêta	12
1.4	Pseudo-code pour l'algorithme d'exploration alpha-bêta	14
1.5	Les 4 étapes d'un algorithme MCTS	18
1.6	Pseudo-code d'un algorithme MCTS	20
2.1	Visuel de StarCraft	31
2.2	Carte Byzantium de StarCraft	51
2.3	Étapes de BWTA (1/2)	52
2.4	Étapes de BWTA (2/2)	53
2.5	Visuel des informations de BWTA	56
2.6	Hiérarchie des classes de UAlbertaBot	57
2.7	Flux logique de UAlbertaBot	60
3.1	La perception d'un bot de StarCraft	62
3.2	Micro : encerclement de l'ennemi	64
3.3	Micro : comportement en essaim	64
3.4	Micro : A* et champ de potentiel pour la recherche de chemin	65
3.5	Micro : positionnement et mouvement des unités avec des cartes d'influence	66
3.6	Exemple de *walling*	68

3.7 Niveaux d'abstraction classiques de l'IA d'un bot de StarCraft . . 71

4.1 SRbotOne : Proposition de stratégie 1 82

4.2 SRbotOne : Proposition de stratégie 2 84

4.3 SRbotOne : Proposition de stratégie 3 86

4.4 Processus de choix de la région à sécuriser 90

4.5 Procédure pour déterminer le moment propice à l'attaque 95

4.6 Flux logique de SRbotOne 97

4.7 Hiérarchie des classes de SRbotOne 102

RÉSUMÉ

La recherche en intelligence artificielle dans les jeux de stratégie en temps réel (RTS) fait face à plusieurs challenges. Parmi eux figure le raisonnement spatial, qui est lié à chaque aspect de l'exploitation du terrain. Les travaux de recherche récents en raisonnement spatial se focalisent principalement sur les aspects de bas niveau (réaction, tactique), et ce, sans considérer le niveau plus élevé (stratégie), dans lequel le raisonnement spatial pourrait améliorer la qualité des décisions. On propose dans ce mémoire une approche destinée à donner une plus grande importance au raisonnement spatial sur le plan stratégique dans les RTS. On souhaite construire une stratégie pour chaque partie en fonction de l'analyse du terrain effectuée sur la carte de jeu, dans le but d'augmenter la capacité d'adaptation et la polyvalence d'un bot par rapport à son environnement. Pour évaluer cette approche, on a développé une intelligence artificielle (bot) qui joue au jeu vidéo StarCraft. Un système de décision, basé sur des techniques d'intelligence artificielle, permet de générer une stratégie adaptée aux conditions de l'environnement. Le système calcule le chemin le plus court entre la principale base de chacun des deux joueurs en début de partie. Sur le chemin, un système à base de règles choisit et fortifie la région la plus prometteuse pour la réussite de l'application de la stratégie. De plus, le bot décide par lui-même, lorsque le moment propice se présente, de contre-attaquer ou de lancer une attaque sur son adversaire. D'autre part, on se distingue par la mise en place d'un écosystème de gestion des bases permettant de créer différents types de bases qui passent elles-mêmes leurs commandes selon leurs besoins. L'approche proposée a été validée à l'aide de trois expérimentations, qui démontrent son efficacité. En effet, les décisions prises sont pertinentes et montrent une forte capacité d'adaptabilité en fonction du corpus d'expérimentation. Le bot a participé à une compétition officielle organisée dans le cadre de la conférence IEEE *Computational Intelligence and Games* (CIG) 2016. Il s'est classé $12^{ème}$ sur 16 participants et les données récoltées en contexte réel confirment le potentiel de l'approche présentée.

MOTS-CLÉS : intelligence artificielle, jeux vidéo de stratégie en temps réel, raisonnement spatial, décisions stratégiques, StarCraft.

INTRODUCTION

Les jeux vidéo sont utilisés comme activité de loisir et de divertissement depuis quelques dizaines d'années. Depuis *Pong*, un jeu édité par la société Atari en 1972 qui a connu le succès auprès du grand public, le secteur des jeux vidéo a connu une croissance fulgurante, au point de s'imposer comme une industrie à part entière. Aujourd'hui, les jeux vidéo sont intégrés à la culture populaire et font partie du quotidien d'une large partie de la population.

À partir du début des années 2000, l'intérêt pour la recherche en intelligence artificielle (IA) dans les jeux vidéo a fortement augmenté. En effet, les jeux vidéo sont de plus en plus utilisés comme bancs d'essai pour la recherche en IA (Laird et Lent, 2000). Ils sont envisagés comme un champ de recherche potentiel, aboutissant finalement à la quête d'une IA toujours plus réaliste (Weber *et al.*, 2011b). En 2004, Michael Buro, professeur à l'Université d'Alberta (Canada), a lancé un appel pour motiver la recherche dans les jeux de stratégie en temps réel (RTS) (Buro, 2004). Ce type de jeu fournit un environnement propice pour explorer divers défis complexes qui sont au cœur de l'IA, dans les jeux vidéo et dans beaucoup d'autres contextes, comme le contexte militaire.

Dans ce mémoire, on s'intéresse aux jeux de stratégie militaire, qui sont un cas particulier de RTS. À titre d'exemple, StarCraft figure parmi les jeux de stratégie militaire les plus connus et fréquemment utilisés et cités en recherche. Dans les jeux de stratégie militaire, les joueurs doivent : (1) développer une économie en récoltant des ressources naturelles et en construisant une base et (2), déployer

une force militaire en formant des unités de combat et en investissant dans des recherches technologiques. Le but du jeu est de détruire l'armée et la base de l'adversaire ou des adversaires (le jeu peut comporter de 2 à 4 joueurs).

D'un point de vue théorique, il y a plusieurs différences entre les RTS et les jeux plus classiques, comme les échecs (Ontañón *et al.*, 2013). Contrairement aux jeux de plateau, comme les dames et les échecs, dans les RTS, de nombreuses actions peuvent être exécutées en tout temps durant le jeu. Cela fait exploser le nombre de possibilités, rendant difficile la conception d'algorithmes efficaces pour ces jeux. Pour ces raisons, les techniques d'IA ne peuvent être appliquées directement aux RTS sans définir un niveau d'abstraction ou faire d'autres simplifications. En fait, la prise de décision et la planification doivent être faites sur plusieurs niveaux de hiérarchie.

La complexité de ce genre de jeux est très élevée en raison d'un facteur de branchement (nombre d'actions exécutables à un instant donné) important, ce qui a pour effet de faire augmenter la taille de l'espace d'états. Par exemple, l'espace d'états moyen d'un RTS est estimé à plusieurs ordres de grandeur de plus que pour les échecs et le jeu de go (Ontañón *et al.*, 2013). Pour faire face à cette complexité, on distingue principalement deux grandes familles d'approches. Il y a d'un côté les approches génériques, qui cherchent à faire de l'approximation (Browne *et al.*, 2012), et de l'autre les approches spécifiques, qui sont beaucoup plus scriptées et dépendantes de chaque jeu.

Les chercheurs ont dégagé plusieurs challenges pour ce domaine de recherche en IA (Buro, 2003; Buro, 2004). Parmi eux figure le raisonnement spatial (Ontañón *et al.*, 2013; Robertson et Watson, 2014). Le raisonnement spatial est lié à chaque aspect de l'exploitation du terrain, comme le choix stratégique de l'emplacement des bâtiments, le placement stratégique des unités pour les batailles, la prédiction

de la position d'un ennemi basée sur peu d'informations, etc. Des travaux ont été effectués pour analyser les cartes (Perkins, 2010), tenter de prédire la position des ennemis (Weber *et al.*, 2011b), inculquer à l'IA des techniques utilisées par les joueurs de haut niveau (Certický, 2013), etc. La majeure partie de la recherche en raisonnement spatial dans les RTS cherche à agréger et extraire de l'information spatiale en additionnant des données individuelles pour les regrouper en champs sur une zone (Hagelbäck et Johansson, 2008; Synnaeve et Bessière, 2011c; Uriarte et Ontañón, 2012).

Cependant, les techniques précitées ont des lacunes. Elles permettent rarement d'anticiper les agissements de l'adversaire et souffrent de problèmes d'optimums locaux (Ontañón *et al.*, 2013). En plus de demander beaucoup de ressources (Hagelbäck et Johansson, 2008), ces méthodes de reconnaissance de terrain requièrent souvent de nombreux réglages avant d'être efficaces (Robertson et Watson, 2014) et sont difficiles à paramétrer (Hagelbäck, 2009). D'autre part, la recherche dans les RTS se focalise principalement sur les aspects de bas niveau, c'est-à-dire au niveau tactique (raisonnement à court terme) et au niveau de la réaction (décision immédiate sans raisonnement). Le niveau le plus élevé, c'est-à-dire le niveau de la stratégie, est plus rarement étudié. L'hypothèse portée par les travaux présentés dans ce mémoire est que le raisonnement spatial peut lui aussi améliorer la prise de décision au niveau stratégique.

L'objectif principal de ce mémoire est de présenter une approche destinée à donner une plus grande importance au raisonnement spatial sur le plan stratégique dans les RTS. On souhaite construire une stratégie pour chaque partie en fonction de l'analyse du terrain effectuée sur la carte de jeu, dans le but d'augmenter la capacité d'adaptation et la polyvalence d'un bot par rapport à son environnement. On cherche à démontrer qu'il est possible de bâtir une stratégie en fonction du contexte spatial, sans définir ou programmer de règles spécifiques à une carte don-

née, pour faciliter le maintien et le paramétrage d'une telle intelligence artificielle.

L'approche proposée consiste à calculer le chemin le plus court entre les deux joueurs d'une partie. Sur ce chemin, on détermine la région la plus pertinente à consolider afin de bloquer l'adversaire. Cette région devient le centre des affrontements sur la carte de jeu ; après avoir estimé le moment propice, le bot[1] décide de façon autonome de passer à l'attaque contre l'ennemi.

On a expérimenté cette approche avec le jeu StarCraft, qui fait l'objet de compétitions en IA. Les résultats démontrent la pertinence de la méthode. En effet, les décisions prises sont efficaces et démontrent une forte capacité d'adaptabilité en fonction du corpus d'expérimentation.

Ce mémoire adopte une structure en cinq chapitres. Le premier présente de façon générale l'application de l'intelligence artificielle aux jeux. On y débute l'état de l'art en faisant un survol des connaissances importantes du domaine. Le second chapitre focalise l'attention sur les RTS, qui représentent un défi pour l'IA, et recentre l'état de l'art au regard du statut de la recherche dans ce domaine précis. Le troisième chapitre clôt l'état de l'art du mémoire en s'intéressant plus précisément à la recherche en raisonnement spatial dans les RTS. Dans le chapitre quatre, on aborde la contribution du mémoire à la recherche, avec la mise en place du raisonnement spatial au cœur de la stratégie et de la prise de décision d'une intelligence artificielle de RTS. Enfin, le chapitre cinq détaille les expérimentations effectuées et leur analyse. Finalement, une conclusion clôt ce mémoire.

---

1. Bot = « Joueur virtuel contrôlé par l'ordinateur, capable de reproduire le comportement d'un joueur humain, qui peut remplacer un joueur dans un jeu multijoueur ». Source : `http://granddictionnaire.com/ficheOqlf.aspx?Id_Fiche=39606`.

CHAPITRE I

L'INTELLIGENCE ARTIFICIELLE APPLIQUÉE AUX JEUX

En intelligence artificielle, un agent intelligent est une entité qui peut percevoir son environnement à l'aide de ses capteurs et agir dans cet environnement à l'aide de ses effecteurs. Son intelligence peut être vue comme une fonction qui associe un historique de percepts (données fournies par les capteurs) à une action. Cette fonction est continuellement évaluée à une certaine fréquence et c'est elle qu'on programme en intelligence artificielle afin de résoudre des problèmes. Il existe plusieurs approches pour implémenter une telle fonction. Parmi elles figure la « recherche ». Le processus de résolution peut consister à observer la situation actuelle, énumérer les options possibles, évaluer leurs conséquences et, finalement, retenir la meilleure option possible pour atteindre le but fixé. Il s'agit en fait d'explorer et de trouver la meilleure solution entre toutes. Ainsi, on parle de résolution de problème par recherche, ou de problème d'exploration (des solutions).

Dans ce chapitre, on commence par définir ce qu'est un jeu pour l'intelligence artificielle. On présente ensuite des algorithmes de prise de décision pour les jeux. La section 1.3 dresse un état de l'art des programmes de jeu, dans lequel on relate les performances de l'ordinateur face à des jeux classiques. Puis, on termine par exposer les différences au sujet de l'intelligence artificielle dans les domaines de la recherche académique et de l'industrie du jeu vidéo.

1.1 Les jeux

On s'intéresse aux situations qui apparaissent lorsqu'on essaie de faire des plans à l'avance, dans un monde dans lequel d'autres font des plans opposés aux nôtres. On se retrouve dans une situation où les buts et stratégies respectives entrent en conflit, ce qui entraîne des problèmes dits d'« exploration en situation d'adversité » (Russell et Norvig, 2009), communément appelés *jeux*. On emploie le terme *exploration* pour exprimer la recherche de solutions (à un problème). Ainsi, en intelligence artificielle, les jeux sont des problèmes de recherche de solutions en situation d'adversité.

Un jeu est caractérisé de manière formelle par les éléments suivants :

- Un **état initial**, qui décrit la situation du jeu au début ;
- Les **joueurs**, et celui qui a le droit de jouer dans l'état donné ;
- Les **actions**, c'est-à-dire l'ensemble des coups possibles dans un état donné ;
- Le **modèle de transition**, qui définit le résultat d'un coup ;
- Un **test de terminaison**, qui indique si oui ou non le jeu est fini. Les états dans lesquels le jeu s'arrête sont appelés les « états terminaux » ;
- Une **fonction d'utilité**, ou fonction « d'objectif » ou « de gain », qui donne une valeur numérique pour un jeu qui se termine dans un état terminal donné pour un joueur donné.

Russel et Norvig introduisent également dans leur livre les notions d'**arbre de jeu** et d'**arbre d'exploration**. L'arbre de jeu pour une partie est défini par l'état initial, les actions et le modèle de transition. Il représente l'ensemble des déroulements et dénouements possibles de la partie. Ses sommets sont des états du jeu et ses arêtes sont des coups. L'arbre d'exploration est un arbre partiel qui se superpose à l'arbre de jeu (qui lui est complet), mais qui contient suffisamment de nœuds pour permettre à un joueur de choisir un coup.

En intelligence artificielle, les jeux les plus communs sont d'un type particulier : on les appelle jeux à **somme nulle** et à **information parfaite**. Dans ces jeux, la somme des valeurs de la fonction d'utilité obtenue par les joueurs est toujours égale à zéro, car ces valeurs s'opposent. De plus, ces jeux sont déterministes, ce qui signifie que dans leur environnement, l'état suivant est complètement déterminé par l'état courant et par l'action choisie par le joueur. Ils sont aussi complètement observables, c'est-à-dire qu'un joueur peut tout savoir de l'état de l'environnement à un moment donné. Enfin, on dit qu'ils sont alternés, parce que les joueurs agissent à tour de rôle. Par exemple, les échecs sont considérés comme un jeu à somme nulle et à information parfaite. Effectivement, si un joueur gagne une partie d'échecs, l'autre perd obligatoirement.

Il existe d'autres types de jeux, comme les **jeux stochastiques**, qui incluent une notion d'imprévisibilité ou de hasard sous la forme d'un élément aléatoire, tel un lancer de dés. Une incidence se fait alors remarquer dans l'arbre de jeu, puisqu'il faut y inclure des « nœuds de hasard », ce qui augmente considérablement sa taille. Les jeux peuvent être **partiellement observables** et donc à **information imparfaite**, comme la bataille navale par exemple. Dans ce cas, la taille de l'arbre de jeu augmente beaucoup également. On doit créer des « états de croyance » pour représenter l'ensemble de tous les états logiquement possibles du jeu, étant donné l'historique complet de l'environnement et des actions (Russell et Norvig, 2009).

Les jeux ont la cote en intelligence artificielle, parce qu'ils mobilisent les facultés intellectuelles de l'homme et sont parfois trop difficiles à résoudre. Un jeu simple comme le tic-tac-toe a un arbre de jeu de moins de 9 ! (362 880) nœuds terminaux [1]. De plus, dans les problèmes d'exploration classiques, la recherche d'une solution

1. NB : L'arbre de jeu contient des nœuds représentant des états équivalents et inaccessibles. En réalité, le nombre d'états uniques est seulement 5478 (Source : `https://goo.gl/qy9LDP`).

optimale consiste à trouver une séquence d'actions qui conduit à un état terminal désiré (Russell et Norvig, 2009). Or dans une exploration en situation d'adversité, le joueur doit trouver une stratégie qui spécifie son coup à l'état initial, ses coups dans les états résultant de tous les choix possibles de son adversaire, puis ceux que provoqueront les états résultant des réponses possibles de l'adversaire à ses propres coups, et ainsi de suite jusqu'à une fin du jeu. On se rend rapidement compte que le problème peut devenir complexe à résoudre, d'autant qu'en réalité, le temps alloué pour trouver une stratégie optimale est limité. C'est pourquoi on doit parfois arrêter la recherche avant la fin et retourner une décision imparfaite. Dans la prochaine section, on présente des algorithmes qui permettent de déterminer cette stratégie optimale.

1.2 Algorithmes de prise de décision

1.2.1 Algorithme minimax

Soit deux joueurs `MAX` et `MIN`. `MAX` joue en premier, puis les deux joueurs jouent à tour de rôle jusqu'à la fin du jeu. Une stratégie optimale a des résultats au moins aussi bons que toute autre stratégie contre un adversaire optimal. À partir d'un arbre de jeu, on peut trouver la stratégie optimale en cherchant la **valeur minimax** de chaque nœud. La valeur minimax d'un nœud (MINIMAX(n)) est l'utilité pour `MAX`, associée à l'état du nœud (si les deux joueurs jouent de manière optimale depuis l'état courant jusqu'à la fin). Naturellement, `MAX` préfère les états caractérisés par une utilité maximale, alors que `MIN` préfère les états de valeur minimale. On peut calculer la valeur minimax d'un état s avec l'équation (1.1) :

$$\text{MINIMAX}(s) = \begin{cases} \textit{Utilité}(s) & \textit{si } \text{TEST-TERMINAISON}(s) \\ max_{a \in A(s)} \text{MINIMAX}(\gamma(s,a)) & \textit{si } \text{JOUEUR}(s) = \texttt{MAX} \\ min_{a \in A(s)} \text{MINIMAX}(\gamma(s,a)) & \textit{si } \text{JOUEUR}(s) = \texttt{MIN} \end{cases} \tag{1.1}$$

où $A(s)$ est l'ensemble des actions applicables dans l'état s, et $\gamma(s, a)$ est la fonction de transition retournant l'état résultat de l'application de l'action a dans l'état s.

L'**algorithme minimax** présenté à la figure 1.2 calcule la **décision minimax** pour l'état courant. On peut alors l'identifier à la racine de l'arbre de jeu. Soit l'arbre de jeu à la figure 1.1. Les triangles qui pointent vers le haut sont des « nœuds `MAX` » dans lesquels `MAX` a l'initiative, et les triangles pointant vers le bas sont des « nœuds `MIN` ». Les nœuds terminaux indiquent les valeurs d'utilité pour `MAX` et les autres sont étiquetés par leur valeur minimax. À la racine, le meilleur coup possible pour `MAX` est a_1, parce qu'il mène à l'état ayant la plus forte valeur MINIMAX, et la meilleure réponse de `MIN` est b_1, parce qu'il mène à l'état ayant la plus faible valeur MINIMAX à partir du choix de `MAX`.

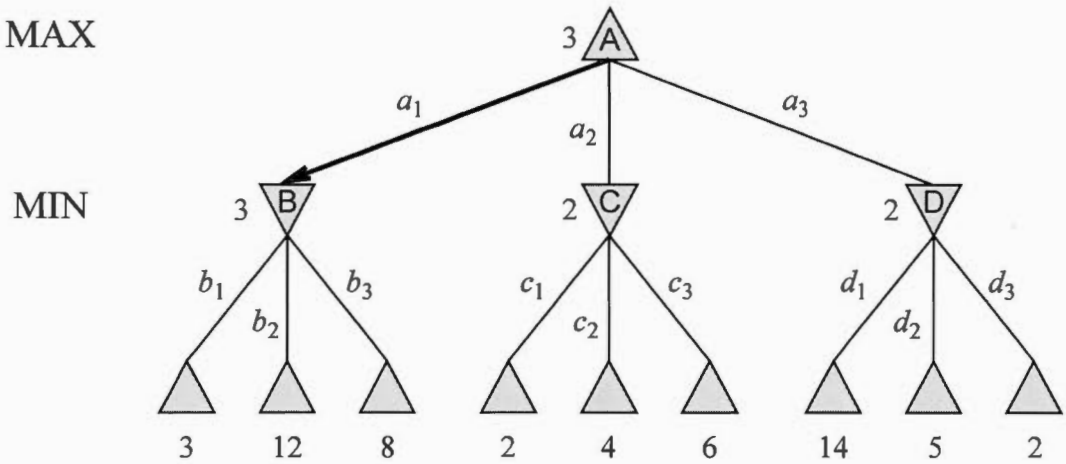

Figure 1.1 Un arbre de jeu. Source : (Russell et Norvig, 2009).

L'algorithme utilise un calcul récursif des valeurs MINIMAX de chaque état successeur par le biais d'une implémentation des équations de MINIMAX. La récursivité commence par développer toutes les feuilles de l'arbre, puis elle fait remonter les valeurs MINIMAX le long de l'arbre niveau par niveau lorsque les appels récursifs sont dépilés. On développe, dans un premier temps, la partie gauche de l'arbre

jusqu'à atteindre les nœuds terminaux, puis on utilise la fonction UTILITÉ pour trouver les valeurs 3, 12 et 8. Ensuite, on prend la plus basse de ces valeurs et on la retourne comme valeur pour le nœud parent. On obtient de la même façon les valeurs 2 pour le nœud `MIN` du milieu et 2 pour le nœud `MIN` de droite. Enfin, on sélectionne la plus grande valeur entre 3, 2 et 2 pour la remonter au nœud racine. Pour cet arbre, le coup optimal pour `MAX` est donc a_1.

fonction DÉCISION-MINIMAX(état) **retourne** une action
  retourner $\arg\max_{a \in Action(état)}$ VALEUR-MIN(RÉSULTAT(état, a))

---

fonction VALEUR-MAX(état) **retourne** une valeur d'utilité
  si TEST-TERMINAISON(état) **alors retourner** UTILITÉ(état)
  $v \leftarrow -\infty$
  pour chaque a **dans** ACTIONS(état) **faire**
    $v \leftarrow$ MAX(v, VALEUR-MIN(RÉSULTAT(état, a)))
  retourner v

---

fonction VALEUR-MIN(état) **retourne** une valeur d'utilité
  si TEST-TERMINAISON(état) **alors retourner** UTILITÉ(état)
  $v \leftarrow +\infty$
  pour chaque a **dans** ACTIONS(état) **faire**
    $v \leftarrow$ MIN(v, VALEUR-MAX(RÉSULTAT(état, a)))
  retourner v

Figure 1.2 Pseudo-code pour l'algorithme minimax. Les fonctions VALEUR-MAX et VALEUR-MIN traversent la totalité de l'arbre de jeu jusqu'à atteindre les feuilles, de manière à déterminer la valeur à remonter pour caractériser un état. Source : (Russell et Norvig, 2009).

L'algorithme minimax réalise une exploration en profondeur d'abord (DFS, pour

Depth First Search) de l'arbre de jeu. Cette définition du coup optimal pour `MAX` suppose que `MIN` joue aussi de manière optimale, c'est-à-dire qu'il maximise la pire issue possible pour `MAX`. Si `MIN` ne joue pas de façon optimale, `MAX` peut obtenir un gain encore plus élevé, mais face à des adversaires sous-optimaux, il peut y avoir des stratégies plus intéressantes que minimax.

Si la profondeur maximale de l'arbre est m et qu'il y a b coups à chaque point, alors la complexité en temps de l'algorithme minimax est $O(b^m)$ et la complexité en espace est $O(bm)$. On voit bien que pour des situations réelles, l'emploi de cet algorithme nécessite un temps qui le rend inefficace. Il est cependant à la base d'algorithmes plus pratiques, comme l'algorithme d'exploration alpha-bêta.

1.2.2 Algorithme d'exploration minimax avec élagage alpha-bêta

Le problème de l'exploration minimax est que le nombre d'états à examiner dépend exponentiellement de la profondeur de l'arbre. Il est impossible d'éliminer l'exposant, mais on peut le diviser par deux pour obtenir une complexité temporelle de $O(b^{m/2})$ dans le meilleur cas. En effet, il est parfois possible de calculer la décision minimax sans avoir à examiner tous les nœuds de l'arbre de jeu. Pour cela, il faut choisir un ordre des nœuds à visiter pour maximiser l'élagage. On appelle cette technique l'**élagage alpha-bêta**. Appliquée à un arbre, elle retourne le même coup que l'algorithme minimax, mais en élaguant des branches qui ne risquent pas d'influer sur la décision finale. L'élagage alpha-bêta s'applique à des arbres de n'importe quelle profondeur et il est souvent possible d'élaguer des sous-arbres entiers de cette façon. Le principe est de considérer un nœud n de l'arbre, de sorte que le joueur ait le choix de se déplacer jusqu'à ce nœud. Si le joueur dispose d'un meilleur choix m au niveau du nœud parent de n ou même plus haut, alors n ne sera jamais atteint dans le jeu. Quand on en sait suffisamment sur n grâce à l'examen de ses descendants, on peut l'élaguer (Russell et Norvig, 2009).

Comme l'algorithme minimax exécute une exploration en profondeur d'abord, les nœuds à considérer sont uniquement ceux situés le long d'un chemin de l'arbre. La technique tire son nom des deux bornes qui sont remontées le long du chemin. α représente la valeur la plus élevée trouvée jusqu'ici pour les choix effectués le long du chemin par `MAX` et β la valeur la plus basse trouvée jusqu'ici pour les choix effectués le long du chemin par `MIN`. L'exploration alpha-bêta met à jour les valeurs α et β au fur et à mesure de sa progression et élague les branches restantes d'un nœud (elle termine l'appel récursif). On procède à l'élagage dès que la valeur du nœud courant est assurément moins bonne que la valeur α ou β concurrente pour `MAX` ou `MIN` respectivement.

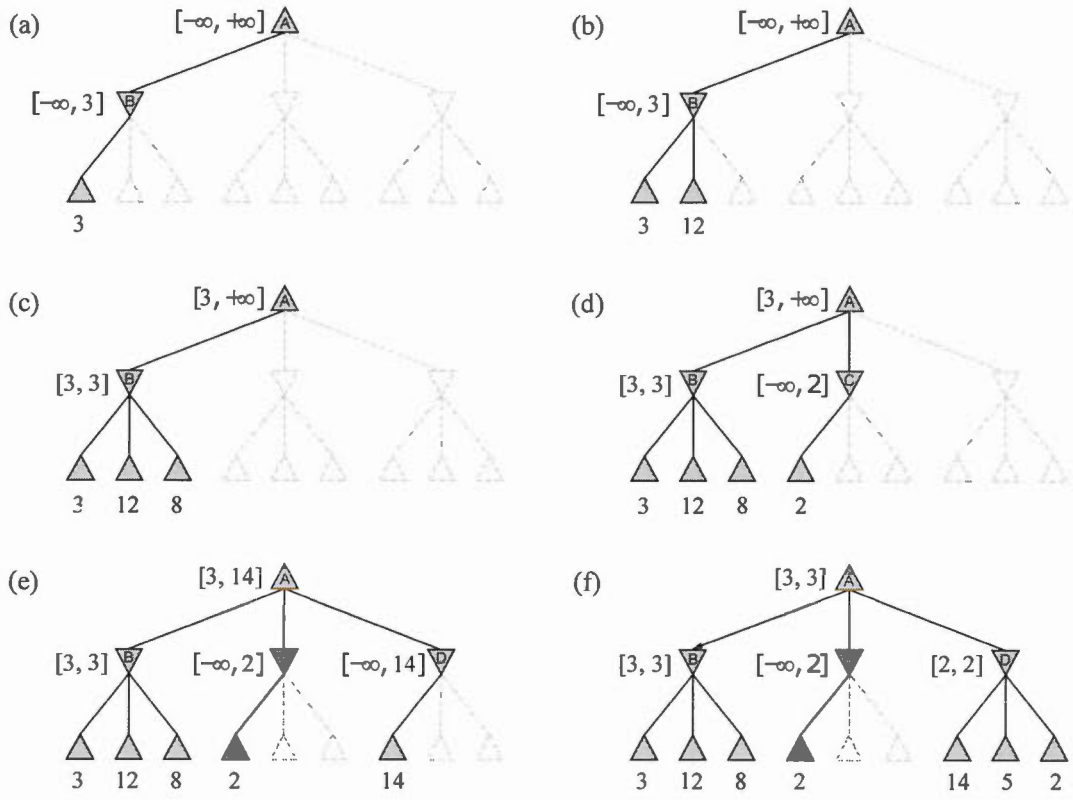

Figure 1.3 Étapes du calcul de la décision optimale avec élagage pour l'arbre de jeu de la figure 1.1. Source : (Russell et Norvig, 2009).

Reprenons la figure 1.1 pour y appliquer l'algorithme minimax avec élagage alpha-bêta. La figure 1.3 montre la trace de l'algorithme. Au point (a), la première feuille sous B a la valeur 3. Par conséquent, B, qui est un nœud `MIN`, a une valeur maximale de 3. Ensuite, au point (b), la deuxième feuille sous B a une valeur de 12. `MIN` ne choisira pas ce coup, la valeur de B reste d'au plus 3. En (c), la dernière feuille sous B a une valeur de 8. `MIN` ne choisira pas ce coup non plus. Comme tous les états successeurs de B ont été explorés, la valeur de B est exactement 3. On peut donc inférer que la valeur de la racine A est au minimum de 3, car `MAX` a un choix auquel est associée la valeur 3 à la racine. Au point (d), la première feuille sous C a la valeur 2. Par conséquent, C, qui est un nœud `MIN`, a une valeur maximale de 2. Or, on sait que B vaut 3, c'est pourquoi `MAX` ne choisira jamais C. Il est donc inutile d'explorer les autres états successeurs de C : c'est un exemple d'élagage alpha-bêta. Le point (e) développe la première feuille sous D, qui a la valeur de 14. Donc, D vaut au maximum 14.

Comme cette valeur est supérieure à la meilleure solution de `MAX` (qui est actuellement 3), on doit continuer d'explorer les états successeurs de D. Remarquons qu'on connait maintenant les limites (3, 14) pour les successeurs de la racine et que sa valeur est au maximum de 14. Enfin, en (f), le deuxième successeur de D vaut 5, qui est inférieur à 14 et devient donc la préférence, mais on doit continuer l'exploration. Le troisième successeur de D vaut 2, qui est lui aussi inférieur. `MIN` le choisira. Donc, D vaut désormais exactement 2. La décision de `MAX` est donc le coup vers B pour une valeur de 3.

En comparant le pseudo-code de l'algorithme minimax (figure 1.2) à celui de l'algorithme d'exploration alpha-bêta (figure 1.4), on remarque qu'on ajoute uniquement deux lignes à la fin de VALEUR-MAX et VALEUR-MIN pour la comparaison avec α et β respectivement, ainsi que pour leur mise à jour. On effectue aussi les changements pour les passer en arguments.

```
fonction EXPLORATION-ALPHA-BÊTA(état) retourne une action
    v ← VALEUR-MAX(état, −∞, ∞)
    retourner l'action dans ACTIONS(état) avec la valeur v
```

```
fonction VALEUR-MAX(état, α, β) retourne une valeur d'utilité
    si TEST-TERMINAISON(état) alors retourner UTILITÉ(état)
    v ← −∞
    pour chaque a dans ACTIONS(état) faire
        v ← MAX(v, VALEUR-MIN(RÉSULTAT(état, a), α, β)
        si v ≥ β alors retourner v
        α ← MAX(α, v)
    retourner v
```

```
fonction VALEUR-MIN(état, α, β) retourne une valeur d'utilité
    si TEST-TERMINAISON(état) alors retourner UTILITÉ(état)
    v ← +∞
    pour chaque a dans ACTIONS(état) faire
        v ← MIN(v, VALEUR-MAX(RÉSULTAT(état, a), α, β)
        si v ≤ α alors retourner v
        β ← MAX(β, v)
    retourner v
```

Figure 1.4 Pseudo-code pour l'algorithme d'exploration alpha-bêta. Source : (Russell et Norvig, 2009).

Si l'élagage alpha-bêta peut permettre d'éviter l'exploration de grandes parties de l'arbre de jeu, il implique quand même l'exploration d'un nombre significatif de nœuds de l'arbre jusqu'aux états terminaux. Le problème est que cette profondeur est généralement impossible à atteindre en pratique, parce que b et m sont trop grands et que les coups doivent être joués dans un laps de temps court (de l'ordre

de la minute aux échecs et beaucoup moins dans les jeux vidéo, par exemple).

Dans son article « Programming a Computer for Playing Chess » (Shannon, 1988), Claude Shannon proposait que les programmes interrompent l'exploration plus tôt et appliquent une **fonction d'évaluation** sur des états afin de transformer des nœuds non terminaux en feuilles terminales. Il suggérait de modifier les algorithmes minimax ou alpha-bêta de deux manières. Premièrement, remplacer la fonction d'utilité par une fonction d'évaluation EVAL, qui donne une estimation de l'utilité de la position. Deuxièmement, remplacer le test de terminaison par un **test d'arrêt**, qui décide quand appliquer EVAL. Limiter la profondeur de cette façon est une technique permettant de prendre des décisions en temps réel.

Comme la fonction d'évaluation retourne une estimation de l'utilité espérée du jeu pour l'état donné, on tombe obligatoirement dans des « décisions imparfaites ». La qualité de l'estimation a une incidence directe sur la qualité des décisions prises. Tout d'abord, la fonction d'évaluation doit ordonner les états terminaux de la même façon que la vraie fonction d'utilité. En effet, les états correspondant à des victoires doivent être mieux notés que ceux menant à un match nul, qui eux-mêmes doivent être mieux notés que ceux correspondant à une défaite. Ensuite, il ne faut pas que les calculs prennent trop de temps, puisque l'objectif principal est d'en gagner. Enfin, pour des états non terminaux, il faut que la fonction d'évaluation soit fortement corrélée avec les chances effectives de victoire (Russell et Norvig, 2009).

Dans le cas de l'algorithme d'exploration alpha-bêta, il faut alors modifier les lignes de la figure 1.4, qui appellent TEST-TERMINAISON par :

si TEST-ARRÊT(état, profondeur) **alors retourner** EVAL(état)

De cette façon, on peut réaliser une exploration avec arrêt à la profondeur sou-

haitée et avoir un outil exploitable dans des situations réelles. Dans la section suivante, on étudie une autre technique permettant de prendre des décisions en temps réel.

1.2.3 Méthodes de Monte-Carlo

Les méthodes de Monte-Carlo représentent un concept très général qui se réfère à l'utilisation de méthodes stochastiques (impliquant la chance et la théorie des probabilités) pour résoudre des problèmes. En d'autres termes, une méthode de Monte-Carlo est « une méthode qui résout un problème en générant des nombres aléatoires et en observant qu'une partie des nombres obéit à une ou des propriétés » [2]. On en utilise pour la résolution avec ordinateur de problèmes numériques où il n'y a pas de solution analytique, mais aussi dans d'autres contextes, comme pour la simulation de systèmes complexes (prévisions météorologiques) ou la résolution d'intégrales et de systèmes d'équations différentielles. Avec ce genre de technique, la qualité de l'estimation s'améliore avec le nombre de données/itérations. Or, la qualité de l'échantillonnage des nombres aléatoires est primordiale pour avoir de bons résultats.

Dans le contexte de l'IA et des jeux, les méthodes de Monte-Carlo sont utilisées pour la prise de décision. Un algorithme reprenant ce principe doit être composé des éléments suivants :

- Une définition du problème et des entrées possibles ;
- Un générateur de nombres aléatoires ou processus d'échantillonnage ;
- Un mécanisme pour faire les calculs avec les nombres aléatoires ;
- Un mécanisme pour assembler les résultats et en déduire quelque chose.

L'idée d'utiliser ces méthodes de Monte-Carlo pour les problèmes d'exploration est

---

2. Source : `http://mathworld.wolfram.com`

relativement récente (Abramson, 1990; Brügmann, 1993), mais ce n'est qu'encore plus tard qu'elle a vraiment été popularisée (Coulom, 2006), notamment pour le jeu de go. Le principe qui en dérive pour explorer un arbre ou un espace d'états en utilisant des méthodes de Monte-Carlo a été appelé ***Monte-Carlo Tree Search*** (MCTS). C'est une méthode pour trouver des décisions approximatives dans un domaine donné, en prenant des échantillons aléatoires dans l'espace de décision et en construisant un arbre de recherche. Ici, au lieu de limiter la profondeur, on limite le nombre de branches à visiter, en sélectionnant des branches aléatoires et en allant souvent jusqu'aux feuilles (nœuds sur lesquels il est possible de calculer l'utilité). Elle est basée sur les concepts suivants :

— La valeur d'une action est calculée approximativement et de façon aléatoire ;
— Les valeurs des actions sont utilisées pour ajuster la politique d'exploration ;
— On construit un arbre de façon partielle, en fonction des résultats des explorations précédentes ;
— Plus on fait d'itérations, plus la précision augmente.

À chaque itération d'un algorithme MCTS, on passe par quatre étapes (figure 1.5). La première est celle de la **sélection**. En partant de la racine, on explore l'arbre jusqu'à trouver un nœud non terminal hors de la frontière et ayant un enfant non visité. À partir de ce nœud, on passe à l'étape d'**expansion**, qui consiste à explorer un des successeurs, choisi de façon aléatoire. Ainsi, on ajoute un nœud à l'arbre. Depuis ce nouveau nœud, on effectue l'étape de **simulation**, dans laquelle on développe ses successeurs de façon aléatoire jusqu'à arriver à une feuille, pour en récupérer l'utilité. Enfin, la dernière étape est la **propagation-arrière** : on va remonter la valeur d'utilité obtenue jusqu'à la racine.

On dit également qu'un algorithme emploie deux politiques (Bauckhage, 2014). La première, TREEWALK, regroupe les étapes de sélection et d'expansion et ajoute un nœud à l'arbre qui est construit. La deuxième, RANDOMWALK, correspond à

l'étape de simulation et sélectionne des actions aléatoires jusqu'à pouvoir retourner une utilité. Il est à noter que les nœuds parcourus à l'étape de simulation ne sont pas ajoutés à l'arbre, contrairement au nœud choisi lors de la première politique et durant laquelle on en ajoute un dans l'arbre.

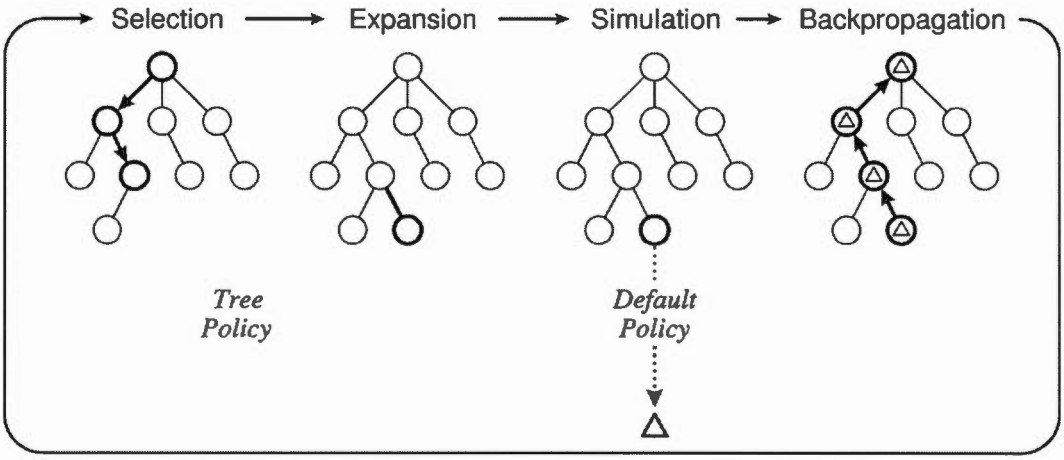

Figure 1.5 Étapes d'une itération d'un algorithme MCTS (Browne *et al.*, 2012).

À l'exécution, l'algorithme (figure 1.6) part de la racine de l'arbre et entre avec le nœud racine dans TREEWALK. Si le nœud n'est pas dans la frontière de l'arbre, on choisit un de ses successeurs de façon aléatoire et on ré-exécute TREEWALK avec lui, de façon récursive. Une fois qu'on a trouvé un nœud de la frontière, on choisit un de ses successeurs de façon aléatoire et on l'ajoute à l'arbre. Après cela, on appelle RANDOMWALK sur ce nœud. Dans RANDOMWALK, si le nœud n'est pas une feuille, on l'étend aléatoirement sans ajouter son successeur à l'arbre et on ré-exécute RANDOMWALK avec lui, de façon récursive, jusqu'à arriver à une feuille. Quand on trouve un nœud feuille, on remonte son utilité jusqu'à la racine en complétant les appels récursifs, ce qui va mettre à jour la valeur d'utilité associée à chaque nœud de l'arbre. Enfin, l'exécution s'arrête quand on le souhaite (après T itérations) grâce à la boucle **pour** de MCTS, et l'algorithme retourne le successeur direct de la racine avec la meilleure valeur d'utilité.

La complexité d'un algorithme MCTS de ce type est en $O(m)$, où m est la profondeur maximale de l'arbre (Bauckhage, 2014). Cependant, il faut garder à l'esprit qu'il est nécessaire de réaliser un grand nombre d'itérations pour obtenir de bons résultats ; ce nombre est souvent déterminé par le temps alloué à l'algorithme. Un autre avantage intéressant d'un algorithme MCTS vient du fait qu'il est possible d'obtenir des informations sur la partie en cours en observant la structure de l'arbre généré. On peut, par exemple, remarquer rapidement qu'une partie est perdue d'avance.

Pour terminer, il faut noter que lors des étapes de sélection et d'expansion d'une méthode MCTS, on ne devrait pas choisir de successeur de façon aléatoire si un nœud a déjà été étendu entièrement et n'est donc plus dans la frontière. En effet, pour des raisons d'efficacité, il est judicieux de choisir plus finement. On se trouve alors devant un problème qu'on appelle *multi-armed bandit problem* (Browne *et al.*, 2012), ou dilemme exploration/exploitation. La question qui se pose est : doit-on continuer de confirmer les statistiques sur la branche profitable, ou plutôt explorer une autre branche qui pourrait l'être encore plus ? En d'autres mots, on souhaite acquérir de nouvelles connaissances tout en maximisant la récompense basée sur les connaissances actuelles.

fonction MCTS(s_0)

Soit *arbre* un arbre initialisé avec un état d'origine s_0

Soit n_0 la racine de *arbre* représentant s_0

pour $i = 1$ à T **faire**

récompense ← TREEWALK(*arbre*, n_0)

retourner le fils de n_0 avec la meilleure récompense

---

fonction TREEWALK(*arbre*, n)

si n a été complètement exploré **et** n n'est pas une feuille **alors**

Soit *successeur* un fils de n choisi de façon aléatoire

récompense ← TREEWALK(*arbre*, *successeur*)

sinon

si n est une feuille **alors** *récompense* ← RANDOMWALK(*arbre*, n)

sinon

Soit *nonExploré* un fils non exploré de n choisi de façon aléatoire

Ajouter *nonExploré* comme fils de n à *arbre*

récompense ← RANDOMWALK(*arbre*, *nonExploré*)

récompense(n) ← *récompense*(n) + *récompense*

retourner *récompense*

---

fonction RANDOMWALK(*arbre*, n)

si n est une feuille **alors**

récompense ← UTILITÉ(n)

sinon

Soit *successeur* un fils de n choisi de façon aléatoire

récompense ← RANDOMWALK(*arbre*, *successeur*)

retourner *récompense*

Figure 1.6 Pseudo-code d'un algorithme MCTS. Source : (Browne *et al.*, 2012).

Pour résoudre ce genre de problème, on introduit les notions de **regret** et d'***Upper Confidence Bounds***. Le regret est la perte attendue en ne faisant pas le meilleur choix ; on va donc naturellement chercher à le limiter. *Upper Confidence Bounds* est une politique qui détermine quelle décision prendre pour limiter le regret. Elle indique de prendre la décision qui maximise $UCB1$ (équation (1.2)).

Cette politique est implémentée dans l'algorithme ***Upper Confidence Bounds for Trees*** (UCT), qui est l'algorithme le plus populaire de la famille des MCTS. Enfin, si on laisse UCT itérer à l'infini, celui-ci converge vers l'algorithme minimax.

$$UCB1 = X_j + \sqrt{\frac{2ln\ n}{n_j}} \tag{1.2}$$

où X_j correspond à la moyenne de récompenses pour le successeur j, n_j représente le nombre de fois que j a été choisi et n le nombre de décisions prises (Browne *et al.*, 2012).

1.3 État de l'art des programmes de jeu

En 1965, le mathématicien russe Alexander Kronrod a appelé les échecs « la drosophile de l'intelligence artificielle ». Alors que les généticiens se servent de la mouche drosophile pour faire des découvertes applicables à la biologie en général, l'IA s'est servi des échecs et d'autres jeux. Une meilleure analogie consisterait peut-être à dire que les échecs étaient à l'IA ce qu'un Grand Prix de Formule 1 (F1) est à l'industrie automobile. Un flux constant d'innovations, qui sont ensuite adoptées par le grand public, voit le jour grâce aux challenges que représentent les jeux pour les chercheurs. Dans cette section, on s'intéresse à des jeux qui représentent ou ont représenté un défi pour l'intelligence artificielle. On dresse un état de l'art sous forme d'historique, dans lequel on relate les performances de l'ordinateur en intelligence artificielle face à des jeux classiques. Les informations

ci-dessous constituent un résumé de la section 5.7 du livre *Artificial Intelligence : A Modern Approach (3rd Edition)* (Russell et Norvig, 2009).

Échecs. Le superordinateur Deep Blue d'IBM s'est rendu célèbre en battant le champion du monde Garry Kasparov (1997). Deep Blue effectuait une exploration alpha-bêta et examinait jusqu'à 30 milliards de positions par coup, il atteignait régulièrement une profondeur de 14. Aujourd'hui, un de ses successeurs, HYDRA, réalise des performances semblables à Deep Blue, mais à une profondeur de 18 coups. Le système RYBKA, multiple vainqueur des championnats mondiaux d'échecs pour ordinateurs, fonctionne même avec des composants du commerce. De plus, les récentes rencontres semblent confirmer la tendance et indiquer que les meilleurs programmes informatiques pour le jeu d'échecs ont distancé tous leurs adversaires humains. Les échecs ne sont donc plus au cœur de la recherche en IA.

Jeu de dames. Jonathan Schaeffer et ses collègues ont mis au point le système CHINOOK, qui fonctionne sur des PC ordinaires. Ce système utilise l'exploration alpha-bêta et joue depuis 2007 de façon parfaite.

Othello. Aussi appelé Reversi, ce jeu a un espace d'exploration plus petit que les échecs. Depuis 1997 (programme LOGISTELLO), on admet que les humains ne peuvent plus égaler les ordinateurs à ce jeu.

Backgammon. La composante aléatoire du jeu de backgammon rend l'exploration en profondeur très coûteuse. Gerry Tesauro (1992) a combiné une méthode d'apprentissage avec des réseaux de neurones pour développer un évaluateur. Après plus d'un million de parties d'entraînement, TD-GAMMON peut se mesurer aux meilleurs joueurs mondiaux. Par ailleurs, les indications du programme quant à certains coups ont parfois complètement modifié les conceptions jusqu'alors admises.

Bridge. L'élément d'incertitude du jeu a encouragé les meilleurs programmes à utiliser des méthodes de Monte-Carlo. Le programme GIB (Ginsberg, 1999) est le meilleur à ce jour et a atteint un niveau d'expert. Il n'a cependant pas réussi à battre les meilleurs joueurs humains, se classant douzième sur trente-cinq dans un tournoi au Championnat du monde humain. La simulation de Monte-Carlo traite efficacement le caractère aléatoire, mais ne traite pas toujours la stratégie.

Scrabble. Le problème du Scrabble vient du fait qu'il soit à la fois partiellement observable et stochastique : on ne sait pas quelles sont les lettres de l'autre joueur ni celles qu'on va tirer au prochain coup. En 2006, le programme QUACKLE a battu l'ancien champion du monde David Boys par 3 à 2. Néanmoins, le jeu n'est pas considéré comme résolu et aucun programme n'est reconnu pour avoir atteint le niveau expert.

Go. Le jeu de go a un espace d'états estimé autour de 10^{170}, alors qu'il est de 10^{50} pour les échecs. Par conséquent, les meilleurs programmes évitent l'exploration alpha-bêta et utilisent des méthodes de Monte-Carlo, qui ne permettent cependant d'obtenir qu'un niveau de bon amateur sur un échiquier complet. Défi complexe et à la pointe de l'intelligence artificielle, le jeu de go est resté, jusqu'en 2016, le jeu phare dans lequel aucun ordinateur n'avait battu un champion humain. Or, en mars 2016, le programme AlphaGo de Google a battu Lee Seedol, l'un des meilleurs joueurs mondiaux classé au niveau maximal (9^e dan professionnel). AlphaGo utilise des méthodes de Monte-Carlo, guidées par un réseau de valeurs et un réseau d'objectifs (Silver *et al.*, 2016).

Comme on a pu le voir, il existe des programmes capables de battre les humains à des jeux comme les échecs, othello et le backgammon sur du matériel informatique standard aujourd'hui. Les humains restent meilleurs dans divers jeux à information imparfaite, comme le poker et d'autres jeux ayant un facteur de branchement

important et pour lesquels on dispose de peu de savoir heuristique, comme le jeu de go. Maintenant qu'un ordinateur a battu un champion humain à ce jeu, la recherche en IA va se chercher de nouveaux défis, encore plus complexes. Pour la suite, de nombreux regards se tournent vers les jeux vidéo.

1.4 IA pour les jeux dans la recherche académique et IA dans les jeux vidéo

1.4.1 Situation actuelle

Lorsqu'on s'intéresse à l'intelligence artificielle dans les jeux, il apparaît clairement qu'une distinction existe entre le domaine de la **recherche académique** et celui de l'**industrie du jeu vidéo**. En effet, les deux mondes emploient des techniques et des algorithmes d'intelligence artificielle, mais il est encore rare que ce soient les mêmes. Le point d'entrée de ce débat se trouve dans la constatation que l'industrie du jeu vidéo emploie la plupart du temps des techniques simples, mais qui sont généralement suffisantes pour donner l'impression d'intelligence et de réalisme, cela dans le but de procurer une expérience de jeu intéressante aux joueurs. Par exemple, on peut citer l'algorithme A* ou les machines à états finis (FSM, pour *Finite-State Machine*) (Rabin, 2013). De l'autre côté, la recherche académique cherche à mettre au point des techniques de pointe et à démontrer leur efficacité.

Mais alors pourquoi l'industrie ne profite-t-elle pas également des avancées de la recherche académique ? Cela peut s'expliquer par plusieurs raisons. Tout d'abord, lors d'expérimentations pour la recherche, les algorithmes disposent de toute la puissance de calcul disponible pour s'exécuter. Or, dans un jeu vidéo, la plus grande partie du temps de calcul par image est utilisée autre part et l'IA doit faire avec l'environnement et les ressources restantes, ou encore celles qu'on veut bien lui attribuer. L'écart est grand si on compare avec les ressources offertes aux techniques de pointe de la recherche (Dicken, 2011).

La raison la plus importante est sans doute le fait que les objectifs ne sont pas les mêmes de part et d'autre. En effet, si la recherche en intelligence artificielle tend à toujours aller de l'avant en gagnant en efficacité et en performance, l'IA dans les jeux a pour sa part comme objectif principal de divertir ses joueurs. L'objectif n'est pas ici de battre le joueur, mais au contraire de perdre de façon à le satisfaire.

De plus, l'industrie doit composer avec des contraintes monétaires plus strictes que le milieu académique. En effet, dans l'industrie, un jeu doit rapporter de l'argent, ce qui représente un argument de poids lors de la réflexion autour de l'IA. Pourquoi prendre beaucoup de temps pour implémenter des systèmes complexes, alors que des solutions plus accessibles, suffisantes, et qu'on maitrise déjà existent ? Pourquoi prendre le risque dans ce marché difficile ?

Enfin, alors que la recherche académique a pour objectif de résoudre des problèmes réels et ne peut pas tricher pour y parvenir, l'intelligence artificielle dans les jeux vidéo peut se permettre de tricher. Effectivement, ce qui compte dans ces œuvres de divertissement est l'expérience finale. Les moyens utilisés pour y parvenir ne sont la plupart du temps pas visibles ni importants. Ainsi, l'IA dans les jeux a le droit de tricher avec les règles de son monde pour être omnisciente, ou encore de délibérément rater sa cible pour le bien de l'expérience à délivrer.

1.4.2 Vers une convergence ?

Lorsqu'on parle d'intelligence artificielle dans les jeux vidéo, on se limite souvent à l'hypothèse que l'IA est utilisée uniquement pour contrôler des personnages non joueurs. En fait, c'est seulement une des nombreuses applications que peuvent trouver les diverses techniques d'intelligence artificielle dans les jeux. Les techniques d'IA peuvent aussi être utilisées pour générer du contenu (niveaux, cartes, règles, puzzles, etc.), modéliser des personnages, adapter des aspects du jeu (comme la difficulté), associer des joueurs pour des parties en ligne, contrô-

ler des systèmes et des économies artificielles, etc. Tous ces éléments constituent des problèmes intéressants auxquels la recherche académique peut apporter des réponses pertinentes (recherche de chemin, planification, etc.) (Togelius, 2011).

En plus des progrès rapides du matériel informatique qui lui permettent de ne plus forcément être limitant, des signes prometteurs commencent à apparaître dans l'industrie. En effet, des techniques plus avancées d'intelligence artificielle ont été implémentées avec succès dans des jeux vidéo commerciaux. On en trouve dans des jeux comme *Black & White* (apprentissage-machine), *F.E.A.R* (comportement en fonction du contexte), *Façade* (interprétation du langage naturel), *Spore* (simulation de forme de vie dirigée par les données) et *les Sims* (interactions entre agents), pour en nommer quelques-uns (Graft, 2015).

De plus en plus d'éléments comme le *game design* utilisent de l'intelligence artificielle et celle-ci s'impose petit à petit. Les outils utilisés pour assister la conception des jeux commencent eux aussi à en bénéficier et l'exploitation de données massives (*Big Data*) ainsi que l'apprentissage machine vont prendre une place majeure à l'avenir. Si les programmeurs en intelligence artificielle dans l'industrie du jeu vidéo ne s'accordent pas tous à propos de la façon dont l'IA va s'intégrer plus profondément dans le domaine, la convergence entre l'intelligence artificielle dans la recherche académique et celle employée dans l'industrie du jeu vidéo est bien en cours et elle sera bénéfique pour les deux parties (Graft, 2015).

CHAPITRE II

JEUX DE STRATÉGIE EN TEMPS RÉEL : UN DÉFI POUR L'INTELLIGENCE ARTIFICIELLE

Comme vu au chapitre précédent, certains jeux suscitent un intérêt particulier dans le domaine de l'intelligence artificielle en raison de leurs spécificités, comme les échecs, le jeu de go ou d'autres. Aujourd'hui, les avancées en intelligence artificielle permettent de pouvoir être toujours plus exigeant et optimiste et de se fixer des défis de plus en plus complexes. Nés grâce à l'informatique, les jeux vidéo entendent garder la même philosophie que leurs ainés les jeux et peuvent être considérés comme un des prochains grands challenges de l'intelligence artificielle en raison de la puissance de leur support.

Dans ce chapitre, on s'intéresse plus précisément au jeu vidéo StarCraft en raison de ses caractéristiques, qui sont exposées à la section 2.1. La recherche dans ce jeu s'articule autour de la création d'un bot (module logiciel qui joue une partie comme un humain) avec lequel les chercheurs mènent leurs expérimentations. On traite de ce sujet dans la section 2.2, en réalisant un état de l'art du domaine. Enfin, la section 2.3 est consacrée à la communauté de recherche qui s'est formée autour du jeu et aux outils qu'elle emploie. On conclut ce chapitre par l'étude en détail d'un bot (section 2.3.3) afin de comprendre les tenants et les aboutissants de la recherche en IA dans StarCraft.

2.1 Le jeu StarCraft : Brood War

2.1.1 Description du jeu

StarCraft est un jeu de stratégie militaire en temps réel (RTS) prenant place dans un univers de science-fiction développé par Blizzard Entertainment. Il est sorti en 1998 a été suivi rapidement, la même année, par une extension nommée **Brood War**. Dix ans plus tard (2007), le jeu est toujours apprécié ; il s'est vendu à près de dix millions de copies.

Le jeu de stratégie en temps réel est un sous-genre des jeux de stratégie dans lequel les joueurs doivent construire une économie (récolter des ressources et construire une base) et une force militaire (former des unités et mener des recherches technologiques) dans le but de battre leurs adversaires (détruire leur armée et leur base). D'un point de vue théorique, il y a plusieurs différences entre les RTS et les jeux déterministes à somme nulle, plus classiques, comme les échecs.

Tout d'abord, les RTS sont des jeux à mouvement simultané, c'est-à-dire que plusieurs joueurs peuvent effectuer des actions en même temps. Ces actions ne sont pas instantanées, c'est-à-dire que leurs effets peuvent prendre du temps à se concrétiser. Aussi, on dit de ces jeux qu'ils sont en temps réel, ce qui signifie que chaque joueur a peu de temps pour décider de sa prochaine action. En comparaison avec les échecs, où les joueurs disposent de plusieurs minutes pour réfléchir à leur prochain coup, dans StarCraft, le jeu affiche 24 images par secondes. Ainsi les joueurs ont seulement 42 millisecondes pour agir avant que l'état du jeu ne change. Ensuite, la plupart des jeux de ce genre, dont StarCraft, sont partiellement observables. En effet, les joueurs ne peuvent pas voir l'entièreté de la carte au départ ; chaque joueur peut uniquement voir la portion de la carte qu'il a déjà explorée (on parle de « brouillard de guerre »). Ces jeux sont également non déterministes, car certaines actions ont des chances de succès.

Enfin, la complexité des RTS est élevée, aussi bien en ce qui concerne la taille de l'espace d'états que le nombre d'actions possibles à chaque cycle de décision. Par exemple, la taille de l'espace d'états des échecs est estimée approximativement à 10^{50}, celle du Poker (Texas Hold'em) à environ 10^{80} et celle du jeu de go à environ 10^{170}. En comparaison, la taille de l'espace d'états d'une partie de StarCraft est estimée à plusieurs ordres de grandeur de plus (Ontañón *et al.*, 2013).

Pour remporter la victoire, les joueurs de **StarCraft** doivent gérer leur économie, leurs technologies et leur armée de façon rapide et stratégique. Les principes de base sont, d'une part, de récolter deux types de ressources (minerai et gaz) et, d'autre part, de les utiliser pour acheter des bâtiments, des améliorations et des unités de combat. En prenant du recul sur le déroulement d'une partie, on distingue deux tâches principales pour le joueur. La **macro-gestion** (macromanagement ou macro) et la **micro-gestion** (micromanagement ou micro). Tandis que la micro-gestion réfère aux événements locaux et plus spécialisés, la macro-gestion concerne le déroulement à plus haut niveau de la partie et la dimension économique. Il est important pour le joueur de StarCraft de bien maîtriser ces aspects et d'assurer l'équilibre entre les deux.

Le micromanagement englobe la gestion de l'armée. On s'occupe des déplacements, de l'attaque et de la retraite avec l'objectif de surpasser l'adversaire. La micro est l'habileté à contrôler les unités individuellement de façon à optimiser le comportement imparfait des unités du jeu (en particulier pour le déplacement). Par exemple, on cherche à éviter les pièges ennemis en les contournant, alors qu'un ordre direct de déplacement vers une cible (laissé aux mains du système de déplacement du jeu) ira droit dedans. En général, le principe de la micro est de garder le plus d'unités possible en vie.

Le macromanagement englobe pour sa part la gestion de l'économie et de la tech-

nologie : on s'assure de récolter suffisamment de ressources et de les dépenser correctement. La macro est l'habileté à produire des unités et à conserver tous les bâtiments actifs durant la partie. Habituellement, un joueur avec une meilleure capacité de macro que son adversaire parvient à produire une armée plus grande. Un autre élément de la macro-gestion est la capacité de s'étendre aux temps appropriés pour permettre une production constante d'unités. Un bon joueur de macro a la capacité d'augmenter sa capacité de production tout en s'assurant de pouvoir disposer de suffisamment de ressources pour en profiter.

Toutes ces opérations prennent place en temps réel ; les commandes sont exécutées quand elles sont reçues. Cela signifie qu'un joueur ne doit pas seulement être bon stratégiquement, mais aussi rapide, habile et capable de faire du multitâche. Cette capacité est d'ailleurs mesurée en APM (actions par minutes).

Si l'idéal est de faire simultanément de la micro et de la macro, il est impossible pour un joueur humain de faire les deux tâches en même temps aussi bien que s'il en faisait une seule. Ainsi, le joueur est forcé de se concentrer sur une de ces tâches à la fois, et la subtilité se trouve dans l'équilibre à trouver entre les deux. On considère généralement qu'une bonne macro est plus importante qu'une bonne micro. En effet, si on considère deux joueurs au même niveau d'APM, que l'un est orienté micro et l'autre macro, le joueur macro aura une meilleure économie et plus d'unités et pourra dominer son adversaire, peu importe la qualité de sa micro. Les capacités de micro entrent en jeu et permettent de départager deux joueurs lorsque ces derniers sont du même niveau de macro. En ce sens, c'est à nouveau la capacité de jongler entre les deux tâches qui fait la différence (Liquipedia, 2016).

Une carte de StarCraft est constituée d'un rectangle, où la longueur et la largeur sont mesurées par le nombre de carrés de 32*32 pixels alignés verticalement et horizontalement. Ces carrés de 32*32 pixels correspondent aux cases employées pour

placer les bâtiments, alors que les cases utilisées pour le déplacement des unités sont de 8*8 pixels. La taille des cartes varie généralement de 64*64 à 256*256 cases (pour les bâtiments). Chaque joueur peut contrôler jusqu'à 200 unités et posséder un nombre illimité de bâtiments. De plus, le joueur incarne une race, qui dispose de 30 à 35 types d'unités et de bâtiments différents, dont la plupart possèdent des capacités spéciales. Tous ces facteurs font de StarCraft un challenge complexe, dans lequel les humains sont toujours bien meilleurs que les ordinateurs.

Figure 2.1 Un début de partie dans StarCraft (Robertson et Watson, 2014).

D'un point de vue théorique, l'espace d'états d'une partie de StarCraft est très important. Considérons par exemple une carte de 128*128. À tout moment, on peut compter entre 50 et 400 unités sur la carte, et chacune présente un état interne

complexe (points de vie, énergie restante, action en cours, etc.). Cette situation mène rapidement à un nombre immense d'états possibles, bien plus grand que pour les échecs ou le jeu de go. En considérant uniquement la position des unités (avec 128*128 positions possibles par unité), pour 400 unités, on obtient environ $(128 * 128)^{400} \approx 10^{1685}$ (Ontañón *et al.*, 2013), et ce, sans prendre en compte les autres facteurs.

Un autre moyen de mesurer la complexité du jeu est de regarder le facteur de branchement b et la profondeur du jeu p, comme proposé par Synnaeve (2012), avec une complexité totale de b^p. Aux échecs, $b \approx 35$ et $p \approx 80$; au jeu de go, $b \approx 30$ à 300 et $p \approx 150$ à 200. Pour déterminer le facteur de branchement dans StarCraft, il faut penser au fait qu'un ordinateur peut effectuer des actions en simultané avec autant d'unités que souhaité. En considérant cela pour 50 à 200 unités, le facteur de branchement se situe entre u^{50} et u^{200}, où u est le nombre moyen d'actions que chaque unité peut exécuter. Estimer u est difficile étant donné que le nombre d'actions disponibles dépend du contexte. En simplifiant beaucoup le problème (faible nombre d'ennemis, mouvements limités, etc.) et en prenant en compte le fait que certaines actions ont des durées d'indisponibilité (*cooldowns*), on arrive à une estimation de 10 actions possibles par unité par image. Le facteur de branchement b est donc estimé entre 10^{50} et 10^{200}, en considérant seulement les unités (sans les bâtiments et leurs actions). Quant à la profondeur p, si on considère qu'une partie dure environ 25 minutes, on arrive à $p \approx 36000$ (25 min * 60 sec * 24 images par seconde) (Ontañón *et al.*, 2013).

Le joueur peut choisir une race parmi les *Protoss*, les *Terrans* et les *Zergs*. Chacune des races a ses qualités, faiblesses, unités, capacités et mécaniques de jeu. Les *Protoss* sont une race psyonique et technologique comportant des unités plus coûteuses que les autres races, parce qu'elles sont plus fortes. Toutes ces unités sont équipées d'un bouclier qui se régénère. Les *Protoss* sont la race la plus puissante

en début et en fin de partie. En termes de mobilité par contre, ils sont moyens par rapport aux autres races.

Les *Terrans* sont une race humaine dont les unités présentent le ratio coût/efficacité le plus élevé. Ils ont la réputation d'être la race la plus adaptable. Une de leurs particularités est qu'ils peuvent soigner et réparer les unités et bâtiments.

Les *Zergs* sont une race xénomorphe, dont les unités sont les moins coûteuses. Ils sont considérés comme la race la plus mobile. Leurs unités sont par conséquent les plus faibles, mais ils comptent sur leur grand nombre pour compétitionner à force égale avec les autres races (Liquipedia, 2016).

2.1.2 Aperçu des stratégies

À travers le temps, StarCraft a évolué sur le plan compétitif pour devenir un « e-sport » impliquant beaucoup de tactiques et de manières de jouer. Le jeu est pratiqué professionnellement depuis 1998, en particulier en Corée du Sud. Il existe de nombreuses stratégies recensées et utilisées au niveau compétitif pour StarCraft (Liquipedia, 2016). Le choix d'une stratégie est en général lié à la race jouée ainsi qu'à la race de l'adversaire. En effet, les particularités de chaque race rendent certaines stratégies très efficaces ou, au contraire, non pertinentes. Le choix de stratégie détermine également le type d'ouverture qu'on applique en tout début de partie.

De la même façon qu'aux échecs, l'ouverture est le premier composant lié à la stratégie qu'on souhaite appliquer. Dans StarCraft, c'est la séquence de bâtiments et d'unités qui est créée au début de la partie. On parle de *build-order* pour désigner une séquence d'unités ou de bâtiments à créer.

Pour cette section, on s'intéresse aux stratégies générales qui sont la plupart du temps mises en œuvre (Liquipedia, 2016). Ces stratégies générales sont ensuite

déclinées en de multiples variantes par les joueurs (timing d'exécution, type ou nombre d'unités, etc.). Elles sont plus ou moins risquées à employer en jeu, et sont utilisées à différents moments d'une partie ainsi qu'à différents niveaux d'engagement de la part du joueur.

Le *scouting*

Le *scouting* est une partie intégrante du jeu qui consiste à espionner son adversaire pour découvrir sa stratégie. Tous les joueurs présentant un bon niveau de jeu emploient ce concept. Le *scouting* est important à tout moment, mais il est surtout réalisé en tout début de partie. L'objectif est d'observer l'ouverture de l'adversaire pour pouvoir ajuster sa propre stratégie. Il permet également de déterminer la composition et la position de l'armée de l'adversaire, ainsi que de découvrir si celui-ci s'est déjà déplacé et étendu sur la carte (en créant notamment de nouvelles bases).

Le *rush*

Cette stratégie se résume à attaquer le plus rapidement possible l'adversaire en investissant toutes les ressources dans la production d'unités, dans le but de submerger l'adversaire et de le battre avant qu'il n'ait pu se développer ou même se défendre. Bien que très efficace, notamment avec la race des *Zergs* en raison des ses spécificités, le *rush* reste une tactique à double tranchant. En effet, utiliser cette technique revient à jouer quitte ou double, à cause de l'engagement de toutes les ressources du joueur. Les joueurs avancés savent se protéger d'un *rush* en début de partie et parviennent alors à retourner la partie à leur avantage et à battre facilement l'adversaire dont le *rush* a échoué. Ce dernier se retrouve alors trop en retard (en termes de ressources) dans la partie pour pouvoir lutter. En revanche, si le *rush* réussit, la partie est gagnée rapidement.

L'expansion

Créer de nouvelles bases et s'étendre sur la carte pour dominer en termes de possession du terrain est un enjeu important dans une partie de StarCraft. En réalité, le nombre de bases détenues par le joueur et ayant des ressources proches à exploiter est un des facteurs décisifs pour le dénouement d'une partie. Étant donné que les ressources sont très importantes pour le développement, le joueur qui en possède ou en sécurise le plus afin que son adversaire n'en profite pas est en position de force. Souvent, avoir un plus grand nombre de bases que l'adversaire est l'objectif principal d'une partie.

La guérilla

Le principe est d'abuser de la composante micro de StarCraft en harassant constamment l'adversaire pour l'obliger lui aussi à faire beaucoup de micro. En appliquant cette technique, on ne cherche pas forcément à affronter l'adversaire, mais plutôt à lancer contre lui de petites attaques localisées tout en évitant les gros conflits. Le but est ici d'éliminer les unités de l'adversaire petit à petit et de jouer avec ses nerfs en l'obligeant sans cesse à abandonner ses projets de macromanagement, sous peine de perdre ses unités.

Le sabotage

Cette stratégie consiste à saper la capacité de récolte et de production de ressources de l'ennemi (en éliminant ses unités de récolte ou de construction) dans le but de le paralyser et de ralentir son développement. Saboter l'approvisionnement de l'ennemi est très efficace en début de partie ; combinée au *scouting*, cette technique peut lui être fatale. L'effet est problématique pour l'adversaire à tout moment de la partie, mais il se fera moins sentir en fin de partie, alors que les deux forces auront déjà constitué une armée conséquente.

Le siège

Le siège est aussi une stratégie classique. Dans StarCraft, le joueur va avancer rapidement vers le camp adverse pour tenter de le contenir et de limiter son expansion. Ainsi, l'ennemi va à terme manquer de place pour se développer et surtout, tomber à court de ressources.

Le *kiting*

L'objectif est de provoquer et d'attirer l'ennemi, de le pousser à suivre alors que l'on recule, dans le but de le déplacer vers la position de notre choix. C'est une stratégie utilisée souvent en combinaison avec la guérilla, avec laquelle on attaque un peu l'adversaire en fuyant vers notre base lorsqu'il riposte, pour ensuite neutraliser ses unités, si elles suivent, avec les défenses de la base.

L'embuscade

L'embuscade est une stratégie militaire élémentaire qu'on peut aussi appliquer dans StarCraft. On va cacher des unités à un endroit stratégique pour piéger l'adversaire, en utilisant par exemple l'avantage conféré par un terrain favorable. On peut également profiter de l'effet de surprise et tenter de perturber la micro-gestion de l'ennemi pour le déjouer. Logiquement, le *kiting* peut aussi servir à mener les unités adverses dans une embuscade.

Le *dropping*

Dans la même veine que le sabotage, mais accessible en milieu et en fin de partie, le *dropping* reprend le concept de frappe, cette fois en plein cœur d'une base adverse. Cette technique nécessite la capacité de transporter des unités (qu'on doit acquérir en cours de partie). L'idée est d'envoyer des unités directement

dans une base ennemie par voie aérienne, dans le but de causer un maximum de dommages en visant les points clés, comme les unités de récolte et de construction, les bâtiments qui produisent des unités offensives, les défenses gênantes pour un assaut, etc.

2.1.3 Challenges pour l'intelligence artificielle

En 2003, les premières recherches en intelligence artificielle pour les RTS ont identifié les challenges suivants (Buro, 2003) :

- La gestion des ressources ;
- La prise de décision sous incertitude ;
- Le raisonnement spatial et temporel ;
- La collaboration (entre plusieurs IA) ;
- La modélisation de l'adversaire et l'apprentissage ;
- La planification en temps réel et en situation d'adversité.

Alors que beaucoup de travail a été effectué pour relever certains de ces challenges, d'autres ont été moins explorés. De plus, de nouveaux challenges ont été identifiés plus récemment et regroupés en six catégories principales par les auteurs d'un article très complet sur le sujet (Ontañón *et al.*, 2013).

Le premier est la **planification**. Comme on l'a vu à la section 2.1.1, la taille de l'espace d'états dans les RTS est beaucoup plus grande que pour les échecs ou le jeu de go. Ainsi, les approches standards de planification en situation d'adversité ne sont pas applicables directement. La planification dans les jeux comme StarCraft peut être vue selon plusieurs niveaux d'abstraction, mais il n'existe pas encore de techniques efficaces en pratique pour résoudre ce genre de problème.

Le challenge suivant est l'**apprentissage**. On peut distinguer trois types de problèmes d'apprentissage dans les RTS. L'apprentissage à partir des données dispo-

nibles : la question qui se pose est comment exploiter les données que constituent les enregistrements de parties ou les informations à propos des cartes, pour apprendre des stratégies à l'avance ? Beaucoup de travail est effectué dans cette direction (Ontañón *et al.*, 2013). De plus, on s'intéresse aussi à l'apprentissage en cours de jeu : comment les bots peuvent améliorer leur stratégie pendant la partie ? La réponse est grâce à l'apprentissage par renforcement ou de la modélisation de l'adversaire, mais le problème vient du fait que les RTS sont partiellement observables. Enfin, l'apprentissage entre parties : que peut-on apprendre pour augmenter les chances de victoire dans la partie suivante ? Certains bots choisissent entre plusieurs stratégies, mais le problème reste non résolu.

Un autre challenge réside dans l'**incertitude**. La planification en situation d'incertitude dans des domaines de la taille des RTS est encore à ce jour un problème sans solutions applicables. Le jeu est partiellement observable et un joueur ne peut pas prédire avec certitude les actions de son adversaire. Pour ce genre d'incertitude, l'IA, comme l'humain, ne peut que se construire un modèle de ce que l'adversaire va probablement faire.

L'**exploitation des connaissances du domaine** est aussi un challenge. Pour les échecs, les chercheurs ont exploité les larges connaissances existant sur le domaine. Pour les RTS, il est encore difficile de trouver comment les bots peuvent exploiter les nombreuses connaissances qui existent. Certains bots implémentent directement plusieurs stratégies et choisissent laquelle appliquer en début de partie, mais apprendre de façon automatique depuis les données est toujours un problème ouvert.

Un autre challenge s'articule autour de la **décomposition des tâches**. À cause de la complexité des jeux comme StarCraft, toutes les approches existant pour faire jouer des bots décomposent la partie en un ensemble de plus petits problèmes.

Avec cette décomposition, le challenge important se situe dans la modélisation d'architectures qui permettent aux différentes techniques d'intelligence artificielle de communiquer entre elles et de travailler ensemble, de résoudre des conflits, etc.

Enfin, le dernier challenge est le **raisonnement spatial et temporel**. Le raisonnement temporel joue un rôle clé dans la raisonnement stratégique, par exemple pour attaquer et se replier au bon moment afin de prendre l'avantage. À un plus haut niveau stratégique, le joueur doit réfléchir à quand lancer des actions aux effets et à l'impact économique à long terme (améliorations, constructions, changements de stratégie, etc.). Quant au raisonnement spatial, on lui consacre la section 3.1 pour l'étudier en détail.

2.2 État de l'art de l'IA appliquée à StarCraft

Les bots qui jouent à StarCraft doivent résoudre beaucoup de problèmes, dont ceux évoqués à la section 2.1.3. On constate que d'une manière générale, ils semblent souvent adopter des approches ad hoc. Pour faciliter la lecture de cet état de l'art, on divise le travail existant dans le domaine en trois niveaux d'abstraction : la **stratégie** (qui se rapproche de la macro), la **tactique** et la **réaction** (qui se rapproche de la micro). La stratégie correspond au processus de prise de décision qui porte sur une longue durée (plusieurs minutes dans StarCraft). La réaction, au contraire, correspond aux décisions immédiates de bas niveau (de l'ordre de la seconde ou moins). La tactique se trouve entre les deux. Aussi, les décisions stratégiques prennent en compte la totalité de la partie, alors que les décisions tactiques et de réaction sont localisées et affectent uniquement des groupes spécifiques d'unités. En général, les décisions stratégiques conditionnent les décisions tactiques, qui elles-mêmes conditionnent les réactions. Cependant, l'effet inverse peut aussi se produire, puisque les informations recueillies lors de la réaction peuvent influencer la tactique, qui peut entraîner un nouveau raisonnement stratégique.

En suivant cette idée, on considère dans la stratégie tout ce qui est lié aux améliorations, aux ouvertures et à la composition d'armées. C'est le niveau qui demande le plus de réflexion à un joueur, puisque ce dernier choisit sa stratégie (qui va définir son attitude dans la partie) en gardant l'aspect tactique en tête. On englobe tout ce qui est lié aux confrontations entre groupes d'unités dans la tactique. Le raisonnement tactique inclut le raisonnement spatial et temporel. Enfin, la réaction décrit comment le joueur ou l'IA contrôle chaque unité individuellement pour maximiser son efficacité en temps réel. La différence entre la tactique et la réaction est que le raisonnement tactique implique une planification à l'avance, alors que la réaction correspond à un *réflexe* face à un mouvement de l'adversaire.

Par exemple, au tout début de la partie, un joueur peut essayer d'utiliser une stratégie de *rush* (qui implique de produire rapidement une armée et d'attaquer aussi vite que possible), puis utiliser une tactique d'encerclement lors de l'attaque. Pendant l'encerclement, il peut décider d'utiliser des techniques réactives pour effectuer des petites attaques suivies de repli, et ce, de façon répétée avec des unités individuelles, pour tenter de maximiser leur efficacité et ainsi perturber l'adversaire (Ontañón *et al.*, 2013).

2.2.1 Approches au niveau stratégie

La prise de décision stratégique dans des environnements en temps réel est encore un problème ouvert. Dans le contexte des RTS, plusieurs techniques d'intelligence artificielle ont été mises à l'essai, comme les approches scriptées (prédéfinies), les approches à base de planification ou celles basées sur l'apprentissage machine. On s'y intéresse dans cette section, qui résume les principales approches recensées par Ontañón et ses collègues (Ontañón *et al.*, 2013).

Les approches scriptées ont été très utilisées dans les jeux issus de l'industrie. Les plus communes sont basées sur des machines à états finis (FSM), ce qui permet au

développeur d'écrire directement la stratégie que le bot va employer (Fu et Houlette, 2004). L'idée derrière les FSM est de décomposer le comportement de l'IA en des états faciles à gérer. On définit, par exemple, les états « attaquer », « récolter des ressources » et « réparer », et on n'a plus qu'à spécifier les conditions qui entraînent les transitions entre eux. On trouve également des jeux commercialisés qui incluent des machines à états composées de façon hiérarchique. Ces approches scriptées ont connu un bon succès et sont utilisées dans beaucoup de projets de recherche académique dans les RTS. Cependant, ces techniques peinent à délivrer des comportements adaptatifs et dynamiques et sont facilement exploitables par les adversaires.

Deuxièmement, les approches qui utilisent des techniques de planification ont aussi été explorées dans la littérature. Elles offrent plus d'adaptabilité stratégique que les méthodes scriptées. Cependant, les contraintes des RTS limitent les approches possibles. Seulement deux solutions ont été explorées pour le moment : les HTN (*Hierarchical Task-Network*) (Hoang *et al.*, 2005) et la planification basée sur les cas (CBP, pour *Case-Based Planning*) (Ontañón *et al.*, 2008). Malheureusement, aucune de ces approches ne prend en considération les problèmes de temps ou d'ordonnancement qui sont majeurs dans les RTS. Une exception notable est le travail de Churchill et Buro, qui ont utilisé la panification pour un système de production d'unités prenant en compte l'économie et les contraintes de temps des différentes actions (Churchill et Buro, 2011).

En ce qui concerne l'apprentissage machine, Weber et Mateas ont proposé une approche de fouille de données pour de la prédiction de stratégie et font de l'apprentissage supervisé sur des enregistrements de parties de StarCraft (Weber et Mateas, 2009). Dereszynski et al. utilisent des modèles de Markov cachés (HMM, pour *Hidden Markov Model*) pour apprendre les probabilités de transitions de séquences d'ordre de construction et gardent les plus probables pour produire des

modèles probabilistes de comportement (Dereszynski *et al.*, 2011). Synnaeve et Bessière présentent un modèle bayésien semi-supervisé pour apprendre des enregistrements et prédire les ouvertures de parties (Synnaeve et Bessière, 2011a). Ils proposent également un modèle d'apprentissage bayésien non supervisé pour la prédiction de l'arbre technologique d'une partie de StarCraft, en utilisant des parties enregistrées (Synnaeve et Bessière, 2011b).

Toujours dans la catégorie de l'apprentissage machine, beaucoup de chercheurs ont utilisé des approches de raisonnement basé sur les cas (CBR : *Case-Based Reasoning*) pour la prise de décision stratégique (Aamodt et Plaza, 1994). Hsieh et Sun emploient un modèle de CBR avec des parties enregistrées pour monter des séquences de construction de bâtiments (Hsieh et Sun, 2008). À leur tour, Certicky et al. utilisent du CBR pour construire leur armée en fonction de la composition de l'armée adverse, et pointent l'importance d'un bon *scouting* pour obtenir de bons résultats (Certický et Certický, 2013).

Enfin, on sait que StarCraft est un jeu partiellement observable, ce qui signifie que le joueur doit découvrir la carte de jeu et peut uniquement voir les zones proches de ses unités. Les zones hors de son champ de vision ne donnent aucune information. C'est une des raisons qui poussent les joueurs à faire du *scouting*. La taille de l'espace d'états dans StarCraft empêche l'application directe de solutions à base de processus de décision de Markov partiellement observables (POMDP, pour *Partially Observable Markov Decision Process*), et peu d'approches tentent de résoudre ce problème. Concernant le problème de l'information imparfaite, un travail notable est celui de Weber et al., qui utilisent un modèle de particules avec mise à jour linéaire de trajectoire pour suivre le déplacement des unités de l'adversaire quand elles ne sont plus visibles (Weber *et al.*, 2011a). Ils produisent également des objectifs tactiques grâce à de la planification réactive et à de l'autonomie dirigée par les buts.

2.2.2 Approches au niveau tactique

Le raisonnement tactique implique la prise en compte des différentes capacités des unités dans un groupe, de l'environnement et de la position des différents groupes d'unités dans cet environnement, dans le but de gagner l'avantage militaire pendant le combat. Par exemple, envoyer des unités rapides, invisibles ou volantes (unités coûteuses) en première ligne contre des unités lourdes est une mauvaise idée, car elles seront vite abattues. On divise cette section en deux parties, l'analyse de terrain et la prise de décision.

L'analyse de terrain apporte des informations structurées et très importantes sur la carte à l'IA pour l'aider lors de la prise de décision. Cette analyse est souvent réalisée une seule fois par carte en tout début de partie et est ensuite sauvegardée dans un fichier à part, pour pouvoir être réutilisée à chaque partie afin d'économiser du temps CPU en jeu. Pour StarCraft, Perkins applique une décomposition de Voronoï (avec élagage) pour détecter des régions et goulots d'étranglement pertinents sur les cartes. Cette technique est implémentée dans la librairie BWTA qu'on étudie à la section 2.3.2 (Perkins, 2010).

D'autre part, le *walling* est le fait de placer des bâtiments à l'entrée de sa propre base afin de bloquer le chemin et d'empêcher l'ennemi de pénétrer à l'intérieur. Cette technique est souvent utilisée pour survivre aux agressions rapides et permet de gagner du temps pour produire plus d'unités. Certicky a reproduit cette technique en la transformant en un problème de satisfaction de contraintes et en utilisant de la programmation déclarative (ASP : *Answer Set Programming*) (Certický, 2013).

Pour la prise de décision stratégique, de nombreuses approches ont été étudiées, comme l'apprentissage machine ou les techniques d'exploration d'arbres. Hladky et Bulitko ont testé des demi-modèles de Markov cachés (HSMM, pour *Hidden*

Semi-Markov Model) et des filtres de particules pour le suivi d'unités (Hladky et Bulitko, 2008). Bien que leur expérimentation soit faite sur des jeux de tir à la première personne (FPS, pour *First-Person Shooter*), les résultats peuvent s'appliquer aux RTS également. Ils montrent que la précision des cartes d'occupation (*occupancy map*) est améliorée dans les HSMM en utilisant des modèles de déplacement (appris du comportement des joueurs). Kabanza et al. améliorent un système probabiliste de suivi des tâches de l'adversaire en encodant les stratégies en tant qu'HTN, qu'ils utilisent pour faire de la reconnaissance de plan et trouver des opportunités tactiques (Kabanza *et al.*, 2010). Sharma et al. combinent du CBR et de l'apprentissage par renforcement pour permettre la réutilisation de parties de plans tactiques (Sharma *et al.*, 2007). Une autre technique combine les informations du travail de Perkins et de la prise de décision tactique, en assignant des scores aux régions et en cherchant des correspondances dans des enregistrements de parties de joueurs professionnels. Enfin, Miles et Louis donnent l'idée de l'*IMTrees*, un arbre dans lequel chaque feuille est en fait une carte d'influence et chaque nœud intermédiaire est une opération de combinaison (addition, multiplication) (Miles et Louis, 2007). Ils utilisent ensuite des algorithmes évolutionnistes pour apprendre des IMTrees pour chaque décision dans le jeu qui implique du raisonnement spatial, en combinant des cartes d'influence basiques.

Les techniques de recherche dans un arbre de jeu ont aussi été explorées pour la prise de décision tactique. Churchill et Buro présentent l'algorithme ABCD (*Alpha-Beta Considering Durations*), un algorithme d'exploration d'arbres pour les batailles tactiques dans les RTS (Churchill *et al.*, 2012). D'autres emploient des techniques de Monte-Carlo pour faire de la planification d'attaque. Pour que les techniques de recherche dans un arbre soient applicables à ce niveau, de fortes abstractions de l'état du jeu doivent être faites pour en réduire la complexité, de même que des abstractions ou des simplifications de l'ensemble des actions pos-

sibles. De plus, le *scouting* est aussi important pour la prise de décision tactique, que stratégique. Malgré tout, peu de travail a été effectué en ce sens. Enfin, il faut préciser que les approches de cette section, incluant les techniques de recherche d'arbre, font l'hypothèse d'avoir des informations parfaites (Ontañón *et al.*, 2013).

2.2.3 Approches au niveau de la réaction

Dans la catégorie de la réaction, on cherche à maximiser l'efficacité des unités, incluant le contrôle en simultané de plusieurs unités de différents types, dans des batailles complexes et sur des terrains hétérogènes.

Les champs de potentiel et les cartes d'influence sont les techniques les plus utilisées pour la prise de décision réactive. Quelques utilisations des champs de potentiel incluent l'évitement d'obstacles, l'évitement des tirs de l'ennemi ou le fait de se maintenir hors de sa portée. Ils ont aussi été combinés avec l'algorithme de recherche de chemin A* pour faire de l'évitement de piège. Hagelbäck et Johansson présentent un bot multi-agent basé sur des champs de potentiel pour résoudre le problème du « brouillard de guerre » (Hagelbäck et Johansson, 2008). Danielsiek et al. utilisent des cartes d'influence pour créer des mouvements de groupe intelligents, afin de prendre les ennemis à revers (Danielsiek *et al.*, 2008). Malgré leur succès dans les RTS, un inconvénient des techniques à base de champs de potentiel est le grand nombre de paramètres qui doivent être ajustés pour produire le comportement désiré. Des approches pour apprendre automatiquement ces paramètres ont été explorées, par exemple en utilisant de l'apprentissage par renforcement ou des SOM (*self-organizing-maps*) (Preuss *et al.*, 2010). Il est important de noter que les champs de potentiel représentent une technique totalement réactive ; ils ne font aucune sorte d'anticipation. Par conséquent, ces techniques sont sensibles aux problèmes d'optimums locaux (Ontañón *et al.*, 2013).

Il y a également eu du travail du côté des techniques d'apprentissage machine

pour les problèmes de réaction. Un modèle bayésien a été utilisé pour améliorer les résultats des champs de potentiel, ce qui permet une intégration des objectifs tactiques directement dans la micro-gestion. De plus, Madeira et al. ont aussi essayé de l'apprentissage par renforcement (RL, pour *Reinforcement Learning*) pour faire de la micro-gestion décentralisée (Madeira *et al.*, 2006). Ils conseillent l'utilisation de connaissances du domaine pour accélérer l'apprentissage, en raison de la taille de l'espace d'états d'une partie de RTS. Cela requiert d'exploiter la structure existante du jeu dans un processus de décision de Markov partiel, accompagné d'une heuristique. D'autres approches qui cherchent à apprendre un sous-modèle du jeu ont aussi été explorées. Par exemple, Ponsen fait l'essai de techniques d'apprentissage par évolution, mais il rencontre le même problème de dimension (Ponsen, 2004). L'optimisation par évolution en simulant des combats peut être adaptée à tout modèle de micromanagement, et l'a été dans un bot (Othman *et al.*, 2012).

Enfin, des approches basées sur des recherches dans l'arbre de jeu ont été explorées plus récemment pour la micro. Churchill et al. ont présenté une variante de l'algorithme alpha-bêta capable de prendre en compte des actions simultanées et avec une durée (Churchill *et al.*, 2012). D'autres recherches pour la réaction ont lieu dans le domaine des sciences cognitives : Wintermune et al. ont testé des modèles d'attention identiques à l'humain pour le contrôle réactif (Wintermute *et al.*, 2007). Pour finir, même si la recherche de chemin (*pathfinding*) n'entre pas dans la catégorie de la réaction, on l'inclut dans cette section, puisqu'elle se déroule à très bas niveau et ne fait pas partie non plus du raisonnement tactique ou stratégique. L'algorithme le plus commun est A*, mais il est coûteux en temps et en mémoire, ce qui le rend difficilement satisfaisant dans les RTS. Des comportements considérant le mouvement des unités en groupe plutôt qu'en tant qu'individu, comme le *steering* ou le *flocking* (comportements de déplacement en

société d'individus comme les essaims ou les nuées), peuvent être utilisés comme surcouche du système de calcul de chemin de base du jeu (Ontañón *et al.*, 2013).

2.3 Communauté d'IA pour StarCraft

Une importante communauté de chercheurs s'est réunie autour de la recherche en intelligence artificielle dans StarCraft. Ces étudiants et chercheurs de tous les continents se sont regroupés et contribuent chacun de façon active à la création commune de contenu. Disposant de groupes de discussion et de forums, les contributeurs développent leurs bots, proposent des tutoriels et des problématiques et rédigent des articles scientifiques sur le sujet. Actuellement, on recense plus de 60 bots différents et plus de 70 articles ont été publiés, sans compter les mémoires de maîtrise et les thèses de doctorat (Uriarte, 2016).

En effet, cette communauté est reconnue au sein de la recherche et participe à des conférences renommées, dont la conférence IEEE CIG (*Conference on Computational Intelligence and Games*) et la conférence AIIDE (*Conference on Artificial Intelligence and Interactive Digital Entertainment*) organisée par l'AAAI (*Association for the Advancement of Artificial Intelligence*).

De plus, tout au long de l'année et durant des compétitions organisées lors des conférences, les développeurs font s'affronter leurs bots dans des parties et des tournois. Certaines compétitions comme celles de CIG ou de l'AIIDE offrent parfois des récompenses et des prix aux meilleurs et permettent aux chercheurs de tester les capacités de leurs IA. Il existe aussi un tournoi permanent nommé SSCAIT [1] (*Student StarCraft AI Tournament*) qui permet à tous de participer et même de regarder en ligne et en direct les rencontres, et propose un classement et des statistiques.

---

1. `http://sscaitournament.com`

2.3.1 Interface de programmation

Depuis la sortie de la première version de l'interface de programmation BWAPI (*BroodWar Application Programming Interface*) (Heinermann, 2009), il est devenu possible de contrôler StarCraft avec des programmes développés en C++. BWAPI est un framework gratuit et en code ouvert (*open source*) créé spécialement pour permettre aux étudiants, chercheurs, ainsi qu'à toute autre personne intéressée de développer des bots qui jouent au jeu. Actuellement disponible en version 4.1.2 (sortie en juillet 2015), elle offre la possibilité de développer des bots en Java, C# et Python en plus du C++, qui reste le langage le plus utilisé.

BWAPI est utilisée par tous et ne révèle que les éléments de l'espace d'états du jeu visibles par défaut aux programmes d'intelligence artificielle, ce qui signifie qu'un bot aura exactement les mêmes informations que peut obtenir un joueur humain. Les informations des unités dans le « brouillard de guerre » sont rendues inaccessibles, ce qui permet aux développeurs d'écrire des bots qui doivent planifier et opérer dans des conditions à information partielle tout en jouant de façon loyale, sans tricher.

De plus, l'interface désactive également les entrées utilisateur par défaut, ce qui permet d'assurer que personne ne prenne le contrôle et n'interagisse avec le jeu en complément d'un bot en cours de partie. Ces paramètres par défaut peuvent être changés pour des raisons de flexibilité lors du développement, mais sont forcés lors de tournois et compétitions. Notons également que l'interface de programmation (API) permet de debugger les bots ; cette fonctionnalité se révèle très utile, même si elle ralentit beaucoup le jeu. Enfin, il est important de préciser que BWAPI n'est pas une interface de programmation officielle et proposée par l'éditeur du jeu, Blizzard Entertainment. Elle est proposée par une partie tierce. D'un point de vue légal, l'interface viole certaines conditions du contrat de licence de l'utili-

sateur final (CLUF) du jeu StarCraft, mais cette situation est tolérée par Blizzard Entertainment (Heinermann, 2009).

Composée de nombreuses classes et interfaces, l'API permet de contrôler les unités de façon individuelle et de lire tous les aspects pertinents de l'état du jeu. Elle donne aussi la possibilité d'analyser des enregistrements de parties image par image et d'extraire d'autres informations utiles, comme les tendances générales, les ouvertures, les *build-orders* et les stratégies courantes. Grâce à elle, on peut également obtenir les informations complètes sur les types d'unités, les améliorations, les technologies et les armes, etc. Par exemple, on peut citer parmi les plus utilisées les interfaces *Game*, *PlayerInterface* et *UnitInterface*. *Game* est le premier moyen d'obtenir des informations générales sur la partie (nom de la carte, temps de jeu, nombre de joueurs, scores, etc.). *PlayerInterface* représente un joueur unique de la partie et fournit l'accès à toutes ses données (son nom, sa race, son nombre de ressources, etc.). Finalement, *UnitInterface* descend à un niveau plus bas et offre le contrôle et l'accès à une unité de façon individuelle (nombre de points de vie, identifiant, action en cours, etc.).

Pour permettre ce niveau d'accès et de contrôle avec le jeu, BWAPI s'injecte dans StarCraft et lit la mémoire du jeu comme si elle était la sienne. Elle modifie également quelques paramètres de la logique compilée de l'application pour faciliter l'échange entrée/sortie de données entre les deux parties. En effet, on récupère l'adresse des espaces mémoire du jeu et on les convertit grâce à une autre API de Windows pour modifier les permissions d'accès à la mémoire du jeu. Ensuite, on modifie quelques octets pour que les appels dirigent en réalité vers des fonctions de BWAPI. Par exemple, au lieu de « call starcraft_405678 », l'appel devient « call bwapi_onFrame » pour appeler la méthode *onframe* de BWAPI. Au retour de la fonction, le flux retourne à l'exécution normale de StarCraft de façon invisible et transparente pour le jeu.

2.3.2 Analyse de la carte

Développé par Perkins (Perkins, 2010), l'outil BWTA (*BroodWar Terrain Analyzer*) est un complément à BWAPI utilisé par tous les développeurs en intelligence artificielle pour StarCraft. BWTA fournit des fonctionnalités d'analyse de terrain sur les cartes du jeu, incluant le calcul des régions basé sur la fusion de diagrammes de Voronoï et la détection de goulots d'étranglement. Un diagramme de Voronoï est un découpage du plan (pavage) en cellules à partir d'un ensemble discret de points appelés « germes ». Chaque cellule renferme un seul germe, elle forme l'ensemble des points du plan qui sont plus proches de ce germe que de tous les autres germes. La cellule représente en quelque sorte la « zone d'influence » du germe (Wikipédia). Disponible en tant que projet open source, BWTA (`https://code.google.com/archive/p/bwta/`) a été repris et est maintenu par Uriarte. Aujourd'hui, c'est cette version BWTA2 qui est utilisée communément (`https://bitbucket.org/auriarte/bwta2/wiki/Home`).

L'algorithme d'analyse passe par une séquence de huit étapes :

1. Reconnaissance des obstacles sous forme de polygones ;
2. Calcul du diagramme de Voronoï ;
3. Élagage du diagramme de Voronoï ;
4. Identification des régions sous forme de nœuds ;
5. Identification des goulots d'étranglement sous forme de nœuds ;
6. Fusion de régions adjacentes ;
7. Validation des goulots d'étranglement
8. Validation des polygones des régions.

Pour la description de chaque étape, on illustre le résultat à partir d'une carte de StarCraft (figure 2.2).

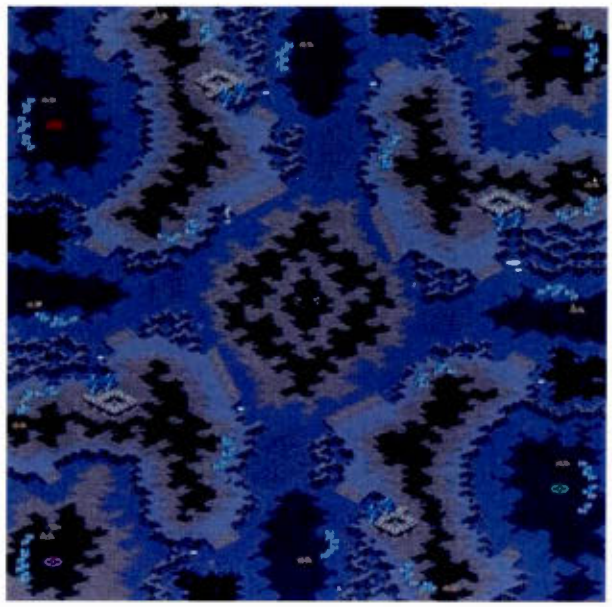

Figure 2.2 La carte Byzantium de StarCraft.

1. Reconnaissance des obstacles sous forme de polygones

Premièrement, l'algorithme convertit le tableau à deux dimensions de la carte en une représentation géométrique des obstacles. Ce procédé s'appelle la vectorisation ; dans notre cas, les pixels sont des valeurs binaires (traversable ou non traversable). Tout d'abord, on applique un remplissage par diffusion (*flood-fill*) sur la carte pour déterminer les composants connectés. Ensuite, l'algorithme trace le contour de chaque composant connecté pour construire une bordure de polygone. Troisièmement, les polygones de composants traversables qui sont inclus dans des polygones de composants non traversables sont représentés comme des trous dans ces polygones, pour que chaque obstacle sur la carte soit représenté comme un polygone avec zéro ou plusieurs trous. Les obstacles qui sont plus petits qu'un certain seuil ne sont pas conservés. Enfin, tous les obstacles sont simplifiés pour réduire le nombre de sommets de chaque polygone.

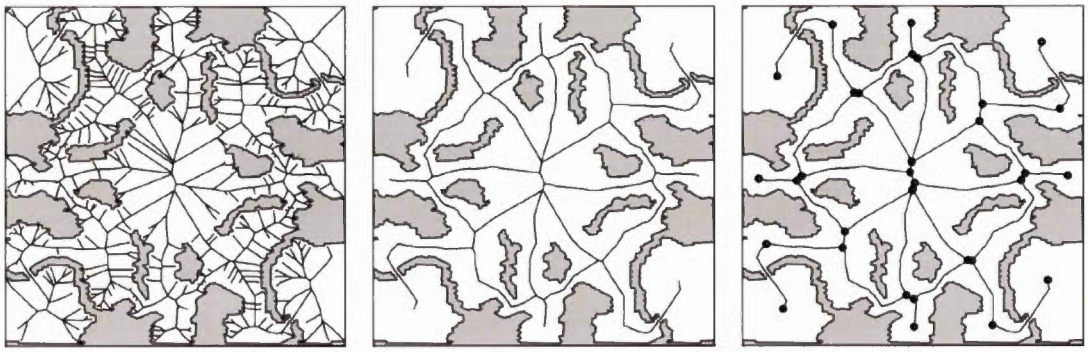

Figure 2.3 Les étapes 1 à 4. (A) Calcul du diagramme de Voronoï ; (B) Élagage du diagramme de Voronoï ; (C) Identification des régions sous forme de nœuds. Source : (Perkins, 2010).

2. Calcul du diagramme de Voronoï

Lors de cette étape, on calcule le diagramme de Voronoï à partir des arêtes des polygones des obstacles. Le calcul est réalisé avec une implémentation de calcul de graphe de Delaunay (*2D Segment Delaunay Graph*) de la librairie GCAL (*Computational Geometry Algorithms Library*).

3. Élagage du diagramme de Voronoï

La troisième étape permet de retrancher les portions inutiles du diagramme de Voronoï. On commence par calculer pour chaque sommet du graphe la distance (radius) entre ce sommet et l'obstacle le plus proche ou la bordure de la carte. Ensuite, l'algorithme itère sur chaque point de terminaison du graphe, et si le radius du sommet est plus petit que celui de son parent dans le graphe, le sommet et l'arête adjacente sont supprimés. La suppression de l'arête peut transformer le sommet adjacent en point de terminaison du graphe ; ainsi le procédé est répété jusqu'à ce qu'aucun autre sommet ou arête ne puisse être supprimé. Cela résulte

en un graphe dont les points de terminaison restants ont un radius plus grand que leurs parents. Enfin, l'algorithme supprime les sommets isolés dont le radius est plus petit qu'un certain seuil. Cette étape réduit de beaucoup la complexité du diagramme de Voronoï, tout en représentant fidèlement la structure de la carte.

4. Identification des régions sous forme de nœuds

Quatrièmement, on commence le processus de partitionnement de la carte en régions. Pour cela on marque tous les nœuds de degré autre que 2 comme nœuds région, parce qu'ils représentent des zones importantes dans la structure de la carte, comme des intersections, des points de terminaison et des composants isolés. Les nœuds de degré 2 sont marqués comme nœuds région seulement s'ils représentent un maximum local et ont un certain radius. Un nœud A est un maximum local si, pour tout autre nœud B dans le radius de A, le radius de B est plus petit que celui de A. Dans la figure 2.3(C), les nœuds région sont illustrés par les points noirs.

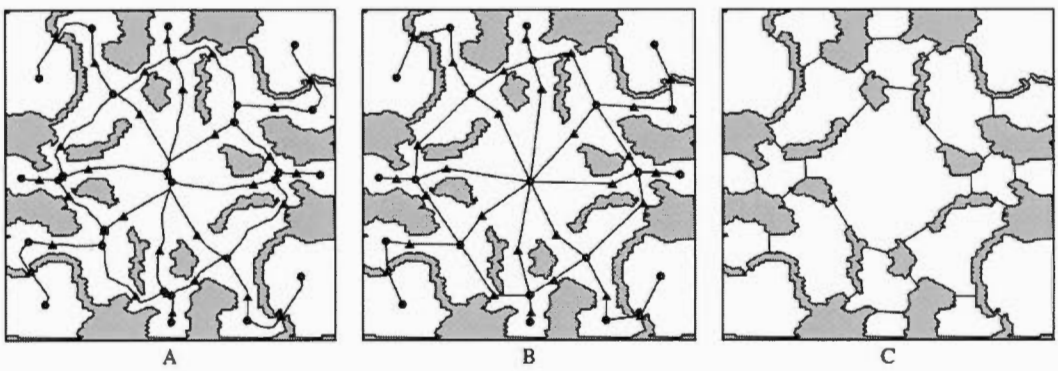

Figure 2.4 Les étapes 5 à 8. (A) Identification des goulots d'étranglement sous forme de nœuds; (B) Fusion de régions adjacentes; (C) Résultat final. Source : (Perkins, 2010).

5. Identification des goulots d'étranglement sous forme de nœuds

Comme tous les sommets de degré différent de 2 sont maintenant des nœuds région, tous les sommets non marqués doivent avoir un degré de 2 et être situés le long d'un certain chemin qui relie deux nœuds région. Ainsi, la cinquième étape suit chaque chemin, trouve le sommet avec le plus petit radius, le marque en tant que goulot d'étranglement et le lie à la paire de nœuds région qu'il connecte. Il est possible qu'un sommet soit à la fois nœud région et goulot d'étranglement. Dans la figure 2.4(A) les goulots d'étranglement sont illustrés par les triangles noirs.

6. Fusion de régions adjacentes

Si l'algorithme passait directement de l'étape 5 à 7, on partitionnerait la carte en un nombre plus important de régions que nécessaire et on placerait des goulots d'étranglement aux mauvais endroits. De ce fait, l'étape 6 fusionne des régions adjacentes et supprime des goulots non pertinents. Deux régions adjacentes sont fusionnées si une des conditions suivantes est vérifiée. Tout d'abord, si le radius du goulot qui les lie est plus grand que 90% du radius du nœud région le plus petit des deux, ou plus grand que 85% du radius du plus grand nœud région. La deuxième condition s'applique spécifiquement dans le cas où une des régions a exactement deux goulots d'étranglement. Pour une région avec deux goulots, elle est fusionnée avec la région adjacente qui est connectée au plus grand des deux goulots seulement si le radius du goulot le plus grand est plus grand que 70% du radius de la région originale. Dans la figure 2.4(B), le diagramme de Voronoï est remplacé par des segments pour lier les nœuds région aux goulots.

7. Validation des goulots d'étranglement

Une fois que les nœuds région et nœuds goulots sont identifiés, il faut calculer la forme exacte du polygone de chaque région et les deux côtés des goulots d'étran-

glement. Pour réaliser cette étape, l'algorithme itère sur chaque goulot et regarde l'ensemble des points les plus proches sur les polygones des obstacles voisins. L'implémentation du calcul de graphe de Delaunay (voir étape 2) facilite cette étape en trouvant les paires de segments d'obstacles qui définissent chaque arête dans le diagramme de Voronoï. Une fois que les deux points terminaux sont trouvés, un segment qui les connecte est ajouté.

8. Validation des polygones des régions

Finalement, après avoir identifié les goulots d'étranglement, chaque nœud région est dans une face unique dans l'arrangement des obstacles et goulots. L'algorithme cherche donc la face correspondante pour chaque nœud région et itère sur les arêtes pour déterminer la bordure du polygone de la région. Le résultat final est montré à la figure 2.4(C), où chaque polygone blanc est une région et où les lignes noires entre deux régions adjacentes représentent les goulots d'étranglement.

Ainsi, BWTA2 offre l'accès aux régions et goulots d'étranglement de la même façon que le fait BWAPI par le moyen de ses interfaces. BWTA2 apporte aussi une classe *BaseLocation*, qui fournit des informations supplémentaires sur les positions potentielles pour des bases et des fonctions utilitaires (calcul de distances, calcul de chemin, etc.). Au final, on peut afficher certaines de ces informations visuellement dans le jeu (figure 2.5).

2.3.3 UAlbertaBot : étude d'un bot

UAlbertaBot est un bot pour StarCraft développé à l'université d'Alberta par David Churchill (Churchill, 2016). Le bot s'est régulièrement distingué aux compétitions internationales majeures d'intelligence artificielle dans StarCraft. Il s'est entre autres classé au premier rang lors de la compétition de l'AIIDE 2013. Le design et les objectifs du projet ont évolué au fil du temps en un système robuste

Figure 2.5 Exemple d'affichage grâce à BWTA. Les contours verts départagent les régions des zones non traversables, la ligne rouge démarque un goulot d'étranglement, les cercles bleu cyan indiquent les ressources et le rectangle bleu une position possible pour une base. (`https://code.google.com/archive/p/bwta/`)

et modulaire capable de jouer les trois races du jeu. UAlbertaBot est disponible en code ouvert. Son concepteur encourage la communauté à s'en servir comme base pour leurs propres travaux. En fait, ce bot a servi de base à de nombreux bots très performants (dont LetaBot, qui a gagné les compétitions SSCAIT 2014 et 2015). De plus, le bot est également utilisé comme outil éducatif dans le cadre d'un cours donné à l'université d'Alberta (*CMPUT350 Advanced Games and AI Programming course*). Il sera présenté plus en détail au chapitre 3, car on a également utilisé ce bot dans le cadre du présent mémoire.

UAlbertaBot est écrit en C++ et utilise BWAPI. Il a été pensé avec un design hiérarchique et modulaire similaire à une structure de commandement militaire.

Cette architecture est bénéfique pour plusieurs raisons. Le nom des classes est intuitif, ce qui facilite l'interprétation, par exemple la classe *ProductionManager* prend en charge toutes les tâches liées à la production de bâtiments et d'unités. Le design modulaire permet de faire des modifications et des mises à jour rapidement et facilement. La structure et la communication verticales entre les entités réduisent la complexité et les duplications de codes. Les informations au sein des classes sont basées sur le principe « doit-savoir ». Par exemple, seule la classe *WorkerManager* sait quelles unités sont assignées à quelles tâches. La hiérarchie des classes est présentée à la figure 2.6.

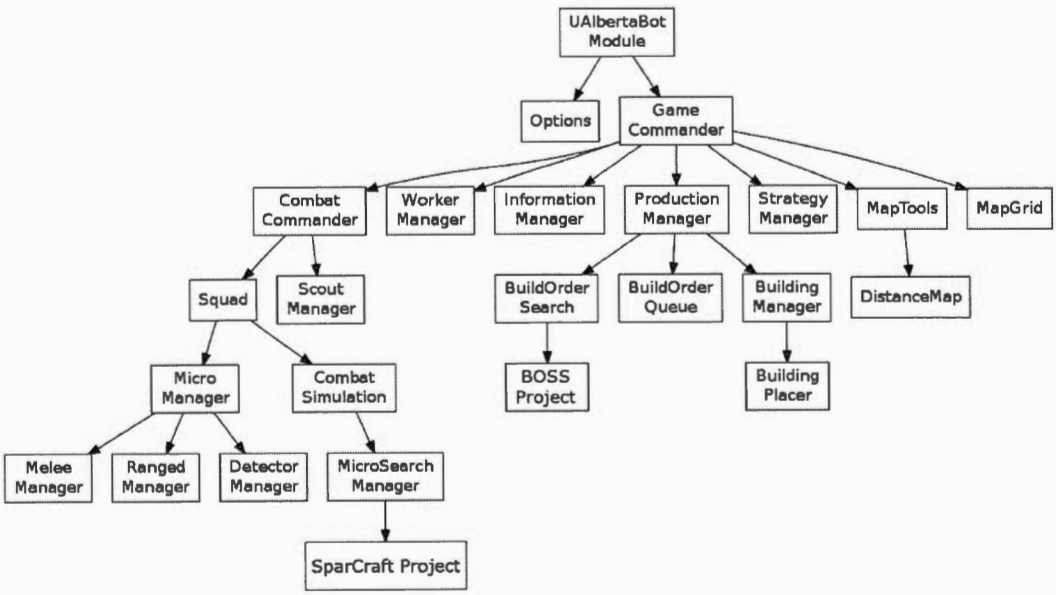

Figure 2.6 Hiérarchie des classes de UAlbertaBot. Contrairement à un diagramme UML, ici, une flèche indique une relation de responsabilité (Source : (Churchill, 2016)).

Il y a quelques cas de communication globale au sein du bot ; ils sont implémentés via des classes *singleton* (patron de conception). Les tâches coûteuses en calculs, comme la recherche de chemin, sont faites dans ces classes et les résultats sont

disponibles globalement. Il y a quatre classes de ce genre. La classe *Information-Manager* calcule et contient des informations importantes à propos des unités (par exemple le nombre de pertes de l'adversaire). *MapTools* fait du calcul de chemin avec distances au sol et met en cache les résultats pour une utilisation ultérieure. *MapGrid* fait des calculs de voisins les plus proches et contient des informations relatives à la carte (sous forme de grille), comme la dernière fois qu'une zone de la carte a été visitée. *StrategyManager* prend des décisions à propos des *build-orders*, du moment auquel s'étendre sur la carte, et d'autres politiques de jeu.

Au démarrage d'une partie, le code commence par lire un fichier de configuration qui définit quelques paramètres et permet des changements de configuration rapides sans modifier le code. Après le démarrage du bot, BWAPI interagit avec StarCraft et appelle sa fonction *OnFrame()* après chaque image du jeu. Cette fonction peut être vue comme une boucle standard dans laquelle toute la logique d'UAlbertaBot est exécutée. La méthode *OnFrame()* d'UAlbertaBot appelle *GameCommander.update()*. Le « travailleur » est l'unité de base du jeu, qui permet de récolter des ressources et construire des bâtiments. Le flux logique complet est détaillé à la figure 2.7.

UAlbertaBot contient un mélange de systèmes dynamiques d'intelligence artificielle et de règles de prise de décision prédéfinies (scriptées). Pour allouer des travailleurs à la collecte de ressources, le système utilise des règles prédéfinies basées sur des connaissances expertes acquises de joueurs professionnels de StarCraft. Le micromanagement est également basé sur des politiques prédéfinies pour chaque type d'unités (les combattants de base suivent des règles différentes de celles des véhicules de combat et de transport, etc.). De plus, un algorithme de remplissage par diffusion est employé pour calculer les plus courts chemins au sol vers une certaine position de la carte. Ces résultats sont ensuite conservés pour réutilisation ultérieure (et ne pas avoir à les recalculer) (Churchill, 2016).

Pour tous les combats, UAlbertaBot simule au préalable la bataille grâce à SparCraft, un système de simulation de combat du même auteur (Churchill *et al.*, 2012). Si la prédiction du système est positive, l'IA engage le combat, sinon l'ordre de repli est donné. SparCraft inclut plusieurs algorithmes développés pour le combat dans les jeux de stratégie en temps réel. Une version modifiée de l'algorithme alpha-bêta qui prend en compte les durées (voir ABCD dans la section 2.2.2), un algorithme UCT (méthodes de Monte-Carlo) modifié pour être utilisé avec les RTS et une technique de recherche gloutonne qui est plus performante que les deux précédentes lors de larges affrontements, mais moins efficace pour les petits combats (Churchill et Buro, 2013).

Enfin, tous les *build-orders* du bot sont planifiés par un système de planification de *build-order* basé sur une recherche par heuristique, nommé BOSS (*Build Order Search System*). Les objectifs pour la construction de bâtiments et d'unités sont donnés par *StrategyManager* au *ProductionManager*, qui utilise ensuite BOSS pour générer dynamiquement des *build-orders* quasi optimaux comparables à des joueurs professionnels de StarCraft. BOSS utilise un algorithme de séparation et évaluation avec parcours en profondeur pour la recherche (Churchill et Buro, 2011).

GameCommander.update()
- WorkerManager.update()
  - Ré-assigner les travailleurs à de nouveaux minerais si les anciens sont terminés
  - Envoyer des travailleurs inactifs récolter des minerais
  - Allouer des travailleurs jusqu'à 3 par raffinerie
  - Déplacer les travailleurs vers la position de construction si nécessaire
- ProductionManager.update()
  - Si un événement actionne un nouveau *build-order*
    - Vider le *build-order* courant
    - Obtenir un nouveau *build-order* de *StrategyManager*
    - Enregistrer le nouveau *build-order* de *StarCraftBuildOrderSearch*
  - Produire l'élément à la plus haute priorité dans le *build-order* si possible
  - Si l'élément à produire est un bâtiment, ajouter la tâche au *BuildingManager*
- BuildingManager.update()
  - Vérifier si les travailleurs assignés à un bâtiment sont morts avant d'avoir fini
  - Trouver une position pour le bâtiment en cherchant en spirale depuis la position désirée
  - Assigner les travailleurs aux bâtiments non assignés et les marquer comme planifiés
  - Pour chaque bâtiment planifié, dire au travailleur assigné de le construire
  - Suivre en continu le statut des bâtiments en construction
  - Si on est *Terran* et que le travailleur est mort en construisant, en assigner un autre
  - Si un bâtiment se termine, supprimer la tâche et marquer le travailleur inactif
- CombatCommander.update()
  - Envoyer un travailleur faire du *scouting* si c'est le bon moment
  - Si des ennemis sont près de notre base, y assigner un groupe de défense
  - Si d'autres unités sont disponibles pour attaquer, attaquer comme suit :
    - Si une base ennemie est visible, l'attaquer
    - Sinon, si on voit des ennemis, les attaquer
    - Sinon, si on connait la position d'un bâtiment ennemi, l'attaquer
    - Sinon, explorer la région de la carte vue la moins récemment
  - Squads.update()
    - Si la force de défense a tué les ennemis, ajouter les défenseurs au groupe d'attaque
    - Faire une simulation de combat avec SparCraft pour le groupe d'attaque
    - Si la simulation prédit la victoire, continuer d'attaquer
    - Si l'attaque continue, appeler la sous-classe *MicroManager* pour chaque unité
    - Si la simulation prédit la défaite, se replier vers la base
    - Si le groupe ne contient pas d'unité, le supprimer
- ScoutManager.update()
  - Si la position de la base ennemie est connue, y aller et continuer d'observer
  - Si le scout n'est pas attaqué, attaquer le travailleur le plus proche, sinon se replier
  - Si on ne peut pas voir la base ennemie, explorer l'emplacement possible de base le plus proche
- InformationManager.update()
  - Si une unité ennemie est visible, enregistrer sa dernière position connue
  - Si une unité ennemie meurt, ajouter la perte en ressource au total perdu

Figure 2.7 Flux logique complet d'UAlbertaBot (Churchill, 2016).

CHAPITRE III

EMPLOI DU RAISONNEMENT SPATIAL DANS LE JEU STARCRAFT

Dans ce chapitre, on s'intéresse de plus près à un des challenges identifiés par la communauté de recherche en IA pour StarCraft, soit le raisonnement spatial. Le raisonnement spatial consiste à prendre connaissance de chaque aspect de l'environnement ou du terrain afin de pouvoir l'exploiter pour prendre des décisions. Dans StarCraft, il est lié à chaque aspect de l'exploitation du terrain, comme le choix stratégique de l'emplacement des bâtiments, le placement des unités pour les batailles, la prédiction de la position d'un ennemi basée sur peu d'informations, etc. Le raisonnement spatial mène aussi à choisir des emplacements où faire du *scouting* et explorer, des endroits pour attaquer ou encore pour placer des troupes, ainsi qu'à faire de la planification de chemin. Il permet d'identifier les impasses et les goulots d'étranglement, les points stratégiques et les zones dangereuses, de gérer l'aménagement de la base, etc.

Dans les pages qui suivent, on commence par faire l'état de l'art du raisonnement spatial appliqué à StarCraft (section 3.1), dans lequel on observe des approches de différents niveaux et où on constate des limites. Ensuite, on propose une nouvelle direction pour la recherche dans le domaine, en présentant la motivation et les objectifs de ce mémoire (section 3.2).

3.1 État de l'art du raisonnement spatial appliqué à StarCraft

Dans la littérature, on emploie aussi les termes « analyse spatiale » ou « analyse de terrain » pour désigner le raisonnement spatial, même si l'analyse de terrain en est davantage une composante (récupération des informations) qu'un équivalent (récupération et utilisation des informations). Cependant, on comprend que ce concept est un élément central et est à la base d'une intelligence artificielle robuste. En effet, le raisonnement spatial est lié à la représentation des connaissances. Or, une bonne représentation des connaissances combinée à l'accès à un maximum d'informations permet ensuite un meilleur travail de raisonnement par rapport au contexte.

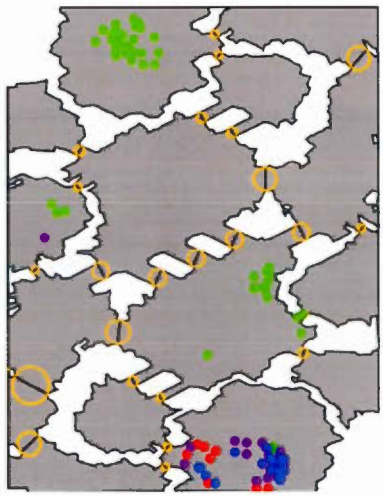

Figure 3.1 La perception d'un bot de StarCraft inclut la position de ses unités (en vert), des estimations de position des ennemis (en bleu), les régions (en gris), les goulots d'étranglement (entourés en orange). La position réelle des ennemis (en rouge) n'est pas visible par le bot. Source : (Weber *et al.*, 2011b).

Beaucoup d'approches présentes dans la littérature se concentrent sur le positionnement des unités (formations, micromanagement). D'autres tentent de reproduire

des comportements observés chez les humains (*kiting*, *walling*). Dans les bots développés avec BWAPI, l'analyse de terrain est faite par un module développé par Perkins, qui utilise des diagrammes de Voronoï pour identifier des régions et des goulots d'étranglement (vu à la section 2.3.2). Tel qu'observé à la section 2.2, on peut constater que la recherche en raisonnement spatial dans StarCraft se fait à plusieurs niveaux d'approches : une approche *tactique*, qui représente le micromanagement et englobe les décisions à court terme, comme le placement des unités pour les batailles ou la recherche de chemin, et une approche plus *stratégique*, le macromanagement, pour des décisions à plus long terme, comme le placement des bâtiments et l'aménagement de la base, ainsi que la gestion de l'économie.

3.1.1 Approche tactique : le micromanagement

Dans leur article *Pattern Formation Based on Potential Field in Real-Time Strategy Games* (Cheng-Yu Chen et Ting, 2012), les auteurs proposent de générer des modèles de formation de combat pour des groupes d'unités en se basant sur des champs de potentiel. Leur méthode contrôle le mouvement des unités et ajuste leur formation en considérant le nombre et le type d'ennemis. De façon plus spécifique, une fois un ennemi détecté, l'objectif est de synchroniser le déplacement des unités individuelles pour encercler l'adversaire, comme à la figure 3.2.

En 2011, Gonzalez et Garrido présentent une méthode pour contrôler les unités, basée sur une intelligence distribuée dans le but d'obtenir de meilleures performances que l'intelligence artificielle de base du jeu (figure 3.3). Cette méthode propose de former les unités de façon à ce qu'elles puissent atteindre une bonne position pour attaquer un groupe d'ennemis du même type. Leurs résultats montrent qu'il est possible d'implémenter une telle technique et d'obtenir une meilleure performance que l'algorithme glouton de l'IA de base de StarCraft. De plus, la nature de la technique la rend extensible, décentralisée et robuste.

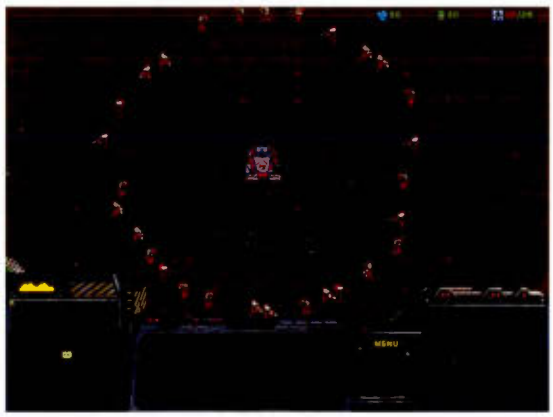

Figure 3.2 Placement des unités pour encercler l'adversaire pendant le combat avec des champs de potentiel. Source : (Cheng-Yu Chen et Ting, 2012).

Figure 3.3 Placement des unités pour le combat avec des comportements de type essaim. Source : (Gonzalez et Garrido, 2011).

D'autre part, les aspects hautement dynamiques et le déroulement en temps réel des RTS font de la planification de chemin, élément crucial pour le déplacement des unités, une tâche difficile pour les bots. Habituellement, ce problème est traité avec des algorithmes comme A*, qui sans adaptations ne fonctionnent pas optimalement dans des environnements dynamiques. Ainsi, une combinaison de l'algorithme A* avec des champs de potentiel a été étudiée pour mieux gérer les aspects dynamiques de StarCraft. Elle affiche des résultats plus intéressants qu'avec une utilisation de A* uniquement (figure 3.4).

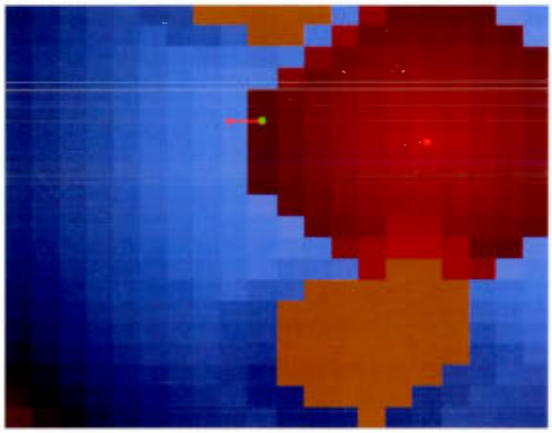

Figure 3.4 Le champ de potentiel pour une unité (point vert) dans un état de retraite stratégique face à un ennemi (point rouge), la zone rouge est répulsive. Source : (Hagelback, 2012).

De plus, des chercheurs de l'université du Nevada (Siming Liu et Nicolescu, 2013) ont comparé des algorithmes génétiques avec des algorithmes d'escalade (*hill-climbing*) pour le micromanagement. Ils utilisent des cartes d'influence pour générer le positionnement en groupe des unités et des champs de potentiel pour guider leur déplacement (figure 3.5). Les comportements obtenus avec un algorithme génétique et deux techniques de *hill-climbing* sont ensuite testés contre l'intelligence artificielle de base du jeu. Leurs résultats indiquent que les algorithmes de *hill-climbing* apportent rapidement des comportements de qualité de 50% à 70% du temps, alors que les algorithmes génétiques produisent des comportements de qualité plus de 70% du temps, mais sont beaucoup plus lents.

Enfin, on peut noter que de nombreuses autres techniques ont été étudiées pour le micromanagment dans StarCraft, comme des algorithmes MCTS, pour déterminer des séquences d'actions optimales pour des unités en combat, ou des algorithmes génétiques pour obtenir des formations tactiques émergentes. En plus des

algorithmes génétiques et *hill-climbers* cités pour le positionnement et le déplacement des unités, d'autres chercheurs ont aussi testé des algorithmes génétiques à cas informés pour le même problème. Deux articles importants dans le domaine recensent ces travaux ainsi que d'autres pour le micromanagment, à base de techniques de Monte-Carlo ou de réseaux de neurones (Ontañón *et al.*, 2013; Robertson et Watson, 2014).

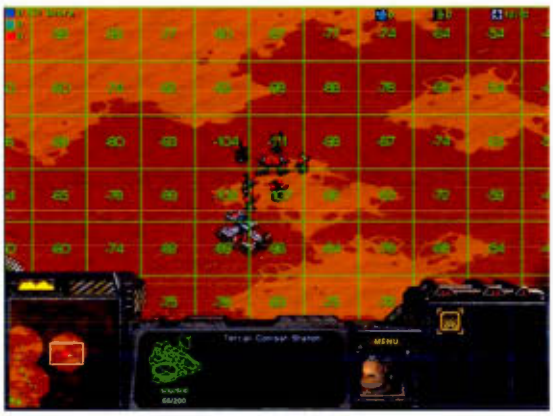

Figure 3.5 Une carte d'influence implémentée dans StarCraft pour le positionnement et déplacement des unités. Source : (Siming Liu et Nicolescu, 2013).

3.1.2 Approche stratégique : le macromanagement

Dans leur article *A Particle Model for State Estimation in Real-Time Strategy Games* (Weber *et al.*, 2011a), les auteurs explorent une approche avec modèle de particules pour tenter de prédire la position des ennemis dans StarCraft, en se basant sur leur trajectoire et les goulots d'étranglement proches au moment où ils ont été aperçus. Ils utilisent une particule unique pour chaque unité plutôt qu'un nuage de particules, parce qu'il n'est pas possible de différencier deux unités du même type, et donc qu'il aurait été difficile de mettre à jour le nuage de particules dans les cas où des unités sont perdues (hors du champ de vision) puis retrouvées.

Pour contourner le problème de différence entre les types d'unités, ces chercheurs les divisent en classes et apprennent un modèle de déplacement pour chacune à partir d'enregistrements de joueurs professionnels sur une variété de cartes de StarCraft. Cette technique est implémentée dans leur bot (EISBot), ce qui leur permet de prédire la position des ennemis après les avoir vus et de lever des conditions qui déclenchent des comportements tactiques au sein du bot. D'après les résultats, ce système a permis d'augmenter les performances d'EISBot de 10%.

Le *kiting* (technique qui consiste à attaquer un ennemi pour ensuite se mettre immédiatement hors de sa portée, voir section 2.1.2) fait partie d'un des objectifs des chercheurs tentant la re-création de comportements humains par les bots dans StarCraft. Alors que les cartes d'influence sont utilisées avec succès pour contrôler le déplacement d'unités, Uriarte et Ontañón (2012) proposent de les utiliser également pour la simulation du *kiting* dans le jeu. Pour ce faire, ils divisent le problème en trois sous-problèmes : (1) décider quand faire du *kiting* ; (2) créer une carte d'influence pour guider le déplacement des unités et (3) sélectionner une cible. Leur solution finale est viable en termes de calcul pour l'ordinateur et utilisable par des bots en temps réel ; elle est d'ailleurs implémentée dans leur bot NOVA (Uriarte et Ontañón, 2012).

Toujours dans le but de reproduire des comportements de joueurs, Certický (2013) s'intéresse au problème du *walling*. Cette technique est employée par les joueurs pour bloquer l'entrée de leur base et survivre aux agressions de leurs adversaires en début de partie. Il résout le problème de trouver le placement approprié des bâtiments pour fermer le goulot d'étranglement à l'entrée de la base du joueur en utilisant de la programmation par contraintes. Il commence par faire une liste des bâtiments utilisables (avec leur taille et l'espace requis), puis il utilise BWAPI pour récupérer des informations sur le terrain. Il teste ensuite des combinaisons et vérifie s'il existe encore un passage, jusqu'à trouver une solution avec un solveur de

programmation par contrainte. Généralement, un petit exécutable externe effectue ces calculs en début de partie, juste après l'analyse de la carte de jeu. En sortie, on récupère le type des bâtiments à construire ainsi que leur position.

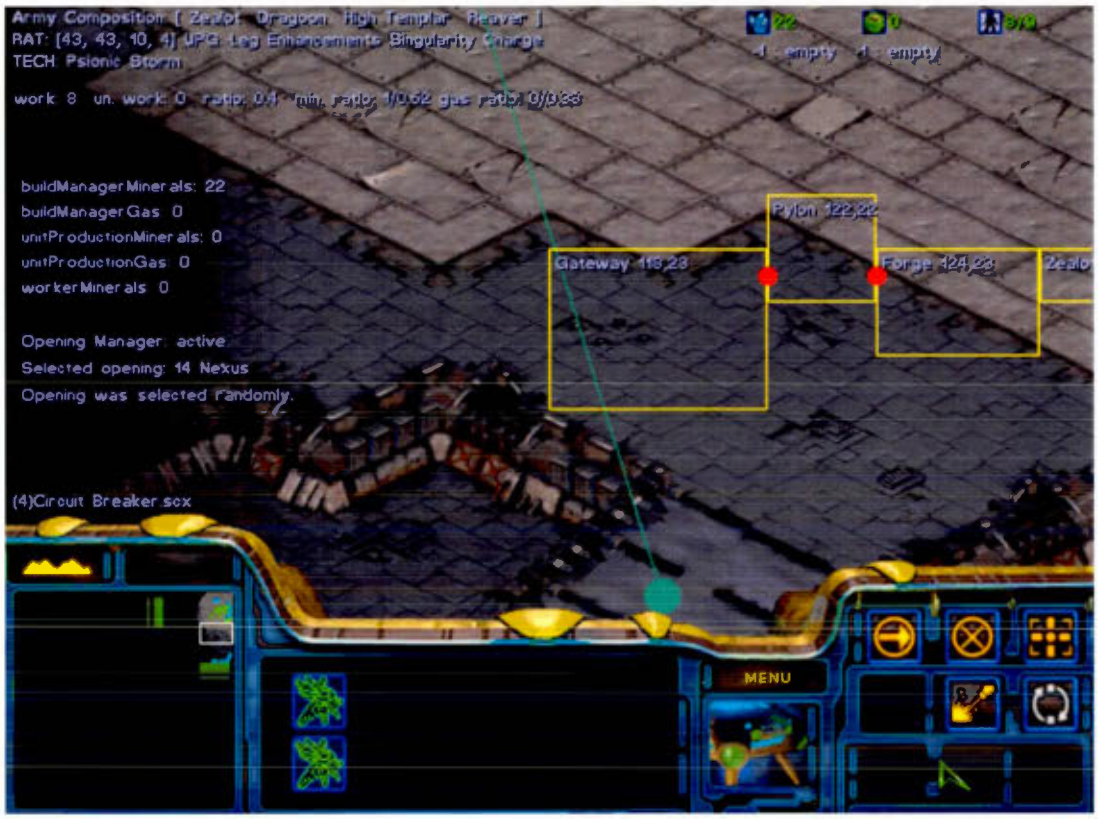

Figure 3.6 Un exemple de *walling* : on peut voir en jaune la position recommandée pour les bâtiments ainsi que leur type, pour bloquer l'entrée en bas à droite. Source : (Certický, 2013).

3.1.3 Limites et constat

Malgré l'intérêt porté au raisonnement spatial par la communauté de recherche et le travail déjà accompli, beaucoup d'autres idées restent à explorer dans le domaine, d'autant plus que les techniques actuelles sont encore imparfaites. Ef-

fectivement, si les champs de potentiel et les cartes d'influence offrent des performances intéressantes, ils ne sont pas sans inconvénients. Ces techniques souffrent, par exemple, de problèmes de type optimum local et demandent beaucoup de ressources (Hagelbäck et Johansson, 2008). De plus, ces méthodes de reconnaissance de terrain ont besoin de beaucoup de paramétrage et de réglages de précison avant d'être performantes pour une partie de StarCraft (Hagelbäck, 2009). D'autre part, à l'heure actuelle, les systèmes d'intelligence artificielle pour les RTS ne parviennent à reproduire que des tactiques simples de joueurs expérimentés, alors qu'on souhaite pouvoir générer des plans complexes pour approcher de plus en plus du niveau des professionnels (Uriarte et Ontañón, 2012). Par exemple, on pourrait vouloir faire du *kiting* et attirer l'adversaire dans une embuscade planifiée grâce au raisonnement spatial, ou exploiter encore davantage le terrain.

Après avoir exploré la littérature sur le raisonnement spatial dans les RTS et plus spécifiquement dans StarCraft, plusieurs constats peuvent être faits. En effet, on remarque que beaucoup plus d'attention a été portée sur le micromanagement et la gestion de bas niveau des unités que sur le macromanagement et la stratégie à plus haute échelle. En réalité, la majorité des travaux concernent le micromanagement, une petite partie la tactique de niveau intermédiaire, mais au meilleur de nos connaissances, on ne trouve pas d'étude au niveau supérieur représenté par la stratégie.

C'est pour cette raison qu'on a choisi de travailler autour du raisonnement spatial dans les jeux de stratégie en temps réel. Dans un deuxième temps, on a choisi le jeu StarCraft parce qu'une communauté de recherche importante et active existe déjà autour du jeu et pour son statut de jeu commercial compétitif très répandu. De plus, tel que mentionné à la section 2.3.1, une interface de programmation (BWAPI) est disponible et maintenue, et des compétitions de bots sont organisées annuellement lors des conférences CIG, AIIDE, etc.

3.2 Motivation et objectifs du mémoire

La motivation principale de ce travail est d'utiliser le raisonnement spatial à un niveau où il n'a *a priori* pas encore été utilisé, un niveau plus élevé que la réaction ou la tactique à court terme, la prise de décision stratégique. L'idée est de définir des stratégies en fonction du raisonnement spatial et du terrain ou de baser la stratégie sur le raisonnement spatial. Le but est de donner la priorité, dans la stratégie, au raisonnement spatial afin de prouver qu'il est pertinent et possible de définir dans les RTS des stratégies en fonction de celui-ci, qui sont capables de reproduire des comportements humains avancés (embuscades, barrages, etc.).

On se fixe par ce choix l'objectif d'améliorer la capacité d'adaptation d'un bot en fonction des cartes de jeu, en comparaison d'autres bots établis. Cela afin d'augmenter la polyvalence de ce type de bot vis-à-vis du contexte dans lequel il évolue, pour finalement offrir plus de diversité et une meilleure expérience de jeu à ses adversaires. Cette approche permettra de découvrir ce qu'apporte le fait de mettre le raisonnement spatial au centre du système et de la stratégie par rapport aux autres solutions existantes. En d'autres termes, on souhaite sortir des conventions habituelles de la figure 3.7 pour augmenter les privilèges et l'importance accordés au raisonnement spatial, pour que sur la figure, il puisse se hisser au niveau de la stratégie.

À la figure 3.7, l'incertitude (qui provient des informations partielles et du fait qu'on ne connait pas les intentions de l'adversaire) est plus grande pour les niveaux d'abstraction les plus hauts. Les temps sur la droite correspondent à des estimations de durée entre deux changements de comportement dans StarCraft. Le raisonnement spatial et le raisonnement temporel sont indiqués pour les niveaux auxquels des solutions « gourmandes » ne suffisent pas.

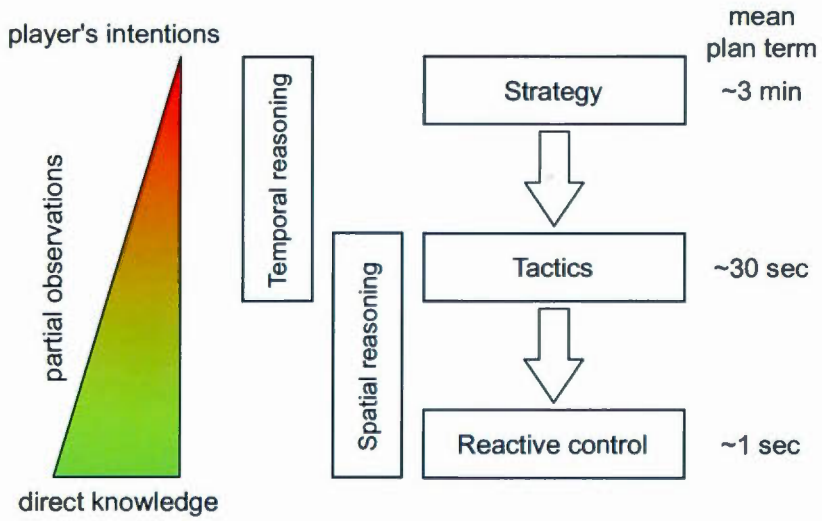

Figure 3.7 Les niveaux d'abstraction classiques de l'IA d'un bot de StarCraft. Source : (Ontañón *et al.*, 2013).

CHAPITRE IV

INTÉGRATION DU RAISONNEMENT SPATIAL AU CŒUR DE LA STRATÉGIE ET DE LA PRISE DE DÉCISIONS

Ce chapitre présente notre approche pour employer le raisonnement spatial à un niveau stratégique et lors de la prise de décision, dans le but d'augmenter la polyvalence des bots de RTS. Pour ce faire, on s'engage dans le développement d'un bot nommé « SRbotOne » (*Spatial Reasoning bot version One*) pour servir de banc d'essai aux idées qu'on souhaite développer. Afin de ne pas ré-inventer la roue, on prend l'initiative de ré-employer la base robuste et open source, déjà étudiée en détail, d'UAlbertaBot.

Dans la section 4.1, on présente le design et les idées de conception de SRbotOne dans lesquelles on définit une stratégie globale en concordance avec l'emploi du raisonnement spatial. La section 4.2 est consacrée aux détails de développement du bot et aux différents systèmes qui y sont ajoutés, avant de se terminer par une comparaison avec UAlbertaBot.

4.1 SRbotOne : Design et stratégie

Puisque l'on souhaite concevoir SRbotOne autour d'une philosophie globale basée sur le raisonnement spatial, on doit penser son design ainsi que sa stratégie en accord avec les objectifs fixés. Pour la stratégie globale d'un bot, un choix doit

être fait dans un premier temps au plus haut niveau de décision possible : attaquer ou défendre ? La grande majorité des bots développés pour StarCraft font le choix de l'attaque, celle-ci est tout à fait légitime.

En réalité, elle représente l'objectif principal, ce pourquoi on joue dans la plupart des RTS, étant donné que le but du jeu est de se développer, créer une armée pour conquérir le monde ou dominer l'adversaire (voire l'anéantir), qui sont les conditions de victoire. C'est le cas dans StarCraft aussi, et si on s'intéresse de façon plus précise au contexte des bots développés pour la compétition dans le jeu, l'attaque est prisée car on peut la considérer plus simple à mettre en place. En effet, un des plus grands défis dans la recherche en IA dans StarCraft est de faire durer la partie longtemps. À mesure que la partie progresse, le nombre d'unités ainsi que les actions possibles augmente, ce qui impacte le facteur de branchement et conduit à une explosion combinatoire. Or, tout prendre en compte complexifie beaucoup le développement. Ainsi, plus on termine la partie rapidement, plus on peut considérer le développement simple, car on a beaucoup moins de cas et scénarios à traiter. Et pour ce faire, on décide d'attaquer (le plus tôt possible) parce que c'est le moyen le plus rapide de finir une partie. Enfin, on constate également que l'attaque est poussée par les stratégies de *rush* qui sont très efficaces en compétition.

Après l'étude approfondie du domaine qu'on a pu faire, on choisit une stratégie défensive pour SRbotOne. Comme on souhaite mettre en avant le raisonnement spatial, défendre offre plus d'opportunités qu'attaquer. En effet, même si l'attaque peut aussi servir de banc d'essai pour le raisonnement spatial, une IA offensive sera basée plus sur le micromanagement, et on a vu que beaucoup de travail a déjà été effectué dans ce domaine. Or, un de nos objectifs est d'utiliser le raisonnement spatial à plus haut niveau (macromanagement et stratégie globale), d'où le choix de la défense. En plus, partir sur une stratégie défensive permet de se différencier

et d'expérimenter d'autres pistes pour la recherche, comme tenter de faire durer la partie plus longtemps et tester des façons de résister ou contrer des stratégies qu'on sait portées sur l'attaque.

Plus précisément, comme la condition de victoire dans StarCraft reste la destruction complète de l'ennemi, on va tout de même devoir attaquer à un moment. Plutôt qu'une stratégie de défense pure, on doit ainsi dévier vers de la contre-attaque. On choisit de défendre dans un premier temps, en utilisant du raisonnement spatial et de partir en contre au moment opportun. L'intelligence artificielle de SRbotOne est donc développée pour commencer la partie en mode défensif, afin de se protéger et attendre les attaques ennemies, et pour contrer au moment opportun ou passer à l'attaque si l'adversaire n'attaque pas lui non plus.

Une fois la stratégie globale choisie, la prochaine interrogation se situe dans le choix de la race qui y correspond le mieux. Tel que vu dans la section 2.1.1, les races dans StarCraft sont équilibrées et ont des spécificités propres. Pour une stratégie de contre-attaque, il apparaît que les *Terrans* représentent la meilleure option. En effet, ils sont réputés pour être la race du jeu la plus adaptative ce qui est un atout pour défendre et attaquer en même temps. De plus, ils sont aussi caractérisés pour avoir une forte capacité défensive, aussi bien grâce à des bâtiments à vocation défensive que des unités. Par exemple, parmi leurs atouts, on peut mentionner que seuls les *Terrans* sont capables de construire des bunkers ou de produire des unités médicales qui peuvent soigner les autres unités. En plus de pouvoir soigner leurs unités, ils sont aussi les seuls à pouvoir réparer des bâtiments et unités mécaniques. Enfin, un joueur *Terran* peut construire des *tanks* qui ont la particularité de posséder deux modes, un mode normal et un mode de siège dans lequel le *tank* se stabilise au sol, le rendant immobile mais lui faisant gagner en portée d'attaque et en puissance de feu. Grâce à ce mode de siège, cette unité est celle qui profite de la plus grande portée de tir du jeu. Ainsi comme on peut le

voir, les *Terrans* offrent de nombreux avantages pour la stratégie qu'on souhaite mettre en place, voilà pourquoi on les choisit.

4.1.1 Stratégie d'ouverture

L'ouverture d'une partie de StarCraft est cruciale puisqu'elle conditionne toute la suite de la partie. À chaque début de partie, tous les joueurs commencent avec quatre travailleurs et un bâtiment principal. La première chose à faire est de lancer la récolte de ressources. À partir de ce moment, les stratégies se différencient, on peut choisir d'envoyer les quatre travailleurs récolter des ressources, ou en envoyer trois seulement et construire quelque chose avec le dernier, etc. Lorsqu'on raisonne à propos de l'ouverture, deux questions principales se posent : quel *build-order* (séquence de production de bâtiments et d'unités) choisir ? Et quand créer la première extension (deuxième base) ?

Étant donné que les ressources jouent un rôle clé dans StarCraft, on met l'accent sur leur production dans SRbotOne, ceci en tentant de maximiser la capacité de production le plus rapidement possible. Concernant la deuxième question, comme pour la production de ressources, on choisit de s'étendre vers une deuxième base le plus rapidement possible. Cependant, on s'impose la condition de ne pas nuire au taux de production de ressources. C'est-à-dire qu'on ne souhaite pas créer une extension dans le temps théorique le plus rapide, mais la lancer au moment le plus tôt, ce qui permet de maximiser la production. Pour donner un exemple, on pourrait récolter juste assez de ressources pour pouvoir construire une nouvelle base et la construire immédiatement, mais cela impacte beaucoup la capacité de récolte de ressources et donc la réserve de ces dernières. En réalité, il est plus judicieux de produire en premier davantage de travailleurs pour augmenter la productivité et ensuite lancer l'extension. Ceci signifie qu'à un même temps T, ayant construit l'extension au plus vite, on se retrouvera avec moins de ressources en réserve qu'en

ayant produit d'abord des travailleurs supplémentaires puis l'extension. Or, un retard pris en terme de ressources dès le début de la partie est souvent fatal dans StarCraft.

Par chance, le coût ainsi que la durée de production des unités et bâtiments, en plus du taux de récolte de ressources des travailleurs, sont connus pour le jeu. Donc, il est possible de calculer quand créer la première extension tout en maximisant un taux de production de ressources optimal, et d'en déduire par la même occasion le *build-order* à choisir pour remplir les objectifs fixés. Les informations de coût et de durée de production auxquelles on s'intéresse sont présentées dans le tableau 4.1.

Tableau 4.1 Informations de production des unités de début de partie.

Nom	Coût (minerai)	Durée (secondes)
Baraquement	150	80
Centre de Commandement	400	120
Dépôt	100	40
Travailleur	50	20

De plus, on sait qu'un travailleur a un taux de récolte de minerai de 40 par minute. Pour que le calcul soit proche et fidèle à la réalité, plusieurs éléments sont à prendre en compte. Tout d'abord, une notion de *population* existe dans StarCraft. Elle représente en quelque sorte une capacité du nombre d'unités. En jeu, la population qu'un joueur peut avoir est limitée (à 200 au maximum). Chaque unité créée augmente le compteur de population d'une valeur qui dépend de son type (par exemple de 1 pour un travailleur et de 2 pour un *tank*). En début de partie, un joueur commence avec une population de 4 (4 travailleurs) sur 10 (la capacité de population apportée par le *centre de commandement*, bâtiment avec lequel on commence). De ce fait, il existe effectivement des bâtiments qui ont la

particularité d'augmenter la capacité limite de population du joueur, et durant la partie le joueur doit en construire pour augmenter sa capacité et la taille de son armée. Pour les *Terrans*, ces bâtiments sont le *centre de commandement* (+10 sur la capacité de population) et le *dépôt* (+8 sur la capacité de population). Cet élément doit être pris en compte dans le calcul pour l'ouverture.

Deuxièmement, lors du calcul on doit faire attention au nombre de travailleurs disponibles. Ce nombre varie étant donné qu'un travailleur ne peut exécuter qu'une tâche à la fois (récolter des ressources, construire quelque chose, ou effectuer une autre action). Ceci signifie qu'il ne faut pas prendre en compte dans le calcul de production un travailleur qui collectait des ressources mais à qui on a donné un autre ordre : il est occupé pendant la durée de l'autre action et ne récolte rien.

Troisièmement, on doit également prendre en compte et retrancher le coût de production des bâtiments et unités produites dans le calcul. Ceci en gardant la stratégie défensive en tête, d'où l'ajout au *build-order* de la création de bâtiments spécifiques, comme le *baraquement* qui permet de produire des unités pour se défendre en début de partie.

On fait le choix d'augmenter rapidement la capacité de production de ressources, par conséquent la stratégie est de produire des travailleurs supplémentaires de façon constante en début de partie (on ne peut en produire qu'un à la fois). On définit l'équation (4.1) qui permet d'estimer la réserve en ressources qu'on peut avoir toutes les 20 secondes (durée de production d'un travailleur).

$$\text{RÉSERVE}_t = \text{RÉSERVE}_{t-1} - 50 + p - cb \tag{4.1}$$

où p est la production de l'ensemble des travailleurs (qui récoltent) en 20s, et cb est le coût du bâtiment à construire si il y a lieu.

Ainsi, on obtient RÉSERVE_t, 20 secondes après RÉSERVE_{t-1}, le -50 correspond au retranchement du coût de production d'un travailleur. L'application de cette équation durant quelques itérations produit les résultats présentés dans le tableau 4.2.

Tableau 4.2 Estimation de la réserve de ressources en début de partie et déduction des dates de construction des bâtiments.

Bâtiment	Itération	Réserve (minerai)	Temps (secondes)
	0	3.33	20
	1	20.00	40
	2	50.00	60
	3	93.33	80
Dépôt	4	36.67	100
	5	93.33	120
Baraquement	6	13.33	140
	7	96.67	160
Dépôt	8	80.00	180
	9	176.67	200
	10	300.00	220
Centre de Commandement	11	36.67	240
	12	186.67	260
	13	350.00	280

La limite de population oblige à créer des *dépôts* pour pouvoir continuer la production de travailleurs toutes les 20 secondes. De ce fait, le temps de production de ces *dépôts* est imposé et influence la quantité de ressources en réserve, ce qui conditionne la suite de la planification. En ajoutant comme souhaité un *baraquement* dès que possible pour la défense, on fixe indirectement le moment où l'on pourra ensuite effectivement créer la première extension, d'où les résultats présentés.

L'obtention de ces données pour les premières minutes de jeu permet de déterminer que pour assurer une capacité de production de ressources maximale, il faut lancer la création de la seconde base à exactement quatre minutes après le début de la partie, en suivant également les dates de création des bâtiments avant. Créer l'extension plus tôt ou plus tard impactera négativement la capacité de production de ressources. En plus des temps de création des bâtiments et de la première extension, on peut naturellement déduire de ces résultats le *build-order* optimal d'ouverture à appliquer pour SRbotOne. En conséquence, la séquence d'unités et de bâtiments à produire en début de partie mise en place est la suivante :

⟨*SCV, SCV, SCV, SCV, Dépôt, SCV, SCV, Baraquement, SCV, SCV, Dépôt*⟩

SCV est le nom dans StarCraft d'un travailleur de race *Terran*. On peut remarquer qu'on ne spécifie pas dans le *build-order* d'ouverture la liste complète jusqu'au *centre de commandement* : ceci se justifie par la présence au sein du bot d'un système qui prend ensuite le relais (mais respecte les écrits en planifiant automatiquement les derniers travailleurs et l'extension). En résumé, la stratégie d'ouverture choisie consiste à produire des travailleurs de façon constante et les envoyer récolter des ressources en permanence. Juste avant que la limite de population soit atteinte, on envoie un travailleur construire un *dépôt*, et un autre construire un *baraquement* dès que possible. Ils sont ensuite ré-affectés à la collecte de ressources. À nouveau, dès que la réserve le permet, on envoie un travailleur construire la seconde base, ce qui marque généralement la fin de l'ouverture d'une partie de StarCraft.

Pour terminer concernant la stratégie d'ouverture, une des ambitions de départ a été d'implémenter le système de *walling* décrit par Certický (section 3.1.2). Malheureusement, la solution proposée par l'auteur ne permet pas d'obtenir des résultats satisfaisants dans des conditions réelles. En effet, le résultat optimal

souhaité est obtenu uniquement sur un faible nombre de cartes de jeu, le système étant inefficace la plupart du temps. De plus, en addition à son coût en ressources élevé, lorsque le système fonctionne, une gestion complexe supplémentaire doit être mise en place. Si l'entrée de la base est alors défendue des intrusions précoces ennemies, le joueur qui l'utilise se retrouve lui bloqué à l'intérieur, ce qui empêche la récolte de ressources supplémentaires et l'extension rapide. Bien sûr, il existe des solutions que les joueurs humains emploient, cependant on a jugé que l'effort et la gestion supplémentaire constituent des inconvénients trop importants pour justifier l'emploi de cette technique. De ce fait, bien qu'implémenté dans SRbotOne, le *walling* n'est pas utilisé dans la stratégie d'ouverture mise en place.

4.1.2 Choix de la stratégie principale

La définition de la stratégie principale de SRbotOne est passée par plusieurs itérations lors de sa mise en place. Afin de respecter les objectifs présentés auparavant et d'utiliser du raisonnement spatial, on a dans un premier temps imaginé une stratégie de contrôle de territoire et de monopolisation des ressources. La figure 4.1 représente une carte de jeu sous forme de graphe pour schématiser cette première intuition et permet d'en illustrer les lignes directrices.

Les sommets représentent des régions et les arêtes indiquent qu'il est possible de passer d'une région à l'autre. Les sommets contenant un carré représentent les bases que l'on possède et le cercle inscrit dans un autre désigne la base ennemie. Les petits carrés, ronds noirs et triangles bleus autour d'un sommet indiquent que la région correspondante contient respectivement des bâtiments défensifs, des unités et des ressources. Enfin, les flèches rouges entre deux régions indiquent le mouvement de troupes.

En commençant en bas au milieu de la figure, l'idée est de s'étendre rapidement et le plus possible sur la carte et de construire des bases sécurisées par des défenses

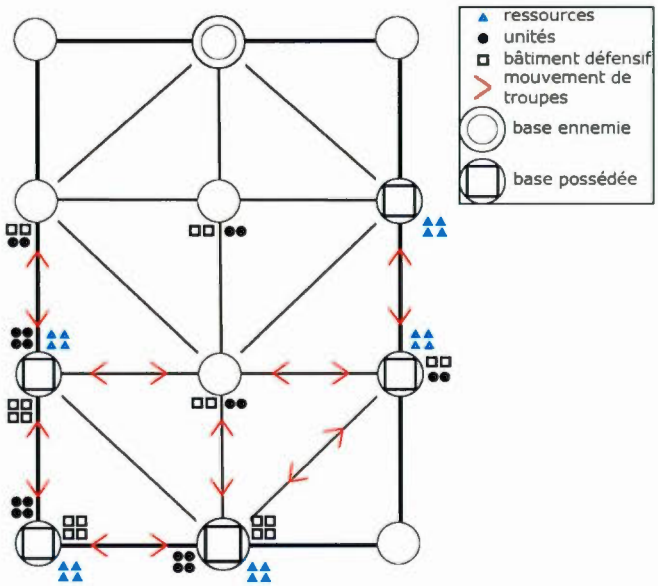

Figure 4.1 Première stratégie envisagée pour SRbotOne.

dans des régions contenant des ressources. De cette façon, on s'assure un contrôle du territoire et des ressources, et une vision d'ensemble sur la carte permettant d'anticiper les mouvements de l'adversaire. On part sur le principe de construire beaucoup de bases et de bâtiments défensifs et de répartir des unités partout sur la carte. Ainsi, on peut ensuite mettre en place des patrouilles et embuscades entre les régions occupées et attendre l'adversaire pour le piéger et le contrer par la suite. De plus, cette stratégie offre l'avantage de le priver des ressources, on part du principe que plus on possède ou on lui empêche d'accéder aux ressources (sans forcément les exploiter) moins lui en aura. De ce fait, on tente de gagner la supériorité sur l'ennemi grâce aux ressources, reprenant le principe militaire d'attrition.

Malheureusement, cette stratégie implique une division des unités sur la carte pour la répartition et la mobilisation de beaucoup de ressources pour la construction des défenses. Ces ressources sont finalement des pertes, puisqu'en réalité on constate

que l'ennemi envoie toutes ces forces au combat et tente de se rendre vers la base principale de la façon la plus directe possible. Ainsi, la plupart des régions qu'on aura protégées ne seront même pas explorées par l'ennemi. D'autre part, mener de petits fronts comme le suggère cette première tentative ne se révèle pas être une bonne idée. En effet, puisque l'adversaire envoie toutes ses unités d'un coup, s'il tombe sur une des régions que l'on aura fortifié, il passera sans problèmes vu que l'on a dispersé les unités sur la carte et que l'on se retrouve en infériorité partout où il pourrait attaquer. Pour finir, en plus de sa complexité, cette stratégie n'est pas viable parce qu'en pratique, une partie de StarCraft ne dure pas assez longtemps pour permettre de s'étendre autant et imposer un contrôle si étendu de la surface de jeu.

À partir de ce constat, on a pu mettre au point une deuxième stratégie (figure 4.2). À présent, l'objectif n'est plus de contrôler la carte et les ressources (qui ne sont au final pas si déterminantes), mais de deviner le chemin que va emprunter l'ennemi pour attaquer. Une fois le chemin entre les deux bases trouvé, on choisit une région sur ce chemin qu'on va renforcer au maximum en y plaçant beaucoup de bâtiments défensifs et d'unités. De cette façon, on tente de forcer une zone de combat et un gros front en maximisant les chances puisqu'on ne divise plus les troupes et on attend que l'adversaire vienne attaquer et se heurter aux fortifications préparées, avant de lui porter une contre-attaque fatale.

Concernant la politique d'expansion, on se concentre sur la création d'une deuxième base uniquement, et on se contente de créer des avant-postes et de placer quelques unités dans les zones à ressources, pour tenter d'au moins gêner l'adversaire s'il s'en approche (en plus de récupérer l'information). On envoie aussi des unités se placer dans les goulots d'étranglement des régions les plus proches de l'ennemi, pour espionner ses agissements et pouvoir anticiper s'il s'avère qu'il ait décidé de prendre un autre chemin que le chemin prévu, ou pour pouvoir réagir si l'estima-

tion est fausse. De ce fait, on a la possibilité de réagir et de déplacer latéralement ou vers l'arrière l'ensemble des troupes stationnées dans la région fortifiée si l'on se rend compte que l'ennemi tente de contourner les défenses avec toute son armée. Si les unités qu'on a placées à l'avant survivent ou arrivent à s'échapper à temps, on obtient même la possibilité d'attirer l'ennemi et le rediriger vers la région qu'on aura sécurisée, en employant par exemple la technique du *kiting*.

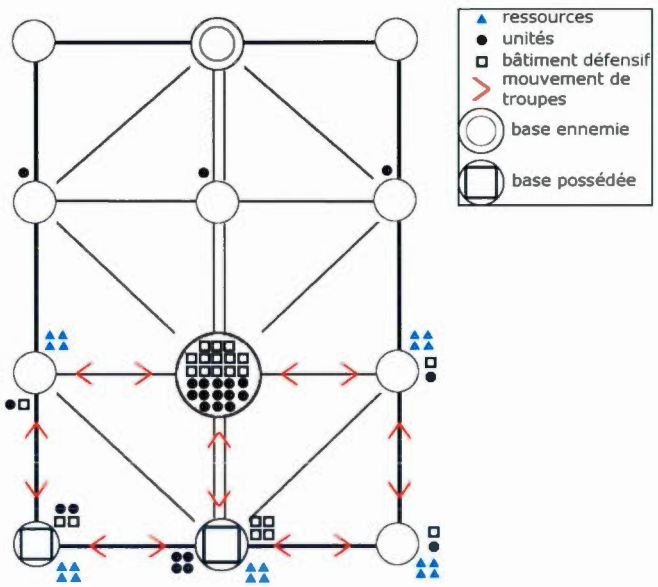

Figure 4.2 Deuxième hypothèse de stratégie pour SRbotOne. Les doubles arêtes qui relient les deux bases illustrent l'estimation qu'on fait du chemin qu'empruntera l'adversaire pour attaquer.

De la même façon, l'implémentation de cette stratégie a permis d'obtenir des observations et d'affiner encore la stratégie pour la troisième et dernière itération (figure 4.3). On a une nouvelle fois constaté que répartir trop les unités sur le terrain n'est pas bénéfique et constitue une perte en ressources qu'on a tout intérêt à limiter, afin de pouvoir renforcer davantage la région choisie. Car d'un autre côté, des tests durant le développement ont montré que l'estimation du chemin le

plus probable de l'ennemi est généralement fiable et peut être suivie pour choisir une région à défendre. Ainsi, on peaufine encore la stratégie en supprimant tous bâtiments défensifs et unités placées hors des zones clés de la carte, ce qui résulte en la fortification et le placement initial des unités sur seulement deux régions (la base de départ et la région qu'on décide de sécuriser).

On continue tout de même à s'étendre vers une deuxième base dans un premier temps et dès que possible en rapport au calcul de la section précédente. Aussi, même si la figure 4.3 ne l'indique pas, durant la partie on est amené à créer d'autres extensions par besoin de ressources (cela n'influe pas sur la stratégie globale). De plus, on décide de ne pas protéger du tout les extensions pour ne gâcher aucune ressource et pouvoir se focaliser sur la production d'unités et de défenses pour la région de combat. Les régions d'extensions contiennent uniquement un *centre de commandement* pour qu'on y amène les ressources collectées.

Cette simplification s'avère payante et réaliste en conditions de jeu. Au final, la stratégie de SRbotOne consiste donc à calculer le chemin entre les bases des deux joueurs et déterminer quelle région empruntée par ce chemin est la plus pertinente à sécuriser pour pouvoir bloquer l'ennemi. On effectue cette tâche en parallèle à l'ouverture de la partie, ainsi tant que la région clé n'est pas définie on va produire des unités qui ont pour vocation de défendre la base. Dès que la région est sélectionnée, on envoie des travailleurs pour y construire des bâtiments défensifs (*bunkers*, tourelles anti-aériennes), ainsi que les troupes qui attendaient à la base pour protéger les travailleurs lors de la construction. À chaque *bunker* construit, les unités sur place y entrent et le remplissent en se préparant à l'attaque ennemie. Les autres tiennent une position entre les défenses et la base ennemie. À partir d'un certain seuil de bâtiments défensifs qu'on juge suffisant, les troupes qui ne peuvent se protéger à l'intérieur viennent se réfugier et attendre derrière les défenses.

Vient alors une phase d'attente de l'attaque de l'adversaire. Si l'on survit à quelques attaques ennemies et que l'on juge qu'il est temps de contre-attaquer, on envoie toutes les forces restantes à l'assaut de sa base jusqu'à la fin. Durant toute la partie, on produit sans cesse des unités qui viennent s'ajouter à la défense de la région fortifiée ou qui vont directement vers la base ennemie si on a lancé le contre. Dans les cas où l'ennemi n'attaque pas, on décide de sortir à un moment pour marcher vers la base ennemie avec l'ensemble de l'armée.

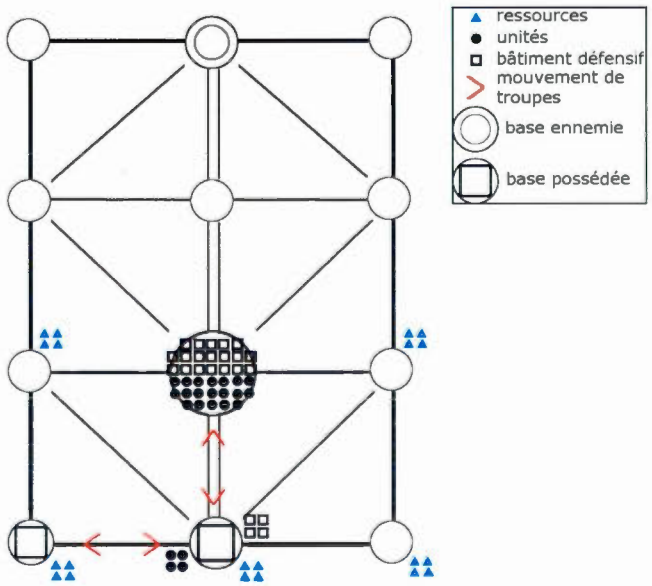

Figure 4.3 Stratégie finale employée par SRbotOne. On optimise l'utilisation des ressources et la sécurisation de la région choisie pour l'affrontement.

4.1.3 Choix de la région à sécuriser

Ayant choisi une stratégie basée sur une région dans laquelle la majorité de la partie doit se jouer, on comprend que le choix de cette région est primordial. On doit s'assurer que c'est un bon endroit pour bloquer et repousser l'adversaire, et surtout qu'on se trouve sur le chemin qu'il va prendre, ou en tout cas celui qu'il

empruntera le plus probablement. Le processus de sélection de la région à sécuriser qu'on appelle « région clé » est composé de plusieurs étapes.

Premièrement, le besoin préalable à tout ce qui suit est de connaître la position de la base ennemie. Pour obtenir cette information, on envoie en tout début de partie un travailleur explorer la carte vers les emplacements les plus probables où la base ennemie peut se trouver. Le placement des deux joueurs en début de partie est aléatoire, cependant des zones sont prédéfinies (et toujours les mêmes) pour chaque carte de jeu, et chaque joueur se voit attribuer une de ces zones au hasard. Ces zones sont connues grâce à l'interface BWTA, ce qui permet d'envoyer le scout (travailleur) explorer des endroits précis jusqu'à trouver l'adversaire. En plus d'obtenir la position de la base ennemie, cette étape permet aussi de déterminer sa race, ce qui offre une information d'importance qu'on peut ensuite utiliser pour adapter certains détails.

Dans un second temps, on tente de trouver un chemin entre les bases des deux joueurs. On peut s'interroger sur comment trouver le bon chemin sachant qu'il y en a toujours plusieurs possibles. L'observation des tendances générales en compétition de bots dans StarCraft, et de l'IA d'origine du jeu, démontre (de façon prévisible) que le chemin à privilégier est le chemin le plus court entre les deux bases. Ceci s'explique par le fait qu'au début de partie, les premières unités de chaque race ont sensiblement la même vitesse de déplacement, et qu'aucune unité volante n'est accessible. D'autant que puisque beaucoup de stratégies sont basées sur le *rush* de l'adversaire, l'objectif commun est d'arriver le plus vite chez l'adversaire et par conséquent de prendre le plus court chemin. En conséquence, on emploie l'algorithme A* au sein de SRbotOne pour trouver ce chemin dans le jeu.

Étant donné que l'algorithme A* retourne un chemin sous forme de liste de positions (positions de cases, la carte étant découpée en cases), une troisième étape est

nécessaire avant de pouvoir réfléchir à la région clé. Celle-ci est assez triviale mais obligatoire : on va parcourir la liste complète des cases que composent le chemin obtenu pour en extraire une nouvelle liste de régions auxquelles appartiennent ces cases. Ainsi, on transforme le chemin de positions en un chemin fait d'une liste de régions qui sera plus exploitable par la suite.

Quatrièmement, avec les éléments dont on dispose, on peut enfin déterminer quelle sera la région clé. Pour ce faire, on parcourt la liste des régions sur le chemin qu'on a calculé. Si la région qu'on observe correspond à la région de départ d'un des deux joueurs, on l'ignore, car une région contenant une base principale ne peut pas être la région clé.

Si la région est trop éloignée de l'endroit où l'on démarre (ou trop proche de la base de l'ennemi) on l'ignore également, car on prend trop de temps pour s'y rendre et pour la sécuriser. En effet, une région trop proche de l'ennemi est plus susceptible d'être attaquée rapidement, avant qu'on ait pu bien la fortifier, et prend aussi plus de temps à être approvisionnée par les renforts. Ainsi, puisque la liste des régions qui relient les deux bases est ordonnée de la base du joueur à celle de l'ennemi, on arrête de chercher si on dépasse la moitié de la liste.

Si la région qu'on étudie ne remplit pas une des deux précédentes conditions, alors on la définit comme région clé. À partir du moment où l'on a défini une région clé, on continue de parcourir les régions dans la liste et on va les comparer à la région clé actuelle, si elles ne remplissent pas les conditions qui empêchent d'être région clé elles-mêmes.

Lors d'une comparaison entre la région clé et une autre région de la liste, si la seconde a plus de goulots d'étranglement que la région clé, elle devient alors la région clé. De cette façon, on choisit de sécuriser la région sur le chemin qui dispose de plus de régions voisines, ce qui permet de bloquer un maximum de

chemins annexes. Si les deux régions que l'on compare ont le même nombre de goulots d'étranglement, alors on va comparer leur taille. Ainsi, si la région qu'on a sélectionnée dans la liste est plus petite que la région clé actuelle, alors elle devient la nouvelle région clé. Ceci simplement parce qu'il est plus facile de défendre et sécuriser complètement une région de plus faible superficie. En procédant de cette manière, on dispose, après avoir parcouru la liste complète des régions sur le chemin le plus court entre les deux bases, d'une région clé unique qu'on va pouvoir sécuriser.

Après l'obtention de la région clé, une dernière étape est requise avant de pouvoir exploiter l'information et mettre la stratégie en place. Effectivement, si l'on dispose de la région à sécuriser, il faut encore savoir à quel endroit placer les défenses. Plus précisément, la réelle question concerne le placement des premiers bâtiments défensifs, l'objectif est de remplir au maximum la région, mais le placement des premières défenses est important. Dans la pratique, il est rare qu'on ait le temps de fortifier entièrement une zone avant de subir la première attaque, la position à laquelle on a placé les défenses est alors cruciale et influence même la suite de toute la stratégie. Des défenses mal placées au départ ne résisteront pas à l'assaut ennemi, ce qui peut entraîner la perte de la partie par exemple, ou retarder la fortification de la région clé, ce qui entraînera également par la suite l'inefficacité de la défense. Ainsi, et pour les mêmes raisons qu'auparavant, on décide de placer les premières défenses au niveau du goulot d'étranglement de la région clé le plus proche de l'endroit où l'on a débuté la partie. Pour réaliser cette étape, on parcourt la liste des goulots d'étranglement de la région clé et on compare la distance du centre du goulot à la base. La position du goulot à la distance la plus proche est choisie pour les premiers bâtiments.

Les conditions de jeu réelles d'une partie de StarCraft obligent à prendre en compte une dernière subtilité. En effet, il existe des endroits sur la carte où il est impossible

de construire en raison d'irrégularités sur le terrain. Ce paramètre peut entraîner un problème et invalider le processus de sélection de la région clé mis en place. Il peut arriver qu'aucune place ne soit constructible dans la région clé ou qu'il ne soit possible de construire qu'un trop petit nombre de défenses. Dans ce cas, un système a été mis en place pour placer cette région dans une nouvelle liste de régions non candidates pour être la région clé, et pour relancer le processus de sélection en ne prenant pas en compte les régions présentes dans cette liste. Le processus final de choix de la région à sécuriser est résumé par la figure 4.4.

pour toute région r sur le chemin entre les deux bases
  si la région r est une région de départ **alors l'ignorer**
  si r est dans la liste des régions non candidates **alors l'ignorer**
  si r est trop éloignée **alors l'ignorer**
  si on n'a pas encore de région clé **alors** r le devient et on continue avec la suivante de la liste
  si r a plus de goulots que la région clé **alors** r devient la région clé
  sinon si r a autant de goulots que la région clé **alors**
    si r est plus petite que la région clé **alors** r devient la région clé
  fin du si
fin

Figure 4.4 Processus de choix de la région à sécuriser.

Pour finir, on a implémenté au sein de SRbotOne une fonctionnalité supplémentaire pour la sélection de la région clé afin d'imposer qu'elle soit la région voisine à la région de départ (afin de gagner du temps). Ceci dans le but de lutter contre les stratégies de *rush*, qui grâce à leur rapidité pouvaient parvenir à passer avant qu'on commence à sécuriser la région choisie. Cette fonctionnalité est paramétrable, on peut par exemple l'employer contre des adversaires connus pour leurs stratégies de *rush* ou contre les ennemis d'une race en particulier (comme les *Zergs*).

4.1.4 Adaptations stratégiques et déroulement

Afin de terminer l'explication de la stratégie déployée par SRbotOne, on doit encore détailler la composition de l'armée qu'on produit, ainsi que les variations introduites et le raisonnement temporel qui régit les décisions concernant le passage de la défense à l'attaque.

La composition de l'armée est choisie pour être la plus optimisée possible avec la stratégie directrice mise en place. On décide de tout miser sur la production de deux types d'unités différentes uniquement, les *marines* et les *medics*. Dans StarCraft, le *marine* est l'unité de combat de base de la race *Terran*, sa facilité de production (disponible dès le début de la partie) et ses particularités en font un choix logique pour la stratégie visée. En effet, le *marine* peut être placé dans les bâtiments défensifs comme le *bunker* (que l'on utilise beaucoup avec la stratégie choisie), ce qui lui fait en plus gagner en portée d'attaque. Vu que l'on se focalise sur ce type d'unité, on effectue aussi des recherches en jeu pour augmenter ses caractéristiques et capacités.

Dans le jeu, une *recherche* est une augmentation des statistiques de base d'un type d'unité, une nouvelle capacité pour un type d'unité, ou une nouvelle technologie pour le joueur, qu'on obtient (uniquement pour la partie en cours) moyennant des ressources et du temps. C'est-à-dire qu'un joueur doit effectuer les recherches à chaque partie si il veut en avoir les bénéfices. Pour les *marines*, la stratégie prévoit qu'on achète, par le biais des recherches, des améliorations de portée d'attaque, de puissance d'attaque et d'armure, ainsi que la capacité dite de *stim pack*. Ainsi, chacun des *marines* que l'on possède ou produit verra ses caractéristiques augmenter et obtiendra la nouvelle capacité. *Stim pack* est une capacité qui augmente la vitesse de déplacement et d'attaque du *marine* de 50% pour une durée d'environ 10 secondes et coûte 10 points de vie à l'unité qui l'utilise. Une unité ne

peut utiliser qu'un *stim pack* à la fois, c'est à dire que pendant les 10 secondes d'activation de la capacité, un *marine* ne peut en utiliser un autre (il faut attendre la fin des 10 secondes).

Ainsi, l'armée est composée de *marines* augmentés et son plein potentiel est atteint grâce à l'emploi des *medics*. Le *medic* est une unité de classe moyenne qui peut soigner ses alliés. On produit des *medics* avec l'objectif de conserver un ratio de un pour cinq *marines*. L'apport de *medics* dans la composition de l'armée est important et non négligeable. En plus de permettre à l'armée de résister plus longtemps grâce aux soins, il permet également d'utiliser au maximum la capacité *stim pack* des *marines*, puisque les soins permettent aussi de contrer son coût et donc de l'utiliser presque en permanence. De cette façon, on peut exploiter les *marines* au maximum de leur capacité, et la stratégie consiste à en produire le maximum possible en continu. On fait le choix de produire beaucoup d'unités peu chères en misant sur le grand nombre plutôt que de produire moins d'unités plus chères, car c'est ce qui correspond le plus à la philosophie recherchée. Cette composition d'armée est connue des joueurs de StarCraft, on l'appelle communément « MnM » pour « Marine and Medics ».

Dans le but de contrer les autres bots qui utilisent de l'apprentissage et de pouvoir s'adapter aux adversaires et aux événements qui surviennent durant une partie, on introduit dans la composition standard de l'armée un petit nombre d'autres unités selon les cas. Par exemple, avec la stratégie que l'on a choisie, il faut s'attendre à subir des situations de siège. Or, si l'ennemi déploie un grand nombre d'unités ou des unités de plus haut niveau comme des unités de siège, il est possible que les défenses ne puissent tenir et finissent par céder. C'est pourquoi dès lors que l'on détecte des unités de siège, on décide d'en produire également ayant constaté que ce genre d'unité est aussi très efficace pour se défendre lors d'un siège. Dans ces cas, on produit des *tanks* (entre 2 et 4) qui, même si ils sont des unités d'attaque

initialement, sont aussi efficaces en défense grâce à leurs tirs qui font des dégâts de zone (pouvant toucher plusieurs unités à la fois), et leur longue portée. Cela les rend très pratiques lorsqu'on est assailli par beaucoup de troupes ennemies. En plus, l'investissement n'est pas vain, puisque dans les cas où l'on contre-attaque, les *tanks* servent aussi.

Dans d'autres situations, comme la détection d'un type particulier d'unité, ou le fait de jouer contre une certaine race, on apporte également des variantes à la composition de l'armée. Par exemple, lors de la détection de *lurker* qui est un type particulier d'unité *Zerg* qui peut s'enfouir sous terre et ainsi attaquer sans être vu, on ordonne la création de troupes volantes (*wraith*), celles-ci étant les unités de contre par excellence du *lurker*. Cette évolution de l'armée permet en plus de gagner beaucoup en capacité de déplacement et d'exploration. De plus, si les conditions sont réunies, il est tout à fait possible de combiner et de mixer les variantes de stratégie, ainsi on peut se retrouver avec l'armée standard composée en plus de *tanks*, unités volantes et autres.

Comme le choix de la région que l'on décide de sécuriser, le choix du moment où l'on va contre-attaquer ou attaquer l'ennemi est crucial dans la stratégie, et déterminant dans le dénouement final de la partie. En effet, il faut trouver le bon moment, celui où l'on a l'avantage sur l'adversaire. La complexité d'une partie de StarCraft rend cette décision non triviale et difficile à résumer en peu de critères. Avec la stratégie mise en place dans SRbotOne, la problématique réside dans le fait de ne pas attendre trop avant de contre-attaquer, mais aussi de ne pas envoyer toute l'armée à l'attaque trop vite. Cette situation peut être vue comme un quitte ou double, à partir du moment ou l'on décide d'attaquer, si c'est trop tôt, on n'aura peut-être pas assez de troupes pour passer les défenses de l'ennemi et le vaincre totalement. Dans ce cas, il pourra à son tour attaquer et remporter la victoire facilement ayant déjà éliminé toute opposition. D'autre part, si l'on

attend trop avant d'attaquer, soit on ne profitera pas de l'avantage gagné par les défenses que l'on a prévues, car on laissera le temps à l'ennemi de reconstituer son armée, soit de la même façon, attendre trop l'aura laissé former une armée assez conséquente pour passer les fortifications et gagner la partie. Le quitte ou double semble peut-être extrême, mais dans StarCraft, pour la victoire finale, mieux vaut engager toutes les unités que l'on possède dans la bataille plutôt que des plus petits groupes. En effet, le nombre est souvent plus important que le type des unités, et ce fait est d'autant plus constaté avec des stratégies comme « MnM » dont c'est le principe de base.

Au sein de SRbotOne, à chaque fois que l'on se questionne sur le fait d'attaquer ou non, on passe par une procédure de vérifications qui permet de déterminer si c'est le bon moment (figure 4.5). Tout d'abord, on observe le temps écoulé depuis le début de la partie. Une partie de StarCraft se découpe en trois temps, le début, le millieu et la fin. Le début de partie représente les 5 ou 6 premières minutes de jeu, le millieu est considéré durer jusqu'à la 18*e* minute, et le temps écoulé après est considéré comme fin de partie. Dans le bot, la première vérification s'occupe de voir si l'on a atteint ou dépassé le temps qui marque la fin de partie. Si oui, on retourne la commande d'attaquer systématiquement. Le problème vient du fait qu'on peut considérer qu'en fin de partie, l'adversaire est capable de produire les unités les plus coûteuses du jeu (très puissantes) qui prendront le dessus sur les unités de base et la stratégie « MnM ». On n'a donc plus le temps d'attendre, il faut attaquer. Si la fin de partie n'est pas atteinte, on effectue alors d'autres vérifications. On précise aussi que le design de SRbotOne permet de changer et d'ajuster la valeur de basculement des 18 minutes, ce qui permet de perturber et d'empêcher les ennemis qui utilisent de l'apprentissage de facilement cerner le comportement du bot, ainsi que de s'adapter à différents adversaires et repousser l'action si l'on sait qu'on a davantage de temps avec eux.

Dans les cas où la condition de temps n'est pas atteinte, on va attendre de repousser deux fois l'ennemi avec les défenses dans le but de l'affaiblir et de prendre l'avantage sur lui. À partir de ce moment, on vérifie le nombre de pertes de l'ennemi. S'il a perdu 33% de ressources de plus que SRbotOne lors des affrontements, on déclare que c'est le bon moment pour contre-attaquer. Sinon, on vérifie si on dispose d'une armée de plus de 40 unités pour le combat (sur 200 maximum), dans tel cas, on décide également d'attaquer. Si aucune des deux conditions précédentes n'est vérifiée, on continue d'attendre et de défendre.

Les valeurs de 33% et 40 unités ont été fixées empiriquement (au cours de tests durant le développement) et montrent des résultats satisfaisants mais peuvent encore être affinées. Puisqu'ils sont vérifiés fréquemment, ces critères apportent des comportements non figés et évolutifs. En effet, on pourra donner l'ordre d'attaque, mais se trouver dans des situations où il est préférable durant l'attaque de se replier afin de ne pas perdre toute l'armée si l'ennemi se défend bien. Dans ces cas, étant donné que le ratio ne sera plus aussi favorable, la décision de se replier et de retourner défendre dans la zone sécurisée sera prise automatiquement.

si le temps de jeu est supérieur à *tAttaque* **alors attaquer**
si on a résisté à au moins deux attaques adverses **alors**
  si l'ennemi a 33% de pertes de plus **alors attaquer**
  sinon si on a au moins 40 unités pour le combat **alors attaquer**
  sinon défendre
fin

Figure 4.5 Procédure pour déterminer le moment propice à l'attaque.

Pour clôturer cette section, on présente le flux logique de SRbotOne (figure 4.6), étant donné qu'on s'est basé sur UAlbertaBot, on peut le comparer au sien (figure 2.7). Lors de la conception de SRbotOne, chaque partie de UAlbertaBot a

été retouchée à plus ou moins grande échelle, cependant le flux logique ne s'en retrouve pas forcément modifié, c'est pourquoi sur la figure 4.6 certains niveaux sont identiques. Afin de rendre plus claire la compréhension, les ajouts et modifications apportées au flux logique sont indiqués en gras sur la figure. On peut constater des changements majeurs au niveau du *CombatCommander*, ce qui est logique étant donné que la majorité des éléments stratégiques des bots y est contenue.

Comme on peut le voir, la stratégie et l'activité de SRbotOne sont conditionnées par la région clé. La gestion différente des troupes par rapport UAlbertaBot entraîne également des divergences dans la gestion des *Squads*. On note aussi l'ajout important du calcul de la région qui est pris en charge par *InformationManager*, même s'il dépend aussi de *ScoutManager* indirectement.

Enfin, on remarque l'ajout d'un élément *BaseManager* au début du flux logique, conçu pour permettre une meilleure gestion des expansions et bases (manquante dans UAlbertaBot parce que certainement pas un sujet majeur du bot). Ses tâches principales sont la mise à jour des données sur les bases et la récupération des commandes (besoins ou requêtes de production) en bâtiments et unités de chacune des bases. Il est intimement lié au *ProductionManager* mis en place au départ dans UAlbertaBot. On revient en détails sur ce nouveau manager et d'autres ajouts dans la section suivante.

GameCommander.update()
- — **BaseManager.update()**
  - — **Mettre à jour les informations sur les bases**
  - — **Récupérer les commandes en bâtiments et unités des bases**
- — WorkerManager.update()
  - — Identique à UAlbertaBot
- — ProductionManager.update()
  - — Identique à UAlbertaBot
- — BuildingManager.update()
  - — Identique à UAlbertaBot
- — CombatCommander.update()
  - — Envoyer un travailleur faire du *scouting* si c'est le bon moment
  - — **Si la région clé est connue** :
    - — **Si une bataille est en cours, mettre à jour les informations**
    - — **Sinon, si on doit attaquer** :
      - — **Vider les *bunkers***
      - — **Attaquer la base ennemie**
      - — **Utiliser les *stim packs***
    - — **Sinon, attendre une bataille**
    - — **Si on est attaqué, se réfugier dans les *bunkers***
  - — Squads.update()
    - — Faire une simulation de combat avec SparCraft
    - — Si la simulation prédit la victoire, continuer d'attaquer
    - — Si l'attaque continue, appeler la sous classe *MicroManager* pour chaque unité
    - — Si la simulation prédit la défaite, se replier vers la base **ou la région clé**
    - — **Si un groupe dépasse 4 unités, en créer un nouveau**
- — ScoutManager.update()
  - — Identique à UAlbertaBot
- — InformationManager.update()
  - — **Si la région clé n'est pas déterminée, la calculer**
  - — Si une unité ennemie est visible, enregistrer sa dernière position connue
  - — Si une unité ennemie meurt, ajouter la perte en ressource au total perdu

Figure 4.6 Flux logique de SRbotOne, avec différences en gras par rapport à UAlbertaBot (figure 2.7).

4.2 SRbotOne : Détails de développement

4.2.1 Système de gestion des bases

Lors de la réflexion pour le choix de la stratégie principale, et notamment la première hypothèse formulée, on a identifié un manque dans la solution proposée par UAlbertaBot. En effet, la stratégie étant basée sur la création de nombreuses bases, on a remarqué l'absence au sein d'UAlbertaBot d'un système de gestion de bases. Le bot se contente de produire des *centres de commandement* pour s'étendre, mais ne dispose pas de code pour une gestion plus avancée, comme la répartition des bâtiments et unités entre les bases, leur spécialisation, etc. En fait, tout est programmé pour être créé près du point de départ du joueur.

Ainsi, SRbotOne introduit une classe *Base* permettant de créer des objets concrets et un manager (*BaseManager*) de plus haut niveau pour en permettre la gestion. Une base est caractérisée par une localisation (position, nombre de ressources disponibles, etc.), une liste de bâtiments construits, une liste de bâtiments commandés, et la région dans laquelle elle se trouve. Elle contient aussi des attributs comme des limites de compte pour des bâtiments spécifiques, des compteurs pour les types de bâtiments, et des booléens permettant par exemple de savoir si elle est la base principale, si elle a un mur, etc.

BaseManager contient la liste des bases du joueur et sert d'interface entre les bases et les autres modules du système. Il permet la mise à jour des informations et attributs concernant les bases, l'interaction éventuelle avec le *WallingManager* (si la base est la principale et qu'on utilise du *walling*), en plus de la communication avec le *StrategyManager* et le *ProductionManager* (qu'on a modifiés à partir de la version de UAlbertaBot), pour la planification de *build-orders* (séquences de bâtiments à construire).

Durant une partie, à chaque création de *centre de commandement*, un objet *Base* est créé automatiquement et spécialisé en fonction des informations du terrain environnant. C'est-à-dire qu'on personnalise la base selon la taille de la région dans laquelle elle se trouve. On paramètre notamment les limites de nombre de bâtiments de la base pour s'assurer que la base ne sera pas trop grande et ne dépassera pas la région, ou qu'on ne voudra pas mettre trop de bâtiments pour la place disponible. Ce système a été pensé pour fonctionner avec un système de commandes de bâtiments pour chaque base et coopère parfaitement pour en limiter les abus et incohérences.

En effet, dans SRbotOne chaque base est autonome et commande elle-même ses propres bâtiments lorsque le bot a terminé de produire les bâtiments planifiés auparavant, et qu'il entame le processus de nouvelle planification de *build-order*. De cette façon, on peut répartir la production de bâtiments et d'unités entre les bases et même l'automatiser, contrairement à UAlbertaBot. De façon plus intéressante, ce système permet de créer des bases spécialisées dans certaines tâches. Par exemple, créer une base concentrée uniquement sur la production d'unités, alors qu'une autre sera spécialisée dans la récolte de ressources, tandis qu'une autre sera utilisée pour augmenter la population maximum du joueur, etc. En pratique, on l'utilise pour différencier la base principale des extensions et de la base qui représente la région clé, en créant des types de bases différents.

4.2.2 Système de gestion des fronts

On emploie le système de gestion des fronts au sein du *CombatCommander*, pour la mise en place de la stratégie globale de défense et de contre-attaque de SRbotOne. Comme évoqué dans la section 4.1.4, on s'intéresse aux fronts, et plus particulièrement aux fronts dans la région clé pour déterminer le moment le plus opportun pour passer à l'action, et attaquer la base ennemie ou riposter. Dans

l'implémentation actuelle, on s'attend à être pris à parti par l'adversaire et l'objectif est de survivre au moins deux fois, c'est pourquoi ce système de gestion de fronts est important.

Par conséquent, on développe une classe *Battlefront* pour solutionner ces situations. Une instance de *Battlefront* contient une liste des unités de SRbotOne (présentes dans la région), une liste des unités de l'ennemi (visibles dans la région), un compteur de ressources perdues par SRbotOne (qui additionne les coûts en ressources des unités perdues par SRbotOne), et un compteur identique pour les pertes ennemies. Elle enregistre aussi les dates de début et de fin du front, et dispose de booléens pour vérifier la fin du front et si SRbotOne a gagné ou perdu ce front. Le *CombatCommander* maintient une liste des fronts et est le module dans lequel s'effectue la logique du système.

En pratique, dans StarCraft, à partir du moment où la région clé est trouvée, on commence à vérifier la présence d'un front dans la région. Un front est créé dès lors que les unités de SRbotOne dans la région clé sont attaquées et qu'il y a au moins trois unités ennemies dans la région. Ensuite, tant qu'un front n'est pas terminé et que l'affrontement est en cours, on met à jour en permanence les informations du front (compteurs de pertes, listes d'unités). Un front est considéré comme terminé lorsque toutes les troupes ennemies sont battues et que les troupes de SRbotOne ne sont pas attaquées pendant 20 secondes. Il peut aussi se finir si l'ensemble des unités adverses ne sont pas vaincues mais se replient, dans ce cas, on attend une minute avant de déclarer le front terminé et remporté. Si l'ensemble des unités de SRbotOne sont éliminées, le front est également terminé et perdu. À chaque fois que l'on se rend dans le *CombatCommander* dans l'exécution du code et qu'un front n'est pas en cours, on observe pour voir si un front n'a pas commencé, et on s'interroge sur la justesse du moment pour attaquer en suivant la procédure décrite à la figure 4.5. De cette façon, SRbotOne peut surveiller la présence de

fronts et le bot peut prendre intelligemment la décision de passer de la défense à l'attaque, lorsque les critères fixés sont atteints.

4.2.3 Comparaison avec UAlbertaBot

Si on reprend la figure 2.6 pour la comparer avec la figure 4.7, on peut observer les apports et modifications de SRbotOne par rapport à UAlbertaBot sur lequel il est basé. En vert, on présente les classes complètement nouvelles et en bleu, celles qui ont été instaurées par UAlbertaBot mais largement modifiées pour intégrer la philosophie et la stratégie mise en place au sein de SRbotOne.

Ainsi, on retrouve les classes *Battlefront*, et *BaseManager*, *Base*, *WallingManager*, ajoutées pour participer respectivement à la gestion du système de fronts, et du système de gestion de bases et extensions qu'apporte SRbotOne. Parmi les classes modifiées, *ProductionManager*, *BuildOrderSearch*, *BuildingManager* et *Building-Placer* ont été adaptées pour fonctionner avec le nouveau système de gestion de bases en place. Un deuxième ensemble composé de *CombatCommander* et *Squad* a été modifié pour permettre la gestion des troupes et fronts telle qu'on l'a implémentée dans SRbotOne. Ce même ensemble joint à *StartegyManager* regroupe des modifications concernant la stratégie globale de l'IA. Enfin, *InformationManager* et *GameCommander* contiennent respectivement des ajustements pour la définition de la région clé qu'on utilise, ainsi que des modifications d'ordre général pour le fonctionnement de SRbotOne au cours d'une partie.

UAlbertabot est un projet dont l'ordre de grandeur est d'environ 32000 lignes de code. SRbotOne compte un nombre de lignes dont l'ordre de grandeur est de 36000, ce qui constitue une augmentation d'environ 12.5%. De plus, avec les changements apportés aux classes existantes, on peut estimer que plus d'un tiers de la logique d'origine a été modifiée pour SRbotOne afin d'agir sur le comportement de haut niveau de UAlbertaBot.

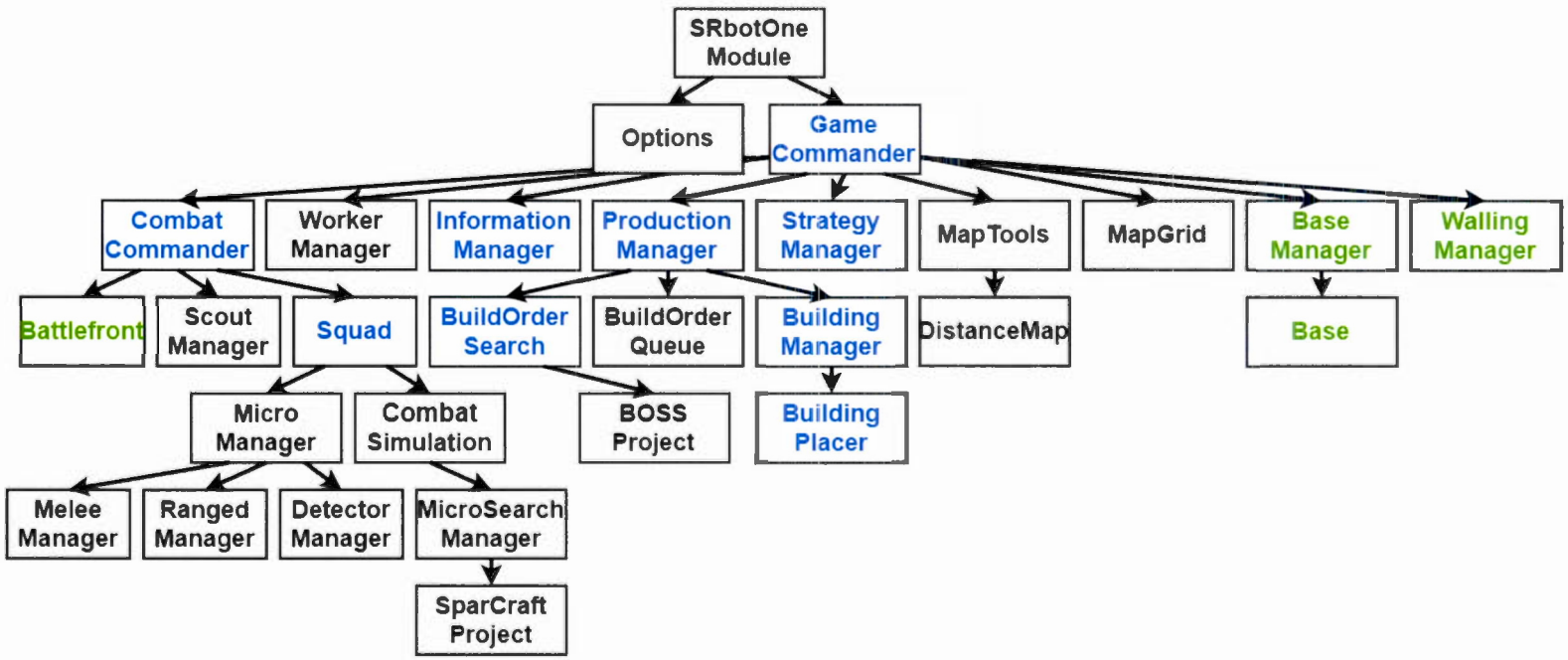

Figure 4.7 Hiérarchie des classes de SRbotOne. Contrairement à un diagramme UML, ici, une flèche indique une relation de responsabilité.

CHAPITRE V

EXPÉRIMENTATIONS ET ANALYSE DES RÉSULTATS

Pour évaluer SRbotOne, nous avons réalisé trois expérimentations sous forme d'enchaînements de parties de StarCraft. L'objectif de ces expérimentations est de confirmer qu'il est possible de construire une stratégie pour StarCraft en fonction du raisonnement spatial, en conservant de bons résultats, et dans le but d'améliorer l'adaptabilité des bots. Afin de pouvoir analyser plus précisément les apports de notre approche, nous avons développé un système de récolte de données au sein de SRbotOne qui génère des fichiers de journaux (section 5.1).

La première expérience a consisté à faire jouer SRbotOne contre l'IA originale du jeu (section 5.2). La deuxième fut de comparer SRbotOne à d'autres bots disponibles en code ouvert et qui ont déjà participé, avec succès, à des compétitions internationales de bots (section 5.3). Comme troisième expérimentation, nous avons soumis SRbotOne à une compétition dans le cadre de la conférence CIG 2016 (section 5.4). Pour conclure, la fin de ce chapitre (section 5.5) est consacrée à une analyse globale des résultats, tendances, et enseignements de l'ensemble des expérimentations, avant de se terminer par une comparaison avec UAlbertaBot.

5.1 Système de fichiers de journaux

Dans le contexte de StarCraft, il est difficile d'évaluer les performances et impacts d'une stratégie en observant uniquement le pourcentage de victoires, étant donné le nombre incalculable d'événements qui peuvent se produire et entrer en jeu dans la détermination du résultat d'une partie. Or, on a besoin de pouvoir obtenir des données plus pertinentes afin de pouvoir analyser les apports de la stratégie et de ses paramétrages. Puisque, en effet, en plus de tirer des enseignements, ces informations peuvent être utiles pour peaufiner et affiner certains paramètres en vue d'améliorer les performances d'un bot. C'est pourquoi on développe au sein de SRbotOne un système de fichiers de journaux personnalisé, qui en plus d'enregistrer les informations essentielles, cible surtout les points capitaux de la stratégie dans le but de déterminer la qualité des prises de décisions.

Pour les observations sur la prise de décision du bot, on garde en mémoire les moments de la partie dans lesquels l'IA a décidé de prendre l'initiative et de passer à l'attaque. Dans le code, on enregistre les données au fur et à mesure et lors de la prise de décisions (dans *CombatCommander*). On développe un programme en complément pour rassembler les données et faciliter leur analyse.

Dans un premier temps, on peut compter le nombre de victoires et de défaites sur la série et obtenir le pourcentage de victoire. Deuxièmement, de façon plus intéressante, on peut calculer le pourcentage de victoire lorsqu'on décide d'attaquer avant la période considérée comme la fin de partie (avant 18 minutes de jeu). Cela signifie qu'on a résisté à au moins deux fronts et que le ratio est assez avantageux, ou qu'on dispose d'assez d'unités avant 18 minutes de partie. On peut aussi déterminer le pourcentage de victoire quand on attaque après ces 18 minutes ou après un temps personnalisé. On doit également prendre en compte le pourcentage de victoire par plantage ou dépassement du temps de l'adversaire (ce qui arrive). Fi-

nalement, on peut extraire beaucoup d'autres informations intéressantes comme le nombre minimal de ressources perdues de l'ennemi, à partir duquel on gagne à tous les coups. De plus, on peut encore affiner chacune des statistiques précédentes en les recoupant par ennemi, par carte de jeu, ou même par ennemi et carte à la fois, si on dispose de suffisamment de données.

5.2 SRbotOne *vs* l'IA originale de StarCraft

On a organisé des parties entre SRbotOne et l'intelligence artificielle originale du jeu StarCraft version 1.16.1. SRbotOne utilise la version 4.1.2 de BWAPI et la version 2.2 de BWTA. Le tableau 5.1 décrit les spécifications de la machine utilisée.

Les parties ont été accélérées au maximum (42ms par image contre 67ms par image à vitesse normale) et leur lancement en continu est automatisé et paramétré de la façon suivante :

- le type de partie est toujours un contre un (SRbotOne contre l'IA du jeu) ;
- la race de l'adversaire est choisie aléatoirement entre les trois disponibles ;
- la race de SRbotOne est toujours *Terran* ;
- le point de départ des joueurs est choisi de façon aléatoire parmi les points prédéfinis pour chaque carte.

À chaque partie, une carte est choisie aléatoirement parmi 30 cartes tirées des listes officielles des compétitions AIIDE 2015 et CIG 2016. Ce corpus contient :

- 10 cartes à deux joueurs (2 points de départ par carte) ;
- 8 cartes conçues pour trois joueurs (3 points de départ par carte) ; et
- 12 cartes à quatre joueurs (4 points de départ par carte).

À noter que, bien que plusieurs cartes soient conçues pour plus de deux joueurs, il n'y a toujours que deux bots qui s'affrontent dans nos expérimentations. Mais

les cartes peuvent être de plus grande taille et la différence se fait surtout sur le nombre de points de départ. Cela indique qu'il peut y avoir plus de variations avec ces cartes.

Tableau 5.1 Caractéristiques de la machine de test principale.

Système d'exploitation	Windows 7 Pro SP1, 64bits
Processeur	Intel Core 2 Quad Q6600 2.40 Ghz
Mémoire vive	4 Go

Dans le cadre de cette expérimentation, on a analysé les résultats de 1400 parties, d'une durée d'environ 20 minutes de temps en jeu par partie, ce qui représente plus de 460 heures de temps en jeu pour SRbotOne. Le temps en jeu est le temps réel à vitesse normale du jeu (67ms par image), il ne représente pas le temps qu'ont nécessité les tests.

5.2.1 Résultats

Les résultats sont présentés dans le tableau 5.2. Les deux premières lignes donnent les nombres et pourcentages de victoires de SRbotOne, et les lignes 3 et 4 regroupent ses nombres et pourcentages de défaites. Quant aux deux dernières lignes, elles affichent les totaux. Les colonnes représentent de gauche à droite les cas où la partie s'est terminée avant que SRbotOne ne contre-attaque, les cas où la partie s'est terminée après une contre-attaque de SRbotOne, et les cas où SRbotOne a attaqué uniquement en fin de partie (situations dans lesquelles SRbotOne a choisi de ne pas contre-attaquer, mais d'attaquer délibérément son adversaire à un temps donné). Enfin la dernière colonne affiche également les totaux.

Avant d'analyser les données, on peut signaler qu'aucun plantage n'a été subit durant cette expérimentation. Considérant le nombre de parties et les heures de

jeu que cela représente, on peut tirer la conclusion que la version actuelle de SRbotOne est stable et robuste. De plus, il faut noter que Blizzard Entertainment ne fournit pas de détails sur l'implémentation de l'IA d'origine du jeu, ainsi on ne peut pas déterminer du niveau d'informations dont elle dispose. Celle-ci pourrait donc user de méthodes qu'on peut considérer comme de la triche à la place d'un joueur.

Tableau 5.2 Résultats de l'opposition à l'IA de StarCraft.

		Avant CA	Après CA	Lors A	Total
Victoires	#	0	444	101	545
	%	0 %	31.72 %	7.21 %	38.93 %
Défaites	#	113	393	349	855
	%	8.07 %	28.07 %	24.93 %	61.07 %
Total	#	393	793	214	1400
	%	8.07 %	59.79 %	32.14 %	100 %

On affiche 38.93% de victoire sur l'ensemble des parties. Les pourcentages sont répartis entre le moment où l'on choisit de contre-attaquer avant la période de fin de partie (31.72%), et le moment où l'on décide d'attaquer alors que l'ennemi attend la fin de partie (7.21%). Logiquement, le pourcentage de victoire avant la contre-attaque est de zéro, étant donné qu'il n'est pas possible de gagner contre l'intelligence artificielle du jeu sans passer à l'offensive durant la partie.

Les chiffres montrent également que SRbotOne adopte une tendance assez nette à contre-attaquer l'IA plutôt que d'attendre la fin de partie. Cela signifie que les conditions qui entraînent la contre-attaque (nombre d'attaques subies, nombre pertes de l'ennemi, etc.) sont souvent atteintes lorsqu'on joue contre l'IA de base de StarCraft.

Pour juger de la qualité du système de décision mis en place, comparons les pourcentages de victoire aux pourcentages de défaite. On remarque que la large majorité des victoires est obtenue avec la décision de contre-attaquer. Proportionnellement, cela représente $31.72/38.93 = 81.47\%$ des victoires pour cette décision. En mettant ces chiffres en parallèle avec leurs équivalents pour les défaites, on peut opposer les 81.47% de victoire proportionnelle résultant du choix de contre-attaquer aux $28.07/61.07 = 45.96\%$ de défaite proportionnelle pour le même choix. On constate que cette décision apporte un ratio largement positif et, ainsi, on peut en déduire que SRbotOne identifie correctement la tendance à suivre. De plus, le système de décision en place est en accord avec cette tendance. On peut valider, grâce à ces données, le fait qu'il choisit globalement les décisions les plus pertinentes en fonction du déroulement des parties, et du comportement de l'IA de base du jeu. En effet, contre-attaquer apporte plus souvent la victoire que la défaite, et c'est le comportement majoritairement choisi par SRbotOne.

Enfin, on peut extraire une statistique intéressante qui exprime le nombre de pertes en ressources (minerai) de l'ennemi, grâce aux défenses que l'on produit, à partir duquel on gagne systématiquement une partie. Autrement dit, le nombre de ressources correspondant aux unités ennemies perdues dans la zone qu'on a choisi de défendre, à partir duquel la victoire est assurée à 100%. Lors de cette expérimentation, ce nombre s'élève à 14125 (de minerai), mais une telle quantité est atteinte un nombre relativement limité de fois, étant donné que les pertes moyennes sur l'ensemble des parties s'élèvent à 2220,22.

5.3 SRbotOne *vs* d'autres bots

La deuxième expérimentation consiste en un tournoi opposant SRbotOne aux 22 bots inscrits à la compétition AIIDE 2015. La majorité de ces 22 bots ont été développés dans des laboratoires de recherche en IA qui s'intéressent aux jeux. Le

code source et binaire de ces 22 bots est disponible en raison des règlements des compétitions majeures qui exigent aux participants de publier leur code source après chaque compétition.

Les compétitions sont gérées grâce au *StarCraft AI Tournament Manager* développé en Java par David Churchill (développeur d'UAlbertaBot). Cet outil permet l'automatisation et la gestion des parties, le contrôle des règles du jeu et la collecte des résultats.

En raison des limites techniques de StarCraft et BWAPI, il est impossible de lancer simultanément deux bots dans une même instance de (machine) Windows. Ainsi, pour jouer une partie entre deux bots, il faut au moins deux machines Windows, reliées en réseau, chacune exécutant un bot. Plutôt que d'utiliser des machines virtuelles sur une même machine physique, on a choisi de lancer les expérimentions sur deux PCs aux spécifications similaires. De ce fait, on utilise la même machine que pour la première expérimentation (tableau 5.1) et une seconde avec le même système d'exploitation, un processeur Intel Core 2 Duo E8500 3.16 Ghz et 4 Go de mémoire vive. L'outil gérant le tournoi (*StarCraft AI Tournament Manager*) assigne aléatoirement les PCs aux bots pour chaque partie. Cela élimine tout biais qui aurait pu exister dans le cas où l'un des deux PC serait plus performant que l'autre.

On utilise la même version de StarCraft sur chacun des ordinateurs (1.16.1). Les versions employées de BWAPI et BWTA dépendent de chaque bot. Certains bots sont développés avec BWAPI 3.74, 4.01, ou 4.1.2 comme SRbotOne. Pour BWTA, l'utilisation est libre et propre à chacun, comme pour la première expérimentation, SRbotOne utilise la version 2.2.

5.3.1 Règlement du tournoi

Pour ce tournoi, le type de partie est toujours un contre un (un bot contre un autre bot). Chaque bot spécifie sa race pour qu'elle soit choisie par le programme (un faible nombre joue une race aléatoire), et le point de départ des joueurs en partie est choisi aléatoirement. Les cartes utilisées sont les mêmes que celles de la compétition AIIDE 2015, soit 3 cartes à 2 joueurs, 2 cartes à 3 joueurs, et 5 cartes à 4 joueurs. De plus, dans le but de tester SRbotOne en conditions de compétition officielle, on applique le règlement mis en place pour le tournoi de la conférence CIG 2016.

StarCraft AI Tournament Manager vérifie le respect du règlement et impose par conséquent une limite de 86400 images par partie, ce qui correspond à une heure de jeu (1 heure = 3600s, et on joue a 24 images par secondes, ce qui donne 24*3600=86400). Si une partie se prolonge jusqu'à cette durée, elle est arrêtée et le gagnant est déterminé par le score calculé par le jeu. En effet, à la fin d'une partie, chaque joueur se voit attribuer un score qui est déterminé par son utilisation des ressources, les unités produites et tuées (ennemies), ainsi que les bâtiments produits et détruits (ennemis).

D'autre part, un bot ne devrait pas ralentir le jeu, une limite de 42ms par image est imposée. Si cette limite est dépassée un nombre trop élevé de fois, le bot est déclaré perdant et la victoire est créditée à son adversaire. Ainsi, pour le tournoi de l'expérimentation, comme pour la compétition CIG 2016, un bot est considéré perdant pour une partie si : 200 images ou plus excèdent 55ms, 10 images ou plus excèdent 1 seconde, ou 2 images ou plus dépassent 10 secondes.

Évidemment, la triche est interdite. Les bots ne doivent pas utiliser les options de BWAPI qui permettent l'accès à toutes les informations du jeu (information parfaite). Tout bot au comportement malicieux est disqualifié. Dans le même ordre

d'idée, un plantage du jeu en cours de partie est considéré comme une défaite pour le système responsable. Mettre le jeu en pause est aussi interdit et donne la victoire à l'adversaire. Le règlement autorise l'exploitation de certains bugs spécifiques et connus du jeu (empilement d'unités volantes, etc.), et d'autres sont formellement interdits (mines alliées, empilement d'unités terrestres, etc.).

Enfin, le tournoi est de type *toutes rondes* (*round-robin* ou *all-play-all*). C'est-à-dire que les participants se rencontrent tous un nombre égal de fois, et tous une fois sur chaque carte. Dans le cadre de l'expérimentation, avec 23 bots et 10 cartes, chaque bot joue ainsi 220 parties, ce qui donne (23*220)/2 = 2530 parties pour le tournoi. Dans la section suivante, on présente les résultats de 2529 parties en raison d'un bug connu du programme qui ne lance pas la dernière partie prévue.

5.3.2 Résultats

Le tableau 5.3 regroupe les statistiques générées à la fin du tournoi par *StarCraft AI Tournament Manager*. Il est ordonné par rapport au pourcentage de victoire dans l'ordre décroissant. Il présente notamment la durée moyenne des parties (en minutes), le nombre de parties par bot qui a atteint la limite de temps (1 heure), le nombre de plantages par bot et le nombre de parties où le bot a été disqualifié pour avoir trop ralenti le jeu.

SRbotOne se classe $14^{ème}$ sur 23 avec 48.18% de victoire sur 220 parties. On peut constater que la durée moyenne de partie pour les meilleurs est très courte, ce qui est logique étant donné que la majorité d'entre eux emploie une stratégie de *rush* (les résultats en prouvent l'efficacité). Cependant, ayant basé notre stratégie sur la défense, on a au contraire l'objectif de faire durer la partie plus longtemps. Or, SRbotOne est deuxième dans la liste en termes de durée moyenne de partie, par rapport au pourcentage de victoire (17 minutes et 8 secondes en moyenne). De plus, si on compare avec UAlbertaBot sur lequel on s'est basé, le temps moyen de

partie est augmenté considérablement.

Tableau 5.3 Résultats de l'opposition à d'autres bots (tournoi).

Bot	Parties	Victoires	Défaites	% Victoire	TempsMoy	Heure	Plantage	Lenteur
ZZZKBot	219	200	19	91,32	6:06	0	1	0
UAlbertaBot	220	192	28	87,27	8:13	4	0	2
Overkill	220	184	36	83,64	9:50	1	0	1
Skynet	220	168	52	76,36	9:50	1	0	8
Xelnaga	220	168	52	76,36	14:24	10	10	0
IceBot	220	165	55	75	13:38	2	1	1
Aiur	220	162	58	73,64	11:50	3	0	1
LetaBot	220	153	67	69,55	9:54	2	2	0
TerranUAB	220	127	93	57,73	13:40	7	8	0
CruzBot	220	121	99	55	16:42	18	0	2
NUSBot	220	115	105	52,27	11:10	3	30	4
Cimex	220	111	109	50,45	16:30	12	8	3
OpprimoBot	220	108	112	49,09	18:05	0	11	0
SRbotOne	**220**	**106**	**114**	**48,18**	**17:08**	**4**	**0**	**9**
Stone	220	85	135	38,64	18:49	33	0	1
Oritaka	220	82	138	37,27	15:38	18	0	0
Bonjwa	220	78	142	35,45	19:15	32	0	0
Yarmouk	220	63	157	28,64	14:52	16	3	0
SusanooTricks	220	54	166	24,55	20:00	30	8	10
Tyr	220	38	182	17,27	17:07	39	0	0
tscmoo	219	18	201	8,22	16:24	36	0	0
GarmBot	220	16	204	7,27	16:51	37	0	0
Ximp	220	15	205	6,82	8:08	0	2	193
Total	2529	2529	2529	N/A	14:05	154	84	235

La couleur sur le nom des bots représente leur race, violet pour *Zerg*, jaune pour *Protoss*, bleu pour *Terran*, et gris pour aléatoire.

Comme lors de la première expérimentation, on remarque qu'aucun plantage n'est survenu cette fois-ci non plus, ce qui confirme la stabilité de la version actuelle de SRbotOne. D'autre part, les données montrent qu'on a atteint à quatre reprises la limite de l'heure de jeu. Après analyse des résultats détaillés, les quatre parties ont été comptées comme des défaites. Cela s'explique par le départage au score qui a été défavorable à chaque fois. La raison vient sans doute du fait que les

Marines, que l'on produit en masse, ne rapportent pas beaucoup de points par rapport aux unités de haut niveau que produisent les adversaires en fin de partie.

Si on s'intéresse à la colonne des disqualifications pour ralentissement du jeu (Lenteur), 9 parties ont été perdues. Sur 6 de ces 9 parties, SRbotOne avait pourtant l'avantage. Cela signifie que la disqualification s'est produite alors que le bot était en supériorité et allait gagner la partie. Le problème vient du fait qu'on disposait d'un trop grand nombre d'unités, et que la structure d'UAlbertaBot n'est pas conçue pour en gérer un si grand nombre, d'où les ralentissements. Étant donné que UAlbertaBot n'utilise que des stratégies de *rush*, il n'est pas conçu pour avoir un maximum d'unités et jouer des parties jusqu'à l'heure de jeu. On en montre en quelque sorte les limites avec notre stratégie, optimiser le système pour ces cas serait une piste intéressante. En effet, avec ces 6 victoires, le taux de victoire de SRbotOne passerait à plus de 50%, ce qui lui ferait gagner 2 places au classement.

Les résultats détaillés fournis par SRbotOne sont présentés dans le tableau 5.4. Le tableau adopte la même structure que le tableau 5.2, à l'exception de trois colonnes. Les colonnes « Lors A18 », « Lors A23 » et « Lors PT » représentent respectivement les cas où SRbotOne a attaqué à la 18^e minute (sans avoir décidé de contre-attaquer auparavant), SRbotOne a attaqué à la 23^e minute (sans avoir décidé de contre-attaquer auparavant), la partie a été interrompue en raison d'un plantage ou de trop nombreux ralentissements. Cela s'explique car, en tournoi, SRbotOne change quelque peu sa stratégie contre les adversaires connus pour attaquer en *rush*, d'où les deux colonnes pour l'attaque à 18 et 23 minutes (voir figure 4.5, *tAttaque*). Les statistiques de défaites lors d'un plantage ou d'un nombre trop important de ralentissements ne sont pas affichées pour SRbotOne, car elles sont prises en comptes dans les chiffres des autres colonnes.

Ici, les 48.18% de victoire sont répartis entre le moment où l'on choisit de contre-

attaquer avant la période de fin de partie (11.36%), le moment où l'on décide d'attaquer à la minute 18 (30.92%), le moment où l'on décide d'attaque après 23 minutes de jeu (0.45%), et les victoires par disqualification adverse (5.45%). Comme lors de la première expérimentation, le pourcentage de victoire avant la contre-attaque est de zéro.

Tableau 5.4 Résultats de l'opposition à d'autres bots (fichiers de journaux).

		Av CA	Ap CA	Lors A18	Lors A23	Lors PT	Total
V	#	0	25	68	1	12	106
	%	0 %	11.36 %	30.92 %	0.45 %	5.45 %	48.18 %
D	#	68	28	18	0	N/A	114
	%	30.91 %	12.73 %	8.18 %	0 %	N/A %	51.82 %
T	#	68	53	86	1	12	220
	%	30.91 %	24.09 %	39.1 %	0.45 %	5.45 %	100 %

Les chiffres montrent que SRbotOne adopte une tendance assez nette à attaquer plus tard et attendre la fin de partie, ce qui représente le scénario inverse à la première expérimentation contre l'IA originale de StarCraft. Cela signifie que les conditions qui entraînent la contre-attaque (nombre d'attaques subies, nombre pertes de l'ennemi, etc.) sont plus rarement atteintes lorsqu'on joue contre d'autres bots. On peut attribuer ce phénomène à la qualité des autres bots qui montrent une bien meilleure gestion des unités que l'IA de base du jeu (et subissent moins de pertes).

On remarque ici que la majorité des victoires est obtenue en combinant les décisions d'attendre la période de fin de partie (18^e et 23^e minutes), cela représente $30.92+0.45 = 31.37\%$. Proportionnellement, cela représente $31.37/48.18 = 65.10\%$ des victoires pour la décision d'attendre la période de fin de partie. En mettant

ces chiffres en parallèle avec leurs équivalents pour les défaites, on peut opposer les 65.10% de victoire proportionnelle résultant du choix d'attendre la période de fin de partie, aux 8.18/51.32 = 15.79% de défaite proportionnelle pour les mêmes choix. On constate que cette décision apporte un ratio largement positif, ainsi on peut en déduire que SRbotOne identifie correctement la tendance à suivre. De plus, le système de décision en place est en accord avec cette tendance. On peut une nouvelle fois valider, grâce à ces données, le fait qu'il choisit globalement les décisions les plus pertinentes, en fonction du déroulement des parties et du comportement de son adversaire. En effet, lors de cette expérimentation, attendre la période de fin de partie apporte plus souvent la victoire que la défaite, et c'est le comportement majoritairement choisi par SRbotOne.

Enfin, les statistiques de défaites indiquent un nombre important de 30.91% de parties perdues avant même d'avoir pu contre-attaquer. Cette donnée peut montrer deux choses. Tout d'abord, un grand nombre d'adversaires privilégient effectivement les stratégies de *rush* en début de partie, et par conséquent empêchent SRbotOne d'installer ses défenses. Et ensuite, que dans certains cas, les défenses de SRbotOne en début de partie n'arrivent pas à résister aux attaques ennemies, ce qui entraîne sa défaite.

Pour finir, on extrait à nouveau la statistique qui exprime le nombre de pertes en ressources (minerai) de l'ennemi grâce aux défenses que l'on produit, à partir duquel on gagne systématiquement une partie. Lors de cette expérimentation, ce nombre s'élève à 6750 (de minerai), mais comme la première fois, cette quantité est atteinte un nombre relativement limité de fois. Les pertes moyennes adverses sur l'ensemble des parties s'élèvent à 572.

5.4 Compétition CIG 2016

SRbotOne a participé à la compétition officielle d'IA pour StarCraft, organisée par l'université de Sejong (Corée du Sud), dans le cadre de la conférence IEEE CIG 2016. Les résultats de la compétition ont été dévoilés lors de la conférence, du 20 au 23 septembre 2016, à Santorini en Grèce. Pour l'année 2016, 16 bots ont été soumis pour participer en provenance de 13 pays.

Parmi ces 16 participants, 7 sont des entrées ayant concouru à une compétition en 2015 et qui n'ont pas subi de modifications. Les 9 autres sont des bots dont la version a été mise à jour, ou de nouveaux bots. On peut noter que SRbotOne est la seule entrée complètement nouvelle (bot et développeur) ; en effet, d'autres bots sont nouveaux, mais leurs développeurs ont déjà participé à des compétitions antérieures. SRbotOne est également le seul bot à adopter une stratégie défensive et d'attente, tous les autres participants ont une stratégie globale offensive et agressive (5 d'entre eux emploient le *rush* principalement).

Malheureusement, on ne dispose pas des informations concernant le matériel et les spécifications des machines utilisées pour les parties. Cependant, le règlement utilisé est le même que celui de l'expérimentation 2 (section 5.3.1). L'ensemble de cartes utilisé pour la compétition est composé de 1 carte à deux joueurs, 2 cartes à trois joueurs, et 2 cartes à quatre joueurs.

Le tournoi est organisé avec le programme *StarCraft AI Tournament Manager* et se déroule en deux phases. La première est une phase qualificative pour sélectionner la meilleure moitié des participants, en se basant sur le pourcentage de victoire. Dans la deuxième phase, seule la meilleure moitié des bots joue pour former un classement final. Les résultats de la première phase ne sont pas utilisés pour la phase finale, et le classement final est basé uniquement sur la seconde phase de compétition. Les deux étapes de la compétition se déroulent en mode

toutes rondes (*round-robin*), où tous les participants s'affrontent un nombre de fois égal et équivalent sur chaque carte. Pour la compétition CIG de l'année 2016, environ 1500 parties par participant ont été jouées lors de l'étape qualificative, et 700 parties par bot en phase finale, soit un total de 14800 parties de StarCraft.

5.4.1 Résultats

Le tableau 5.5 présente les résultats générés par le programme gestionnaire du tournoi à la fin du tour qualificatif, comme ce fut le cas lors de l'expérimentation 2 (tableau 5.3). SRbotOne se classe 12^e sur 16 participants avec 22.15% de victoire sur 1499 parties. Ce score ne lui permet malheureusement pas de se qualifier pour la deuxième phase de la compétition. Il est assez loin, puisque la dernière place du top 8 se jouait aux alentours des 63.15%. Par rapport à la deuxième expérimentation, on peut constater que la durée moyenne de partie pour les meilleurs est sensiblement plus longue (plus de 13 minutes contre environ 8 minutes). Cependant, si on observe le classement en termes de durée moyenne de partie par rapport au pourcentage de victoire, SRbotOne conserve sa deuxième place (derrière Tyr avec 17 minutes et 19 secondes pour 61.64% de victoire). De plus, en comparaison avec UAlbertaBot, la durée moyenne des parties est toujours bien plus longue.

Pour la première fois, 3 plantages sont attribués à SRbotOne. Leur raison n'est pas encore connue, mais ce nombre est tolérable en comparaison aux autres participants. De plus, les chiffres montrent que cette fois-ci on a atteint à 19 reprises l'heure de jeu. L'analyse de ces parties en détails montre que 15 défaites ont été comptabilisées pour seulement 4 victoires par départage au score. Cette situation confirme l'hypothèse émise lors de l'expérimentation précédente (manque de points apportés par les unités produites) et démontre une lacune de la stratégie actuelle.

Tableau 5.5 Résultats du tour qualificatif de la compétition CIG 2016.

Bot	Parties	Victoires	Défaites	% Victoire	TempsMoy	Heure	Plantage	Lenteur
Iron	1500	1188	312	79.2	14:11	30	22	7
tscmoo	1498	1153	345	76.97	14:15	8	27	0
LetaBot	1500	1111	389	74.07	13:34	38	15	0
Overkill	1499	1064	435	70.98	11:37	16	22	0
MegaBot	1499	1051	448	70.11	11:44	18	102	4
UAlbertaBot	1499	1038	461	69.25	9:59	30	0	0
ZZZKBot	1499	1037	462	69.18	7:23	0	0	0
Aiur	1498	946	552	63.15	13:20	32	7	0
Tyr	1499	924	575	61.64	17:19	77	25	0
Ziabot	1497	695	802	46.43	9:56	21	183	114
TerranUAB	1498	502	996	33.51	14:11	18	72	1
SRbotOne	**1499**	**332**	**1167**	**22.15**	**14:45**	**19**	**3**	**6**
OpprimoBot	1498	331	1167	22.1	16:46	0	204	0
XelnagaII	1497	310	1187	20.71	13:34	51	677	226
Bonjwa	1499	284	1215	18.95	12:05	32	1	0
Salsa	1497	22	1475	1.47	10:53	0	480	119
Total	11988	11988	11988	N/A	12:51	195	1840	477

La couleur sur le nom des bots représente leur race, violet pour *Zerg*, jaune pour *Protoss*, bleu pour *Terran*, et gris pour aléatoire.

Concernant les disqualifications pour ralentissement, on en recense 6, dont 2 dans lesquelles SRbotOne avait l'avantage. On semble à nouveau se heurter au problème du nombre trop important de troupes, qui perturbe les systèmes mis en place par UAlbertaBot. Cela permet d'identifier un autre point à améliorer pour le bot.

Le tableau 5.6 est construit de la même façon que le tableau 5.4 de la section 5.3.2 et affiche les statistiques obtenues avec l'exploitation des fichiers de journaux générés par SRbotOne. Les statistiques de défaites lors d'un plantage ou d'un nombre trop important de ralentissements ne sont pas affichées pour SRbotOne, car elles sont prises en comptes dans les chiffres des autres colonnes.

Les 22.15% de victoire sont répartis entre le moment où l'on choisit de contre-

Tableau 5.6 Résultats du tour qualificatif de la compétition CIG 2016 (fichiers de journaux).

		Av CA	Ap CA	Lors A18	Lors A23	Lors PT	Total
V	#	0	55	218	12	47	332
	%	0 %	3.67 %	14.54 %	0.8 %	3.14 %	22.15 %
D	#	903	150	113	1	N/A	1167
	%	60.24 %	10.01 %	7.54 %	0.06 %	N/A	77.85 %
T	#	903	205	331	13	47	1499
	%	60.24 %	13.68 %	22.08 %	0.86 %	3.14 %	100 %

attaquer avant la période de fin de partie (3.67%), le moment où l'on décide d'attaquer à la minute 18 (14.54%), le moment où l'on décide d'attaque après 23 minutes de jeu (0.80%), et les victoires par disqualification adverse (3.14%). Les statistiques sont plus faibles mais respectent la même séparation que celle observée lors du premier test contre les autres bots des chercheurs (section 5.3.2). La majorité est obtenue après une attaque tardive et en fin de partie, un nombre plus faible en contre-attaque directe, et enfin le reste par les dysfonctionnements adverses. SRbotOne adopte à nouveau une tendance à attaquer plus tard et attendre la fin de partie.

L'analyse des données amène à tirer les mêmes conclusions par rapport au système de prise de décision de SRbotOne. On remarque que la majorité des victoires est obtenue en combinant les décisions d'attendre la période de fin de partie (18^e et 23^e minutes), cela représente $14.54 + 0.8 = 15.34\%$. Proportionnellement, cela représente $15.34/22.15 = 69.25\%$ des victoires pour la décision d'attendre la période de fin de partie. En mettant ces chiffres en parallèle avec leurs équivalents pour les défaites, on peut opposer les 69.25% de victoire proportionnelle résultant du choix

d'attendre la période de fin de partie aux $7.6/77.85 = 9.77\%$ de défaite proportionnelle pour les mêmes choix. Cette décision apporte un ratio largement positif et, ainsi, on peut en déduire que SRbotOne identifie correctement la tendance à suivre. De plus, on remarque de façon intéressante que les 69.25% et 9.77% sont du même ordre de grandeur que les 65.10% et 15.79% de la deuxième expérimentation. Attendre la période de fin de partie apporte plus souvent la victoire que la défaite, et c'est le comportement majoritairement choisi par le bot.

Enfin, SRbotOne a perdu 60.24% de parties avant même de passer à l'offensive dans cette compétition. Soit le double du chiffre observé lors de l'expérience numéro 2 (section 5.3.2). Les enseignements sont les mêmes, et l'écart conséquent montre un autre aspect des compétitions entre bots dans StarCraft. Les autres bots écrivent des fichiers de journaux après chaque partie, et surtout les réutilisent à chaque début de jeu pour adapter leur stratégie. La plupart implémente plusieurs stratégies et retient les résultats de chacune contre ses adversaires. Ainsi, dès le moment où une certaine stratégie fonctionne contre un adversaire, le bot l'appliquera systématiquement. Le problème est que SRbotOne ne fait actuellement pas le même effort, et qu'en plus il n'implémente qu'une seule stratégie principale. Or, les adversaires qui attaquent rapidement en début de partie obtiennent de bons résultats contre SRbotOne. Par conséquent, l'analyse des fichiers de journaux les conduit logiquement à tendre vers une utilisation systématique de cette stratégie. On peut malheureusement imaginer que plus le nombre de parties augmente, plus les bots s'adaptent et plus le nombre de défaites pour ces raisons augmente. Finalement, on n'obtient pas de seuil de pertes ennemies en ressources à partir duquel la victoire est assurée à 100% cette fois-ci. Les pertes moyennes adverses sur l'ensemble des parties s'élèvent à 410.71.

Les résultats détaillés de la compétition CIG 2016 sont disponibles à cette adresse : `https://sites.google.com/site/starcraftaic/result`.

5.5 Analyse globale des résultats

D'une manière générale, on peut voir que les bots développés par les chercheurs ont un comportement plus agressif que l'IA originale de StarCraft. Les résultats indiquent clairement que les bots ont de meilleures capacités de micro et sont plus efficaces. La question de déterminer qui de l'IA originale ou des bots des compétitions est la meilleure solution reste ouverte à débat. Effectivement, les meilleurs agents intelligents sont supérieurs à l'IA du jeu en opposition directe, mais il ne faut pas oublier que leurs objectifs sont différents. L'IA du jeu doit apporter au joueur une bonne expérience de jeu en lui offrant une opposition, alors que les bots de compétition sont conçus spécialement pour gagner un maximum de parties, et ce contre d'autres bots. Un humain n'appréciera pas le jeu contre un bot qui *rush* rapidement et ne le laisse pas jouer. Ainsi, le choix de la meilleure solution entre les deux est plus compliqué.

D'autre part, StarCraft étant un jeu relativement ancien, il a des pré-requis complètement insignifiants pour les PCs d'aujourd'hui (processeur Pentium 90 MHz, 16 Mo de RAM). Mais l'utilisation de techniques et algorithmes d'intelligence artificielle (comme A*) au sein du jeu peut amener à s'interroger sur l'importance de la configuration matérielle des machines et de son impact. Après avoir testé SRbotOne sur les machines de l'expérimentation 2 (section 5.3) et sur une troisième configuration (Intel Core i7-4710HQ 2.5 Ghz, 8 Go de RAM), on ne constate pas de différences significatives. Pour 200 parties effectuées sur chaque machine dans les mêmes conditions de tournoi, on note une variation de quelques points de pourcentage du nombre de victoires uniquement. Ainsi, on ne peut pas tirer de réelle conclusion ou vérifier une influence importante qu'on pourrait attribuer à la puissance disponible.

SRbotOne base sa stratégie sur la fortification d'une région de la carte de jeu, ce

choix crée une dépendance par rapport aux cartes. En effet, pour chaque carte et même chaque point de départ des cartes, la région choisie varie. Certaines n'ont pas un espace 100% constructible, et peuvent même avoir une topographie qui empêche de construire des lignes de fortification pour former un mur qui bloque le passage. Ainsi, l'efficacité et la robustesse d'une stratégie qui y accorde une telle importance offre des résultats différents sur chaque carte. Les tableaux 5.7, 5.8, 5.9 et 5.10 visibles à la fin du chapitre donnent des statistiques pour chaque carte, en fonction des trois expérimentations effectuées.

En observant les tableaux 5.9 et 5.10, qui représentent respectivement les expérimentations 2 et 3, on peut constater (en retirant les extrêmes) que les résultats pour chaque carte sont relativement rapprochés. De plus, ils sont également très proches et représentatifs des 48.18% et 22.15% respectifs de victoire globale des expérimentations 2 et 3. Ceci signifie que le système en place au sein de SRbotOne est performant et fiable, puisqu'il s'adapte bien et apporte des résultats homogènes, malgré la variation des cartes.

Pour l'expérimentation 1, les tableaux 5.7, 5.8 affichent une répartition plus importante. Cela peut s'expliquer par le nombre plus important de cartes pour cette expérience, mais indique tout de même que des cas particuliers sont encore difficiles à gérer par le système (problèmes de géométrie des zones non-constructibles pour certaines région). La carte *Plasma1.0*, pour laquelle on obtient 0% de victoire n'est pas à prendre en compte pour cette statistique. En effet, c'est une carte spéciale dont les passages entre régions sont bloqués, ce qui ne permet pas d'appliquer la stratégie de SRbotOne. De ce fait, la victoire est départagée au score, et le bot souffre du faible apport en points de ses productions (problème évoqué à la section 5.3.2).

Globalement, on peut conclure que le système en place fonctionne et apporte des

résultats prometteurs. Il reste perfectible, en raison du problème de placement des bâtiments en fonction des zones non constructibles. Mais les résultats par rapport aux cartes ne sont pas à attribuer uniquement à ce système, ils dépendent aussi de la stratégie et de la micro ennemie. De plus, on démontre avec ces résultats qu'il s'adapte bien à chaque carte. En effet, la version actuelle est stable. Elle a l'avantage de pouvoir jouer différemment sur n'importe quelle carte, contrairement à d'autres bots qui ont souvent des comportements spécifiques pour chacune. Or, on remarque un nombre important de plantages sur ces bots, parce qu'ils ne sont simplement pas compatibles avec certaines cartes de jeu.

Les statistiques de pertes moyennes ennemies dans la région fortifiée n'apportent pas beaucoup d'informations. Les pertes moyennes ne semblent pas être fortement liées au pourcentage de victoire, le plus grand nombre ne correspond pas au plus grand pourcentage. Cependant, on remarque tout de même que pour les pourcentages les plus faibles, les pertes moyennes tendent à être dans les plus faibles selon les expérimentations et en dessous de leur moyenne générale. On rappelle que les pertes prises en compte sont uniquement les pertes lors de fronts (groupes d'au moins 3 unités), dans la région sécurisée. De ce fait, les unités solitaires ou groupes de 2 ne sont pas comptées dans ces statistiques.

Concernant la prise de décision à propos du moment propice pour contre-attaquer avant la période de fin de partie, ou pour attaquer après, les trois expérimentations ont montré de bons résultats. Le bot est capable d'identifier la tendance à suivre et de s'adapter à son adversaire. Ses choix sont corrects et pertinents, ils apportent la victoire en majorité, comme le montrent les résultats. Ainsi, on peut en conclure que les objectifs de montrer qu'une stratégie basée sur la raisonnement spatial et de valider le système de décision pour le passage à l'action sont atteints. Malheureusement, si le pourcentage de victoire n'est pas plus important, c'est à cause d'autres lacunes de SRbotOne, que les statistiques font apparaître.

5.5.1 Principales lacunes et pistes d'évolution

Quatre lacunes principales ressortent des données et observations recueillies, celles-ci représentent des pistes d'évolution et améliorations à apporter à SRbotOne.

Premièrement, les résultats montrent un problème important de résistance aux stratégies de *rush*. Ces attaques rapides empêchent le bot d'installer une ligne de défense suffisante. Les parties sont perdues avant même que le système de contre-attaque ne soit en place et actif. Dans ces cas, le bot ne peut développer sa stratégie. Une solution est d'optimiser encore le début de partie de SRbotOne pour qu'il soit plus efficace et s'installe plus vite sur la carte. D'autre part, on peut imaginer un début de partie en deux étapes : tout d'abord utiliser du *walling* pour contrer les *rushs*, puis sécuriser une région et suivre la stratégie initiale.

Deuxièmement, on rencontre des ralentissements qui peuvent entraîner la disqualification. On a identifié le simulateur de combat inclus dans UAlbertaBot comme source du problème. En effet, on simule chaque bataille pour estimer les chances de victoires et donner l'ordre de repli au besoin. Cependant, UAlbertaBot effectuant majoritairement des stratégies de *rush*, il n'est pas conçu pour la simulation de grosses armées en fin de partie. D'où les ralentissements, à cause du temps de calcul plus long. Pour contrer ce problème dans la version actuelle de SRbotOne, on arrête d'utiliser le simulateur à partir d'un certain nombre d'unités engagées au combat. Cela permet de réduire les ralentissements en général, mais en conséquence, on ne gère plus les unités de façon efficace. En effet, elles sont envoyées au combat et attaquent jusqu'à mourir, sans plus de réflexion. Ce manque de gestion entraîne malheureusement de nombreuses défaites.

Une solution plus viable est de réviser les parties de la structure de UAlbertaBot concernées, afin de pouvoir supporter plus d'unités, sans causer de ralentissement et ainsi conserver le simulateur actif. D'autre part, on peut aussi envisager de

changer la stratégie concernant la production des unités en fin de partie. L'idée est de créer un nombre d'unités moins important, mais de plus haut niveau que les *Marines*, qui sont les unités de base, pour éviter la surcharge.

Ensuite, pour poursuivre dans le sens de la dernière solution proposée, les résultats démontrent que si la stratégie de défense est bonne, une lacune se trouve dans la stratégie offensive du bot. En effet, lorsqu'on décide de contre-attaquer ou d'attaquer, on envoie toutes nos troupes à l'assaut vers la base ennemie. Celles que l'on continue de produire y partent également une par une, sans plus de gestion. Une meilleure solution serait d'étoffer la stratégie offensive, pour lui donner plus de diversité et d'efficacité.

Par exemple, on pourrait utiliser du raisonnement spatial pour le micromanagement (champs de potentiel, cartes d'influences), dont les résultats ont déjà été montrés dans la littérature. De plus, l'idée de changer la composition de l'armée pour des unités plus puissantes (et qui rapportent plus de points), peut y être combinée pour résoudre le problème du départage au score à l'heure de jeu, qui est souvent défavorable à SRbotOne.

Enfin, on a pu constater que l'exploitation des fichiers de journaux à des fins d'adaptation stratégiques est un élément crucial. SRbotOne ne les utilise que pour enregistrer des statistiques, contrairement aux autres bots. Or, c'est clairement une lacune pour les compétitions. Afin d'améliorer les résultats, diversifier les stratégies d'attaque du bot, et choisir la plus efficace par rapport à chaque adversaire, selon les résultats lors de précédentes rencontres contre lui, est une idée à développer. Observer l'évolution du pourcentage de victoire du bot au tournoi SCCAIT[1] ne fait que confirmer cette hypothèse.

---

1. `http://sscaitournament.com/index.php?action=scores`

SSCAIT est le tournoi permanent en parallèle aux compétitions CIG et AIIDE (section 2.3). Peu après son inscription, le bot affichait 40% de victoire. Le pourcentage à diminué au fil du temps pour atteindre 24% (au moment de la rédaction). Étant donné que les adversaires apprennent petit à petit les stratégies et points faibles du bot, ses chances de victoire ne font que baisser.

5.5.2 Comparaison avec UAlbertaBot

Après toutes ces expérimentations, on peut comparer les données avec celles d'UAlbertaBot. Opposer les deux bots par rapport au pourcentage de victoire est difficile, car peu pertinent en raison de l'approche adoptée pour SRbotOne. UAlbertaBot est supérieur, il en va de même pour les confrontations directes. En effet, l'objectif principal de SRbotOne n'est pas d'obtenir un meilleur pourcentage, ou de battre celui sur lequel il est basé. Son but premier est d'explorer la possibilité d'utiliser du raisonnement spatial, pour la prise de décision stratégique et la stratégie de haut niveau. Ainsi, différencier les deux bots sur les critères statistiques principaux ne permet pas de prouver la valeur de SRbotOne. La statistique intéressante à observer est celle du temps moyen de la durée des parties.

Les parties de SRbotOne durent plus longtemps que celles d'UAlbertaBot, ce qui illustre leur différence de stratégie. Le second emploie une stratégie de *rush* pour finir la partie rapidement, alors que le premier cherche à faire durer la partie dans une stratégie plus complexe. Ce changement de concept ainsi que les modifications apportés au code initial (gestion poussée des bases, etc.), en plus des lacunes constatées pour SRbotOne dans sa version actuelle, rendent une impression moyenne à partir des résultats, par rapport à un bot développé depuis plusieurs années. En réalité, pour un premier pas, les résultats sont positifs et encourageants. Avec une version plus complète de SRbotOne, reprenant les points à améliorer, on pourra comparer à nouveau avec UAlbertaBot.

Tableau 5.7 Statistiques des cartes de l'expérimentation 1 (1/2).

Carte	Victoires (%)	Pertes ennemies (moy)
(4)EmpireoftheSun	70.83	1439.06
(2)Benzene	58.33	2519.08
(2)MatchPoint1.3	55.81	2153.53
(3)TauCross	53.19	3404.53
(3)Aztec	52.38	2223.80
(3)TauCross1.1	50.90	2780.89
(3)NeoAztec2.1	49.12	2505.35
(4)NeoSniperRidge2.0	49.09	2858.36
(2)Destination	46.93	2368.44
(2)HeartbreakRidge	46.80	2280.72
(3)Alchemist1.0	42.22	2331.6
(4)CircuitBreaker	42.00	2212.58
(4)FightingSpirit1.3	41.50	2570.67
(2)NeoChupungRyeong2.1	41.30	2720.15
(2)Hitchhiker1.1	41.17	2410.41
(4)LunaTheFinal2.3	40.54	1952.00
(2)Destination1.1	40.00	2717.95
(4)Andromeda	37.50	2570.25
(2)NeoHeartbreakerRidge	36.36	2242.43
(4)CircuitBreakers1.0	34.78	2407.02
(3)Pathfinder1.0	32.35	1836.73
(4)Python1.3	31.11	1933.84
(3)GreatBarrierReef1.0	30.23	1655.39

Le tableau est trié dans l'ordre décroissant du pourcentage de victoire. Le chiffre entre parenthèses devant le nom d'une carte indique le nombre de points de départ.

Tableau 5.8 Statistiques des cartes de l'expérimentation 1 (2/2).

Carte	Victoires (%)	Pertes ennemies (moy)
(2)RideofValkyries1.0	29.41	1812.52
(4)ArcadiaII2.02	28.88	2539.62
(4)Andromeda1.0	21.05	2584.17
(4)Fortress	20.83	1721.18
(4)Python	20.00	2700.85
(2)BlueStorm1.2	12.00	483.54
(3)Plasma1.0	0.00	0.00

Suite du tableau 4.5, trié dans l'ordre décroissant du pourcentage de victoire.

Tableau 5.9 Statistiques des cartes de l'expérimentation 2.

Carte	Victoires (%)	Pertes ennemies (moy)
(2)HeartbreakRidge	59.09	587.63
(4)EmpireoftheSun	54.54	551.13
(2)Benzene	50.00	890.09
(4)CircuitBreaker	50.00	796.59
(4)Andromeda	50.00	736.68
(3)TauCross	50.00	476.45
(2)Destination	50.00	438.72
(3)Aztec	45.45	802.40
(4)Fortress	40.90	107.95
(4)Python	31.81	333.00

Le tableau est trié dans l'ordre décroissant du pourcentage de victoire. Le chiffre entre parenthèses devant le nom d'une carte indique le nombre de points de départ.

Tableau 5.10 Statistiques des cartes de l'expérimentation 3.

Carte	Victoires (%)	Pertes ennemies (moy)
(2)RideofValkyries	24.33	260.29
(3)Alchemist	23.66	483.78
(4)Python	21.07	431.02
(3)TauCross	21.00	558.33
(4)LunaTheFinal	20.66	320.17

Le tableau est trié dans l'ordre décroissant du pourcentage de victoire. Le chiffre entre parenthèses devant le nom d'une carte indique le nombre de points de départ.

CONCLUSION

Dans ce mémoire, on a présenté une approche destinée à donner une plus grande importance sur le plan stratégique au raisonnement spatial dans un jeu de stratégie en temps réel. L'objectif est d'améliorer l'adaptabilité et l'expérience de jeu offerte par les bots de ce genre de jeux. Cette approche consiste à construire la stratégie pour une partie en fonction du raisonnement spatial effectué sur la carte. Un système de décision, basé sur des techniques d'intelligence artificielle, permet de générer une stratégie adaptée aux conditions de chaque environnement.

Dans le but d'expérimenter et d'observer la pertinence de l'idée, on a développé un bot, c'est-à-dire une intelligence artificielle, capable de jouer au jeu vidéo StarCraft. En tant que jeu de stratégie militaire en temps réel, StarCraft propose des environnements et possibilités d'interaction variés et complexes. Il représente ainsi un banc d'essai intéressant pour illustrer l'efficacité de l'approche proposée. En effet, c'est un jeu à mouvement simultané dont les actions sont non déterministes et à durée variable, et l'état du jeu est partiellement observable. De plus, les cartes de jeu sont de tailles, topologies et configurations diverses (surfaces non traversables ou non constructibles, multiples points de départ, etc.). Notre bot SRbotOne est fondé sur la structure d'UAlbertaBot, un autre bot de la communauté qui a servi de base.

L'approche proposée dans ce mémoire a été testée en contexte réel de jeu, grâce à trois expérimentations décrites dans le chapitre 5, dont la participation à une compétition officielle de bots pour StarCraft, soit celle organisée dans le cadre de la conférence IEEE CIG 2016. L'implémentation de l'approche au sein de SRbotOne

offre des résultats encourageants. Ces résultats démontrent que l'emploi du raisonnement spatial pour la prise de décision stratégique est une piste prometteuse. Les décisions prises par le bot sont pertinentes et montrent une forte capacité d'adaptabilité en fonction des 30 cartes du corpus d'expérimentation. On le constate en comparaison aux autres agents intelligents, notamment grâce à la statistique du nombre de plantages et problèmes techniques. En effet, SRbotOne n'intègre aucun comportement ou composant spécifique, fonctionnant avec une carte précise uniquement. Au contraire, il est indépendant vis-à-vis des environnements et s'y adapte, ce qui le rend compatible à n'importe quelle carte.

L'approche consiste à calculer le chemin traversable au sol le plus direct, entre les deux bases principales des joueurs en début de partie. Ce chemin est un passage obligatoire pour l'adversaire durant la partie, lorsqu'il poursuit son objectif de victoire et cherche à attaquer la base principale qu'on possède.

Sur ce chemin, un système de règles choisit et fortifie la région la plus avantageuse et prometteuse à la réussite de l'application de la stratégie. Ce système sélectionne une région qui n'est pas trop éloignée de la base, qui a un maximum de voies d'accès, et qui dispose d'un ratio *taille/espace constructible* suffisant pour être sécurisable et résistante aux assauts ennemis.

De plus, contrairement à plusieurs autres bots existants qui favorisent des attaques rapides, SRbotOne favorise l'attente. En effet, il laisse l'adversaire attaquer en premier et mise ainsi sur la force de ses défenses. Il décide par lui-même, lorsque le moment propice se présente, de contre-attaquer ou de lancer une attaque sur son opposant. SRbotOne incite l'adversaire à investir et à perdre des ressources, pour l'affaiblir et le frapper lorsque la situation tourne à son désavantage.

D'autre part, SRbotOne se distingue d'autres bots dont UAlbertaBot par la mise en place d'un écosystème de gestion des bases. Fondé sur un modèle de bases per-

sonnalisées et une chaîne logistique, notre approche permet la création de bases autonomes et spécialisées dans certaines fonctions. Effectivement, on peut envisager différents types de bases qui passent elles-mêmes leurs commandes en bâtiments et unités selon leurs besoins.

En tant que première itération, la version actuelle de SRbotOne n'affiche pas des performances à la hauteur des meilleurs bots existants, du moins pas en termes de pourcentage de victoire. L'objectif d'intégration des concepts explicités auparavant a relayé au second plan la visée pure de performances en opposition aux autres. De plus, les points faibles du bot que les expérimentations ont fait apparaître, comme la vulnérabilité aux stratégies de *rush* et le manque de tactique lors de la phase d'attaque, expliquent également le faible taux de victoire de SRbotOne en tournoi.

Ainsi, pour rivaliser en compétition, on peut tâcher de travailler à améliorer et combler ces lacunes pour augmenter les statistiques du bot. Cependant, en plus de corriger les problèmes aperçus, il pourrait être intéressant de pousser encore plus le concept du raisonnement spatial au cœur de la stratégie. En effet, envisager que le système choisisse et conçoive lui-même les stratégies, grâce à du raisonnement spatial, plutôt que d'en fixer une qui ne fait que s'adapter en fonction de l'environnement, comme actuellement. Combiné à un système de reconnaissance du terrain plus puissant, ce système apporterait beaucoup plus de variété et d'imprédictibilité aux agents intelligents, dont le comportement se répète la plupart du temps (car pré-défini). Enfin, l'ensemble du système bénéficierait beaucoup d'une touche d'apprentissage, aussi bien pour comprendre quelles sont les stratégies pertinentes selon le contexte spatial, que selon l'adversaire.

RÉFÉRENCES

Aamodt, A. et Plaza, E. (1994). Case-based Reasoning : Foundational Issues, Methodological Variations, and System Approaches. *AI Communications*, *7*(1), 39–59.

Abramson, B. (1990). Expected-Outcome : A General Model of Static Evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *12*(2), 182–193.

Bauckhage, C. (2014). Game AI, Lecture 6 (2014-05-12). Récupéré de `https://www.youtube.com/watch?v=nfPnoR6sTmc`

Branavan, S. R. K., Silver, D. et Barzilay, R. (2011). Non-linear Monte-Carlo Search in Civilization II. Dans *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence*, 2404–2410. AAAI Press.

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S. et Colton, S. (2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(1), 1–43.

Brügmann, B. (1993). *Monte Carlo Go.* Rapport technique, Citeseer, Syracuse. Récupéré de `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.10.449&rep=rep1&type=pdf`.

Buro, M. (2003). Real-Time Strategy Games : A New AI Research Challenge. Dans *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, 1534–1535. Morgan Kaufmann Publishers Inc.

Buro, M. (2004). Call for AI research in RTS games. Dans *Proceedings of the AAAI Workshop on AI in Games*, 139–141. AAAI Press.

Buro, M. et Churchill, D. (2012). Real-Time Strategy Game Competitions. *AI Magazine*, *33*(3), 106–108.

Certický, M. (2013). Implementing a Wall-In Building Placement in StarCraft with Declarative Programming. *Computing Research Repository, arxiv.org/abs/1306.4460*, 1–17.

Certický, M. et Certický, M. (2013). Case-Based Reasoning for Army Compositions in Real-Time Strategy Games. Dans *Proceedings of the Thirteenth Scientific Conference of Young Researchers*, 70–73.

Cheng-Yu Chen, Xin-Lan Liao, C.-C. L. et Ting, C.-K. (2012). Pattern Formation Based on Potential Field in Real-Time Strategy Games. *2012 Conference on Technologies and Applications of Artificial Intelligence*, 332–337.

Churchill, D. (2016). *Heuristic Search Techniques for Real-Time Strategy Games.* (Thèse de doctorat non publiée). University of Alberta. Récupéré de `https://dl.dropboxusercontent.com/u/23817376/Thesis/thesis.pdf`

Churchill, D. et Buro, M. (2011). Build Order Optimization in StarCraft. Dans *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 14–19. AAAI Press.

Churchill, D. et Buro, M. (2013). Portfolio Greedy Search and Simulation for Large-Scale Combat in StarCraft. Dans *2013 IEEE Conference on Computational Intelligence in Games*, 1–8.

Churchill, D., Saffidine, A. et Buro, M. (2012). Fast Heuristic Search for RTS Game Combat Scenarios. Dans *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 112–117. AAAI Press.

Coulom, R. (2006). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. Dans *International Conference on Computers and Games*, 72–83. Springer.

Danielsiek, H., Stuer, R., Thom, A., Beume, N., Naujoks, B. et Preuss, M. (2008). Intelligent moving of groups in real-time strategy games. Dans *2008 IEEE Symposium On Computational Intelligence and Games*, 71–78. IEEE.

Dereszynski, E., Hostetler, J., Fern, A., Dietterich, T., Hoang, T.-T. et Udarbe, M. (2011). Learning Probabilistic Behavior Models in Real-Time Strategy Games. Dans *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 20–25.

Dicken, L. (2011). Students : Game AI vs Traditional AI. Vu le 2 août 2016. Récupéré de `http://altdevblog.com/2011/07/11/students-game-ai-vs-traditional-ai/`

Forbus, K. D., Mahoney, J. V. et Dill, K. (2002). How Qualitative Spatial Reasoning Can Improve Strategy Game AIs. *IEEE Intelligent Systems*, *17*(4), 25–30.

Fu, D. et Houlette, R. (2004). *The Ultimate Guide to FSMs in Games*, volume 2

de *AI Game Programming Wisdom*, 283–302. Charles River Media : Massachusetts, USA, (first éd.).

Gonzalez, I. et Garrido, L. (2011). Spatial Distribution Through Swarm Behavior on a Military Group in the Starcraft Video Game. *2011 10th Mexican International Conference on Artificial Intelligence*, 77–82.

Graft, K. (2015). When artificial intelligence in video games becomes...artificially intelligent. Vu le 2 août 2016. Récupéré de `http://www.gamasutra.com/view/news/253974/When_artificial_intelligence_in_video_games_becomesartificially_intelligent.php`

Hagelbäck, J. (2009). Using Potential Fields in a Real-time Strategy Game Scenario (Tutorial). Vu le 2 avril 2015.

Hagelback, J. (2012). Potential-field based navigation in StarCraft. *2012 IEEE Conference on Computational Intelligence and Games*, 388–393.

Hagelbäck, J. et Johansson, S. J. (2008). The Rise of Potential Fields in Real Time Strategy Bots. Dans *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, 42–47. AAAI Press.

Heinermann, A. (2009). Broodwar API, an API for interacting with Starcraft : Broodwar (1.16.1). Vu le 2 janvier 2016. Récupéré de `http://bwapi.github.io/`

Hladky, S. et Bulitko, V. (2008). An Evaluation of Models for Predicting Opponent Positions in First-Person Shooter Video Games. Dans *2008 IEEE Symposium On Computational Intelligence and Games*, 39–46. IEEE, IEEE.

Hoang, H., Lee-Urban, S. et Muñoz-Avila, H. (2005). Hierarchical Plan Representations for Encoding Strategic Game AI. Dans *AIIDE*, 63–68. AAAI Press.

Hsieh, J.-L. et Sun, C.-T. (2008). Building a player strategy model by analyzing replays of real-time strategy games. Dans *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, 3106–3111. IEEE.

Kabanza, F., Bellefeuille, P., Bisson, F., Benaskeur, A. R. et Irandoust, H. (2010). Opponent Behaviour Recognition for Real-Time Strategy Games. *AAAI*, 29–36.

Laird, J. E. et Lent, M. v. (2000). Human-Level AI's Killer Application : Interactive Computer Games. Dans *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, 1171–1178. AAAI Press.

Liquipedia (2016). StarCraft Portal : StarCraft, The Game. Vu le 19 janvier 2016. Récupéré de `http://wiki.teamliquid.net/starcraft/StarCraft`

Madeira, C. A., Corruble, V. et Ramalho, G. (2006). Designing a Reinforcement Learning-based Adaptive AI for Large-Scale Strategy Games. Dans *AIIDE*, 121–123.

Miles, C. et Louis, S. J. (2007). Co-evolving Real-Time Strategy Game Playing Influence Map Trees With Genetic Algorithms. Dans *Proceedings of the 2007 IEEE Symposium on Computational Intelligence and Games*, 88–95. IEEE.

Ontañón, S., Mishra, K., Sugandh, N. et Ram, A. (2008). Learning from Demonstration and Case-Based Planning for Real-Time Strategy Games. In *Soft Computing Applications in Industry* 293–310. Springer.

Ontañón, S., Synnaeve, G., Uriarte, A., Richoux, F., Churchill, D. et Preuss, M. (2013). A Survey of Real-Time Strategy Game AI Research and Competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games*, *5*(4), 293–311.

Othman, N., Decraene, J., Cai, W., Hu, N., Low, M. Y. H. et Gouaillard, A. (2012). Simulation-based Optimization of StarCraft Tactical AI through Evolutionary Computation. Dans *2012 IEEE Conference on Computational Intelligence and Games*, 394–401. IEEE.

Pepels, T., Winands, M. H. et Lanctot, M. (2014). Real-time Monte Carlo Tree Search in Ms Pac-Man. *IEEE Transactions on Computational Intelligence and AI in Games*, *6*(3), 245–257.

Perkins, L. (2010). Terrain Analysis in Real-Time Strategy Games : An Integrated Approach to Choke Point Detection and Region Decomposition. Dans *Proceedings of the Sixth Artificial Intelligence and Interactive Digital Entertainment Conference*, 168–173. AAAI Press.

Ponsen, M. (2004). *Improving adaptive game AI with evolutionary learning.* (Thèse de doctorat). Delft University of Technology. Récupéré de `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.109.6055&rep=rep1&type=pdf`

Preuss, M., Beume, N., Danielsiek, H., Hein, T., Naujoks, B., Piatkowski, N., Stuer, R., Thom, A. et Wessing, S. (2010). Towards Intelligent Team Composition and Maneuvering in Real-Time Strategy Games. *IEEE Transactions on Computational Intelligence and AI in Games*, *2*(2), 82–98.

Rabin, S. (2013). *Game AI Pro : Collected Wisdom of Game AI Professionals.* Natick, MA, USA : A. K. Peters, Ltd.

Robertson, G. et Watson, I. (2014). A Review of Real-Time Strategy Game AI. *AI Magazine*, *35*(4), 75–104.

Russell, S. J. et Norvig, P. (2009). *Artificial Intelligence : A Modern Approach (3rd Edition).* Prentice Hall.

Sanselone, M., Sanchez, S., Sanza, C., Panzoli, D. et Duthen, Y. (2014). Constrained control of non-playing characters using Monte Carlo Tree Search. Dans *2014 IEEE Conference on Computational Intelligence and Games*, 1–8.

Shannon, C. E. (1988). *Computer Chess Compendium*, chapitre Programming a Computer for Playing Chess, 2–13. Springer New York

Sharma, M., Holmes, M. P., Santamaría, J. C., Irani, A. et Ram, A. (2007). Transfer Learning in Real-Time Strategy Games Using Hybrid CBR/RL. Dans *Proceedings of the Twentieth international joint conference on Artifical intelligence*, 1041–1046. Morgan Kaufmann Publishers Inc.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M. *et al.* (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489.

Siming Liu, S. J. L. et Nicolescu, M. (2013). Comparing heuristic search methods for finding effective group behaviors in RTS game. *2013 IEEE Congress on Evolutionary Computation*, 1371–1378.

Synnaeve, G. (2012). *Bayesian programming and learning for multi-player video games : application to RTS AI.* (Thèse de doctorat). Institut National Polytechnique de Grenoble. Récupéré de `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.460.6453&rep=rep1&type=pdf`

Synnaeve, G. et Bessière, P. (2011a). A Bayesian model for opening prediction in RTS games with application to StarCraft. Dans *2011 IEEE Conference on Computational Intelligence and Games*, 281–288. IEEE.

Synnaeve, G. et Bessière, P. (2011b). A Bayesian Model for Plan Recognition in RTS Games Applied to StarCraft. Dans *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 79–84.

Synnaeve, G. et Bessière, P. (2011c). A Bayesian model for RTS units control applied to StarCraft. Dans *2011 IEEE Conference on Computational Intelligence and Games*, 190–196.

Szita, I., Chaslot, G. et Spronck, P. (2010). Monte-Carlo Tree Search in Settlers of Catan. In H. van den Herik et P. Spronck (dir.), *Advances in Computer Games*, volume 6048 de *Lecture Notes in Computer Science* 21–32. Springer Berlin Heidelberg

Togelius, J. (2011). On the Game AI versus traditional AI debate. Vu le 2 août 2016. Récupéré de `http://togelius.blogspot.ca/2011/07/on-game-ai-versus-traditional-ai-debate.html`

Uriarte, A. (2016). StarCraft AI. Vu le 16 janvier 2016. Récupéré de `http://www.starcraftai.com/wiki/Main_Page`

Uriarte, A. et Ontañón, S. (2012). Kiting in RTS Games Using Influence Maps. Dans *Proceedings of the Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 31–36. AAAI Press.

Weber, B. G. et Mateas, M. (2009). A data mining approach to strategy prediction. Dans *2009 IEEE Symposium on Computational Intelligence and Games*, 140–147. IEEE.

Weber, B. G., Mateas, M. et Jhala, A. (2011a). A Particle Model for State Estimation in Real-Time Strategy Games. Dans *Seventh Artificial Intelligence and Interactive Digital Entertainment Conference*, 103–108.

Weber, B. G., Mateas, M. et Jhala, A. (2011b). Building Human-Level AI for Real-Time Strategy Games. Dans *Proceedings of the AAAI Fall Symposium on Advances in Cognitive Systems*, 329–336. AAAI Press.

Wintermute, S., Xu, J. et Laird, J. E. (2007). SORTS : A Human-Level Approach to Real-Time Strategy AI. Dans *Proceedings of the Third Artificial Intelligence for Interactive Digital Entertainment Conference*, 55–60. AAAI Press.