Problèmes d'optimisation dans les jeux avec GHOST

Florian Richoux

LINA UMR 6241, Université de Nantes, France
florian.richoux@univ-nantes.fr

Résumé : Cet article présente GHOST, un solveur d'optimisation combinatoire qu'un développeur de jeux de stratégie en temps réel (RTS) peut utiliser comme une boîte noire pour résoudre tout problème modélisé comme un problème de satisfaction/optimisation de contraintes. Nous montrons une manière de modéliser trois différents problèmes de RTS dans ce formalisme, chacun de ces problèmes appartenant à un niveau d'abstraction spécifique, en utilisant le jeu RTS StarCraft comme environnement de test. Sur chacun de ces trois problèmes, GHOST retourne des solutions de très bonne qualité en l'espace de quelques dizaines de millisecondes.
Mots-clés : Game AI, strategie, temps réel, contrainte, solveur, optimisation, combinatoire.

1 Introduction

On peut voir les jeux comme une simplification du monde : le terrain est plus petit, les règles plus simples et moins nombreuses, rendant ainsi les possibilités plus limitées. Cependant, les jeux sont assez riches pour proposer des environnements dynamiques complexes restant difficile pour un ordinateur à appréhender, faire de la prédiction, avoir une compréhension globale de la situation courante et ainsi de prendre une décision. Ceci est en particulier vrai lorsque l'information est incomplète, forçant l'ordinateur à inférer l'état global du jeu à partir de fragments d'informations. C'est le cas avec les jeux de stratégie en temps réel (RTS), où le brouillard de guerre tel que l'a théorisé Clausewitz masque les coups des adversaires. Les RTS constituent ainsi un terreau favorable à la conception de techniques d'intelligence artificielle qui pourraient par la suite s'appliquer avec succès à d'autres applications que les jeux.

De nombreuses techniques d'IA sont utilisées dans les RTS. Sans en faire une liste exhaustive, nous pouvons citer l'apprentissage par renforcement, le *case-based reasoning*, la programmation bayésienne, les *potential fields*, etc. Nous recommandons au lecteur les articles d'enquêtes de Ontañón *et al.* (2013) et Robertson & Watson (2014) pour avoir un balayage complet des techniques d'IA modernes mises en œuvre dans les RTS.

Cependant, il y a peu de travaux en IA pour RTS utilisant la programmation par contraintes à travers la modélisation de problèmes de satisfaction/optimisation de contraintes (CSP/COP). Entre autres, nous retrouvons des algorithmes de *branch and bound* pour l'optimisation de *build order* (Churchill & Buro, 2011), c'est-à-dire la planification de séquences d'actions afin d'atteindre un but fixé le plus rapidement possible. Nous trouvons également des modèles de génération procédurale de cartes esthétiques résolus par un algorithme évolutionniste (Lara-Cabrera *et al.*, 2014). Les CSP/COP ont l'avantage d'offrir un cadre pratique et homogène capable de modéliser un grand nombre de problèmes combinatoires et d'optimisation, et proposent un ensemble varié d'algorithmes pour leur résolution.

Ce formalisme est largement utilisé en IA pour résoudre par exemple des problèmes de recherche de chemin, d'ordonnancement et de logistique. Les algorithmes de résolution de CSP/COP sont habituellement conçus pour être généraux, capable de traiter de n'importe quel problème modélisé dans ce formalisme. Outre sa généralité, il est également intuitif de modéliser un problème avec un CSP/COP. Tout ceci amène des conditions idéales pour la conception

et l'implémentation d'un solveur général extensible et facile d'utilisation.

2 Architecture de GHOST

GHOST est une bibliothèque écrite en C++ [1], sous licence GNU GPL v3, conçue pour traiter de n'importe quel problème de RTS du moment qu'il soit modélisé par un CSP ou un COP. Il s'agit d'une génération et d'une extension du solveur ad-hoc développé pour optimiser la construction de murs défensifs dans le célèbre RTS StarCraft (Richoux *et al.*, 2014).

Avant de rentrer plus dans les détails de GHOST, nous rappelons très brièvement ce que sont formellement un CSP et un COP. Un CSP est un triplet (V, D, C) où V est un ensemble de variables, D l'ensemble des domaines de chaque variable (c'est-à-dire l'ensemble de valeurs que peut prendre chaque variable), et C l'ensemble des contraintes sur les variables de V. Le but est de trouver une valeur dans D pour chaque variable dans V de manière à ce que chaque contrainte de C soit satisfaite. Un COP est la même chose, avec en plus une fonction objectif à maximiser ou minimiser.

GHOST vise deux types d'utilisateurs :

— L'utilisateur de base, qui souhaite simplement utiliser GHOST pour résoudre un problème pré-encodé et disponible dans la bibliothèque. Cet utilisateur a seulement besoin d'instancier ses variables, leur domaine, ses contraintes et éventuellement sa fonction objectif pour décrire l'instance de son problème. Suite à quoi il n'a plus qu'à appeler la fonction `solve` pour lancer la recherche d'une solution. Tout ceci ne requiert que cinq lignes de C++.

— L'utilisateur avancé, qui souhaite résoudre un problème qui n'est pas proposé par la bibliothèque. GHOST a été conçu afin de faciliter l'implémentation de nouveaux problèmes sans devoir modifier une seule ligne de code du solveur, et sans qu'aucune expertise en programmation par contrainte ne soit requise. De plus, notre solveur a été pensé de manière à ce qu'il y ait le moins possible de paramètres, afin d'éviter la tâche fastidieuse et chronophage d'étalonnage des paramètres avant d'obtenir de bons résultats.

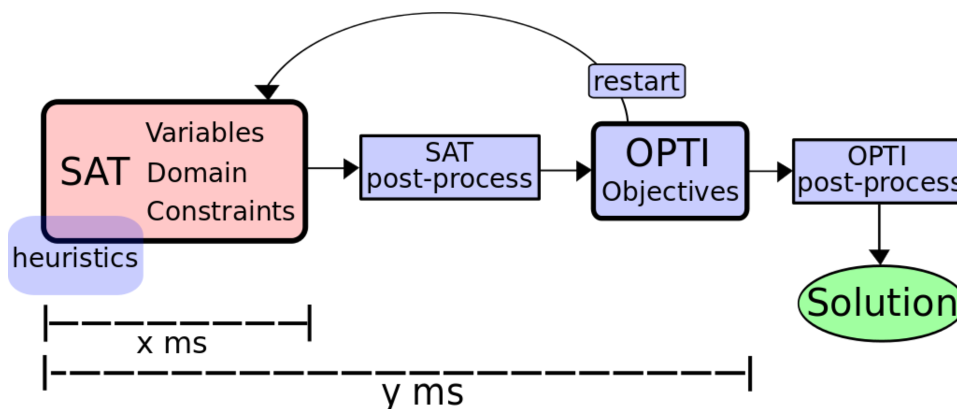

FIGURE 1 – L'architecture du solveur : en rouge, la boucle interne de satisfaction, avec un *timeout* de x millisecondes. En bleu, les mécanismes d'optimisation. Le processus dans son ensemble, en excluant le post-processus d'optimisation, s'exécute en moins de y millisecondes.

Le solveur repose sur une méta-heuristique de recherche locale, implémentant l'algorithme ADAPTIVE SEARCH de Codognet & Diaz (2001). L'utilisateur exécute le solveur via l'appel

1. Code source disponible sur github.com/richoux/GHOST.

à la fonction `solve`. Cette fonction définie dans la classe `Solver` suit les étapes décrites dans la figure 1. Elle est composée de deux principales boucles : la boucle externe d'optimisation, contenant elle-même la boucle interne de satisfaction. Cette dernière, en rouge dans la figure, tente seulement de trouver une solution acceptable, c'est-à-dire vérifiant chacune des contraintes. Il est possible d'appeler `Solver::solve` sans définir de fonction objectif, auquel cas le solveur retourne la première solution acceptable qu'il trouve avant l'expiration de x millisecondes, ou une valeur "erreur" le cas échéant.

Si une fonction objectif est donnée, alors la boucle d'optimisation, en bleu dans la figure, est activée. La fonction objectif peut influencer la boucle de satisfaction si des heuristiques facultatives de choix de variables et/ou de valeurs sont implémentées. La boucle d'optimisation ne fait que comparer la qualité d'une solution trouvée par la boucle de satisfaction par rapport aux solutions précédentes, conserve la meilleure et relance la boucle de satisfaction pendant x millisecondes au plus, le tout avec une échéance de $y > x$ millisecondes. Dans un certain sens, la boucle d'optimisation applique un échantillonnage de Monte Carlo.

La partie optimisation applique également deux post-processus optionnels, le premier à la sortie de la boucle de satisfaction, permettant d'épurer une solution trouvée en écartant toutes variables ne contribuant pas à la satisfaction des contraintes, le second à la sortie de la boucle d'optimisation, permettant d'appliquer des dernières tentatives d'optimisation pour encore améliorer la qualité de la meilleure solution trouvée. Le temps d'exécution de cette dernière est fixé à $(y/100)$ms, ce qui la rend négligeable face au temps total d'exécution.

Concernant les paramètres, il n'en existe que trois : les valeurs x et y du temps de calcul accordé respectivement aux boucles de satisfaction et d'optimisation, ainsi que la longueur de la liste tabou. En effet, l'algorithme ADAPTIVE SEARCH utilise une liste tabou des variables explorées de manière à ne pas revisiter la même variable trop tôt durant la phase de recherche. Le troisième paramètre correspond ainsi au nombre d'itérations que le solveur doit attendre avant de pouvoir revisiter une variable déjà explorée. En réalité, nous avons également implémenté un mécanisme échappatoire afin de ne pas rendre la liste tabou stricte, en autorisant le solveur à tout de même sélectionner une variable tabou s'il n'y a vraiment aucun autre meilleur choix. La liste tabou telle qu'elle est implémentée dans GHOST est plutôt à considérer comme étant une liste de priorité.

Pendant les expérimentations, nous avons réalisé que la même valeur de ce paramètre menait quasi systématiquement aux performances optimales, lorsqu'il est fixé à la valeur $|V|-1$ où $|V|$ est le nombre de variables dans le modèle. Ce paramètre prend alors la valeur $|V|-1$ par défaut et ne nécessite pas a priori d'étalonnage, cependant il reste toujours possible à l'utilisateur de la modifier s'il le souhaite.

L'implémentation de GHOST est composée de cinq classes C++ principales : `Variable`, `Domain`, `Constraint`, `Objective` et `Solver`[2]. Un utilisateur avancé doit implémenter ses propres classes en héritant de `Variable`, `Domain`, `Constraint` et `Objective`. Par exemple, pour créer une classe de variables représentant des unités, ceci peut être fait par `class Unit : public Variable`. Certaines classes sont paramétrées par d'autres classes, comme par exemple `Domain` qui doit connaître la classe des variables. Ainsi, déclarer le domaine d'un problème de sélection de cibles peut être fait par `class TargetDomain : public Domain<Unit>`.

2. Voir le manuel de GHOST (richoux.github.io/GHOST/) pour plus de détails

3 Expérimentations et résultats

Dans cette section, nous détaillons les modèles CSP/COP et les résultats obtenus par GHOST sur trois problèmes à différents niveaux d'abstraction, du moins abstrait au plus abstrait : les contrôles réactifs (également appelés la "micro-gestion"), la tactique et la stratégie. Nous tenons à souligner qu'aucune optimisation du solveur n'a été faite pour traiter ces trois problèmes.

3.1 Contrôles réactifs : la sélection de cible

Le problème de sélection de cible consiste à assigner à chaque unité d'un groupe une cible sur laquelle tirer dans la *frame* courante (c'est-à-dire l'unité de temps du jeu, 1/24ième de seconde pour StarCraft). Autant la satisfaction de ce problème est facile, puisqu'il s'agit simplement d'affecter à une unité en mesure de tirer une cible vivante qui soit à sa portée, autant l'optimisation de ce problème peut être fortement combinatoire : dans beaucoup de RTS comme StarCraft, des pénalités s'appliquent en fonction de la nature de l'unité attaquante et de sa cible. À cela, il faut ajouter des dégâts de zone qu'inflige certaines unités (c'est-à-dire, sa cible et le voisinage de sa cible subiront des dégâts). Si l'objectif est de maximiser le nombre de cibles éliminées par exemple, l'affectation d'une cible à une unité a un impact sur le choix des autres unités.

Nous avons modélisé ce problème ainsi : une variable correspond à une unité du groupe que nous contrôlons. Leur domaine est l'ensemble des unités d'un groupe adverse. L'unique contrainte assure que chacune de nos unités vivantes et prêtes à tirer vise une cible vivante à sa portée, si une telle cible existe. Deux fonctions objectif ont été écrite pour ce problème : *Max damage*, où notre groupe tente d'infliger autant de dommage que possible, et *Max kill*, où notre groupe cherche à détruire le plus de cibles possibles.

Les expérimentations ont été conduites en simulant une bataille entre un groupe de 14 unités provenant de 6 types d'unité de StarCraft en miroir (les unités adverses sont les mêmes que les nôtres et dans une position symétrique). L'adversaire applique une stratégie de sélection aléatoire de cible. Les mouvements n'étant pas pertinent pour le problème de sélection de cible où l'on choisi sa cible pour la *frame* courante, les unités restent statiques dans nos expériences.

Nous avons lancé 100 simulations de bataille avec comme paramètres $x = 2$ms et $y = 5$ms pour les *timeouts*. Nous obtenons les taux de 91% de victoires avec *Max damage* et 82% avec *Max kill*, avec respectivement en moyenne 2,8 et 3 de nos unités survivant à la bataille, sur 14. Pour un temps de calcul extrêmement court, GHOST permet d'avoir une stratégie significativement meilleure qu'une sélection aléatoire.

3.2 Tactique : le wall-in

La tactique n'a pas donné lieu à d'intensives études dans les RTS. On peut noter les travaux de Čertický (2013) sur le problème du *wall-in* et amélioré par Richoux *et al.* (2014).

La technique du mur, ou *wall-in*, est une technique classique de joueurs de RTS afin de protéger l'accès de sa base en créant un goulot d'étranglement à l'entrée de celle-ci afin d'en faciliter sa défense en cas d'invasion. Ce mur se bâtit en jouxtant bout à bout ses propres bâtiments.

Le problème de satisfaction consiste seulement à placer des bâtiments (nos variables) à des positions dans l'espace (leur domaine) afin de créer un mur, à savoir une ligne continue de bâtiments, d'un point A à un point B (nos contraintes). Encore une fois, l'optimisation peut être de multiples natures. Nous en avons considéré trois : 1) composer le mur avec le moins

de bâtiments possibles, 2) avec des bâtiments les moins avancés technologiquement (et donc plus rapides à obtenir), et 3) avec le moins de trous possibles. En effet dans certains RTS comme StarCraft, deux bâtiments placés côte à côte peuvent en réalité être séparés par un trou de quelques pixels, selon la nature et la position de ces bâtiments. Certaines combinaisons peuvent entraîner des trous assez larges pour laisser passer de petites unités, typiquement la plus petite unité du jeu, le *zergling*, qui est de 16×16 pixels. Nous comptons donc comme étant un trou un écartement de 16 pixels ou plus entre deux bâtiments accolés.

Nous avons extrait 48 entrées de base venant de 7 cartes de StarCraft, et lancé quatre expérimentations de 4800 exécutions de GHOST chacune (100 fois chaque entrée) : la première sans optimisation, ne cherchant qu'à résoudre le problème de satisfaction laissant 160ms au solveur. Puis, nous avons lancé une expérimentation par fonction objectif, avec comme paramètres $x = 20$ms et $y = 150$ms. L'optimisation par GHOST montre des gains significatifs, en particulier pour limiter le nombre de trous : pour 4,05 bâtiments en moyenne dans un mur sans optimisation, on se retrouve avec 2,56 bâtiments avec optimisation, avec 98,04% de solutions trouvées en moins de 150ms. Un indice de 1,99 en technologie est optimisé à 1,35 avec 97,54% de taux de résolution. Enfin, là où l'on avait 1,32 trou de 16 pixels ou plus en moyenne dans un mur, on se retrouve avec seulement 0,03 trou avec la fonction objectif associée, avec un taux de résolution de 97,5%. Pour 4800 entrées de base, 4680 murs ont été trouvé dont 4527 parfaits, c'est-à-dire sans aucun trou de plus de 16 pixels.

3.3 Stratégie : le build order

Le problème de *build order* est l'un des rares problèmes déjà traité via la programmation par contrainte dans Churchill & Buro (2011). Il s'agit d'un problème d'ordonnancement d'actions, afin d'atteindre un but fixé en un minimum de temps. Ces actions peuvent être la production d'unités, la construction de bâtiments, ainsi que la recherche de technologies et d'améliorations.

Ce problème peut être vu comme un problème de permutation où les variables sont toutes les actions nécessaires pour atteindre le but donné, le domaine est l'ordre de ces actions et l'unique contrainte est de vérifier la dépendance des actions : souvent, une action A ne peut être effectuée qu'après des actions B et C.

Nous avons extrait le *build order* de 3647 parties jouées par des joueurs de haut niveau et avons comparé en combien de temps GHOST arrive à atteindre le même état de jeu obtenu au bout de 7 minutes de jeu en mode rapide (soit 10.000 *frames*). Ces 7 minutes de jeu en mode rapide correspondent à environ 11 minutes en mode normal, soit 660 secondes.

En moyenne sur 3647 *build orders*, GHOST ordonne les actions afin d'arriver au même état de jeu qu'un joueur humain de haut niveau atteint au bout de 656,02 secondes en mode normal (ne se passant rien d'intéressant en moyenne durant les 3,98 dernières secondes) en seulement 619,62 secondes, le tout avec des *timeouts* de $x = 20$ms et $y = 30$ms. Ceci donne un ratio temps CPU / temps du *build order* de 0,007%. À noter que le taux de résolution en moins de 30ms est en moyenne de 94,4%.

En comparaison, Churchill & Buro (2011) arrivent 90% des fois a des *build order* de même durée que des joueurs de haut niveau, donc de qualité moindre que GHOST, après 3,735 secondes de calcul, donnant un ratio temps CPU / temps du *build order* de 1,5%.

En plus des 3647 parties jouées par des joueurs de haut niveau, nous avons extrait 8 *build orders* venant des meilleurs joueurs professionnels au niveau mondial, tous joueurs sud-coréens. Contrairement aux 3647 *build orders* précédents, ces partie-là ont été choisies "manuellement", expliquant leur faible nombre. Il en résulte que GHOST arrive en l'espace de 30ms a de meilleurs *build order* que les meilleurs joueurs au monde, avec en moyenne des temps de

construction de 597,25 secondes pour arriver au même état de jeu que les professionnels au bout de 643,38 secondes en moyenne. Ce temps est bien plus court que 660 secondes du fait que nous avons arrêté d'enregistrer le *build order* dès que les premières attaques commencent, situation qui arrive plus vite chez les joueurs de très haut niveau. GHOST a trouvé une solution en 30ms dans 96,3% des cas.

4 Conclusion et discussion

Nous avons introduit dans cet article GHOST, une bibliothèque d'optimisation combinatoire visant à résoudre les problèmes liés aux RTS modélisés par un problème de satisfaction/optimisation de contraintes. Les résultats obtenus par GHOST sur trois problèmes classiques dans les RTS à différents niveaux d'abstraction sont très encourageant pour poursuivre le développement de cette bibliothèque et y intégrer de nouveaux problèmes. En effet, le solveur intégré à GHOST trouve des solutions de très bonne qualité en l'espace de quelques dizaines de millisecondes seulement, sans qu'aucune optimisation du solveur ni étalonnage de paramètres (en dehors des *timeouts*) ne soit nécessaire. Les résultats ainsi obtenus sont bien souvent meilleurs que ceux que l'on trouve dans la littérature.

Le lecteur attentif aura réalisé que GHOST est en réalité une bibliothèque qui peut être utilisé pour n'importe quel problème CSP/COP, pas seulement ceux liés aux RTS ni même aux jeux. L'un de nos points de départ était de faire de GHOST une bibliothèque C++ très facile à intégrer dans les IA (dits *bots*) pour le jeu StarCraft. GHOST devrait être bientôt couplé avec la bibliothèque BWAPI 4, nous permettant d'avoir (presque) toutes les informations dont nous avons besoin sur les éléments du jeu, les évènements, etc. De plus, une version de GHOST en C# est en cours, ce qui permettrait d'en faire un *plugin* pour le célèbre moteur de jeu *Unity*. À noter aussi que GHOST est en cours de transfert technologique, via un procédé de double licence (une GNU GPL et une propriétaire), auprès de la société de développement de jeu *Insane Unity* [3] pour le développement de leur prochain jeu RTS. En d'autres mots, ce n'est pas le cœur de GHOST mais ses futures dépendances et intégrations qui en feront une bibliothèque d'optimisation combinatoire pour les RTS, et de manière plus générale, pour les jeux vidéos.

Références

CHURCHILL D. & BURO M. (2011). Build order optimization in starcraft. In V. BULITKO & M. RIEDL, Eds., *AIIDE*, p. 14–19 : The AAAI Press.

CODOGNET P. & DIAZ D. (2001). Yet another local search method for constraint solving. In *proceedings of SAGA'01*, p. 73–90 : Springer Verlag.

LARA-CABRERA R., COTTA C. & FERNÁNDEZ-LEIVA A. J. (2014). A self-adaptive evolutionary approach to the evolution of aesthetic maps for a RTS game. In *IEEE World Congress on Computational Intelligence (WCCI)*.

ONTAÑÓN S., SYNNAEVE G., URIARTE A., RICHOUX F., CHURCHILL D. & 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), 1–19.

RICHOUX F., URIARTE A. & ONTAÑÓN S. (2014). Walling in strategy games via constraint optimization. In *AIIDE* : The AAAI Press.

ROBERTSON G. & WATSON I. (2014). A review of real-time strategy game AI. *AI Magazine*.

ČERTICKÝ M. (2013). Implementing a wall-in building placement in starcraft with declarative programming. *arXiv*.

3. www.insaneunity.com