Résolution de Contraintes

Projet Awélé

I.F.I.P.S. INFO 2005

Thibault Blaiset
Olivier Hécart
Sylvain Machefert
Aurélien Méré

Projet Résolution de Contraintes

Awélé – IFIPS INFO 2005

Table des matières

Introduction

Ce rapport présente le travail effectué dans le cadre du projet de résolution de contraintes d'IFIPS INFO cinquième année. Nous présenterons ici les différents choix effectués au cours du projet, et les raisons de ces choix.

Le sujet était la réalisation d'un joueur d'Awélé capable d'affronter un autre programme bâti sur le même principe. Pour faciliter l'exécution et la confrontation avec un adversaire, le programme devait communiquer en mode console. Afin de répondre au problème et d'avoir un code exécutable le plus rapide possible, nous avons choisi d'utiliser le langage de programmation C.

Structure du programme

Structure de données

La structure de données utilisée est la suivante :

```
typedef struct awele_s {
      char trou[12];
      char capture[2];
      char total[2];
      char ended;
      char best;
} awele;
```

Le tableau *trou* correspond comme son nom l'indique au plateau de jeu utilisé. Les 12 cases du tableau contiennent le nombre de graines présentes à l'instant t dans les différentes cases du plateau. Nous avons choisi le type *char* pour ces valeurs et pour beaucoup d'autres par la suite dans un souci de limitation de la taille mémoire utilisée. En effet, quelque soit l'architecture sur laquelle sera utilisé le programme, l'utilisation d'un char nous assure que celui-ci occupera un octet en mémoire.

Les deux *char capture* correspondent au nombre de graines capturées depuis le début de la partie par chacun des deux joueurs.

Les deux *char total* correspondent chacun au nombre de graines restantes de chacun des deux cotés.

Le *char ended* permet de savoir si la partie est finie.

Le *char best* définit le meilleur coup à jouer selon notre heuristique après calcul par alphabéta.

Programme principal

Le programme principal, après avoir chargé la bibliothèque d'entrée lance la fonction principale, *jeu*.

La partie centrale du programme est la fonction *jeu* définie au sein du fichier awele.c. Cette fonction est composée d'une phase d'initialisation (qui évalue la puissance de la machine et qui effectue les échanges de messages prévus par l'arbitre) ainsi que d'une boucle exécutée tant que la partie n'est pas terminée. Au cours de cette boucle, soit on effectue les calculs nous permettant de jouer, soit on attend la réponse de l'autre joueur.

Si c'est à nous de jouer, on lance l'algorithme qui détermine le coup à jouer (la fonction *pense*) en lui passant en paramètre la configuration actuelle du jeu et le temps restant de la partie. Cette fonction s'occupe de déterminer si le coup est dans le champ d'action des librairies précalculées et, dans le cas contraire, paramètre la profondeur de l'alpha-beta en fonction du temps restant et du

nombre de graines restantes. Dans la boucle principale, le temps pris par cette fonction est calculé à l'aide de timers afin de connaître et conserver le temps total de la partie (pour nous). Le résultat de la fonction pense est enfin passé à l'arbitre via la sortie standard.

Si c'est à l'adversaire de jouer, on se contente d'attendre en guettant l'entrée standard.

En fin de boucle et dans les deux cas, on modifie notre représentation mémoire de la configuration de la partie en fonction du dernier coup joué (par nous ou par l'adversaire).

Heuristique

Littérature

Afin de résoudre le mieux possible le problème de l'awélé, nous avons essayé de trouver une fonction heuristique la plus efficace possible. Cette fonction nous permet de faire des estimations sur le choix de coup à effectuer à un instant donné.

Pour la choisir au mieux nous nous sommes donc dans un premier temps intéressé aux règles de l'awélé et aux différentes tactiques de jeux. Pour cela, nous avons consulté plusieurs sites internet et le livre *Awélé* de Pascal Reysset et François Pingaud (Ed. Chiron-Algo).

Une des informations importantes que nous avons tirée de ce livre est l'exemple donné concernant la finale des olympiades du jeu sur micro-ordinateur de 1990. En effet, lors de cette finale, deux programmes se sont affrontés, l'un disposant d'une une heuristique relativement complexe, basée sur neuf critères, avec plusieurs phases de jeu différentes, l'autre sur une heuristique brute, qui prend seulement en compte le nombre de graines gagnées. Cette finale a donné gagnant le second programme, utilisant une heuristique simple, qui en contrepartie lui permettait de descendre loin dans l'arbre.

Cette première approche, confirmée par la suite par les tests que nous avons effectués, nous a amené à choisir une heuristique simple, et à favoriser la descente dans l'arbre.

Première idée

La première heuristique utilisée, très simple, est la suivante : On calcule la différence, pour un coup donné, entre le nombre de graines restantes de notre coté et le nombre de graines restantes chez l'adversaire.

Notre objectif a dans un premier temps été d'utiliser cette heuristique avec un alpha-béta et d'optimiser le code afin de descendre le plus loin possible. Nous présenterons les différentes optimisations apportées dans la partie concernant l'alpha-béta.

Deuxième idée

Afin d'effectuer des tests de performance, nous avons défini une deuxième heuristique plus poussée correspondant à la somme pondérée des valeurs suivantes :

- La première heuristique: delta des graines des deux cotés.
- Le potentiel: Nombre de graines de notre coté.
- La mobilité de jeu: Nombre de coups jouables.
- Le plus grand barrage: Plus grand nombre de cases vides contiguës.

Cette heuristique, bien que plus compliquée que la précédente reste en un ordre de calcul

correspondant à un parcours du plateau de jeu seulement. D'autres paramètres peuvent être envisagés pour le calcul, en particulier au niveau de la gestion des greniers, de l'aspect défense par rapport aux possibilités de réponse laissées à l'adversaire. Mais ces paramètres plus complexes entraînent des calculs « en avant » qui dégraderaient immédiatement les performances de notre alpha-béta de manière significative.

Conclusion

Pour tester les différences, nous avons fait s'affronter deux programmes implémentant ces deux heuristiques, sans toutefois utiliser de bibliothèque d'entrée (à cause du temps nécessaire pour la génération des bibliothèques) ; les programmes étant par ailleurs en tous points semblables.
Le résultat de ces tests a fait ressortir les deux points suivants :

- L'utilisation de l'heuristique « complexe » n'a pas révélé d'influence sensible sur la durée des calculs du coup à jouer. Nous avons effectivement utilisé des optimisations visant à dérouler toutes les boucles finies sur le calcul du jeu, ce qui provoque finalement une faible différence entre le temps de calcul sur les deux versions.
- Sur l'ensemble des tests, l'heuristique la plus simple s'est avérée gagnante.

Finalement, nous avons donc choisi d'utiliser pour la version finale l'heuristique la plus simple.

Les algorithmes

Pense

Cette fonction calcule pour un état de jeu donné, ce qu'elle considère comme le meilleur coup à jouer.

Dans un premier temps, on regarde si on peut utiliser la bibliothèque d'entrée pour jouer. Si c'est le cas, on joue le coup défini au sein de la bibliothèque pour l'état de jeu donné. Sinon, on va lancer la fonction alpha-béta. La profondeur de recherche dans l'alpha-béta est déterminée en fonction du temps restant et du temps écoulé lors du coup précédent. Nous étudierons ceci plus en détail dans la partie concernant la gestion du temps.

Afin de réduire le temps de calcul, plusieurs optimisations ont été effectuées. L'une d'elles, appelée « coup qui tue », consiste, à la fin d'un alpha-beta, à conserver en mémoire le coup de l'adversaire que l'on considère le plus bénéfique pour lui selon notre heuristique. On conserve aussi le meilleur coup à jouer pour nous, après le coup supposé de l'adversaire. Si au coup suivant le coup joué par l'adversaire s'avère être le coup prévu, le « coup qui tue » est appliqué et on joue directement la réponse que l'on avait conservée sans lancer aucun calcul. Ce « coup qui tue » est moins efficace car il équivaut à une recherche de profondeur moindre par l'alpha-beta et après de multiples tests, nous avons finalement décidé de ne pas l'intégrer à la version finale, car cela avait un effet trop défavorisant.

Une autre optimisation, particulièrement simple mais à laquelle on ne pense pas forcément, est de s'assurer préalablement que l'on peut jouer plusieurs coups différents (à savoir que l'on a plusieurs cases non vides sur notre plateau). Dans le cas inverse, si l'on a qu'une seule possibilité, on effectue celle-ci ce qui nous évite une recherche en profondeur alpha-béta inutile.

Alpha-Béta

Cet algorithme implémente le principe d'alpha-béta en ne gardant en mémoire aucune trace de l'arbre. Il ne fait que le parcourir en maintenant à jour la structure de données décrite plus haut. La

profondeur de recherche est passée en paramètre. Cet algorithme effectue des coupes dès que le nombre de graines gagnées dépasse un certain seuil, passé en paramètre et supérieur à 24. En effet, avec plus de 24 graines gagnées, la partie est terminée et on n'a plus besoin de continuer à réfléchir.

Cet algorithme a lui aussi subi plusieurs optimisations dont la principale concerne la structure de données. En effet, cette structure a été prévue de façon à pouvoir copier exclusivement les parties indispensables de la structure et avec des outils optimisés de copie.

Rappel de la structure :

```
typedef struct awele_s {
      char trou[12];
      char capture[2];
      char total[2];
      char ended;
} awele;
```

Lors de la recherche, la copie de structure se fait à l'aide de la fonction memcpy sur 16 octets. Le caractère ended n'est pas indispensable et la copie est plus rapide sur la plupart des architectures.

Calcul d'un nouveau jeu

Cette fonction est utilisée pour calculer, en fonction d'un coup donné et d'un état de jeu, l'état de jeu résultant. Cette fonction, qui peut sembler anecdotique au départ est en fait assez importante en ce qui concerne les optimisations apportées au programme. En effet, bien que relativement simple, cette fonction est appelée lors de la plupart des alpha-béta et toute instruction économisée permet de gagner un temps non négligeable.

Gestion du temps

La gestion du temps est un point important de l'application, elle doit en effet nous permettre d'adapter la qualité de notre réflexion au temps disponible dans la partie. Elle doit aussi permettre de faire tourner le programme sur des machines aux performances non connues par avance.

Initialisation

La première phase, pour laquelle nous disposons, conformément aux consignes, d'une minute nous permet de définir la profondeur de recherche à laquelle nous allons commencer.

Pour déterminer ceci, nous cherchons la profondeur à partir de laquelle le temps de calcul sera supérieur à une seconde (sur le plateau plein).

Par la suite :

- On ajoute 2 à cette valeur pour obtenir la profondeur dite « dégradée » qui sera utilisée si le temps disponible descend en dessous d'une minute.
- On ajoute 4 à cette valeur et on utilise cette somme comme profondeur de recherche initiale. On part ainsi avec une profondeur initiale qui dépend des performances de la machine.

Le fait de passer les niveaux un par un nous permet de s'assurer que l'on ne dépassera pas la minute d'initialisation autorisée. En effet, le coup qui nous fera quitter la phase d'initialisation (appelons le n) est le premier à prendre plus d'une seconde. La profondeur précédente (n-1) ayant été calculée en moins d'une seconde, et étant donnée que d'un niveau à l'autre, le temps de calcul est multiplié par un facteur 6 (pour simplifier), on ne mettra pas plus de 6 secondes pour calculer le niveau n. Au

final, on considère donc que pour tous les niveaux précédents, on dispose de 54 secondes. Chacun des coups passés ayant duré moins d'une seconde, il y a donc au moins 55 coups.

Sur notre machine de test, un PIII 700, la fonction d'initialisation stoppe généralement entre les niveaux 13 et 15. Chaque passage à un niveau supérieur nécessitant 6 fois plus de calculs, passer les 40 niveaux séparant notre résultat du seuil critique, il faudrait une machine approximativement et « au plus » 6^{40} fois plus puissantes ...

Adaptation

Par la suite, au cours de la partie, la profondeur de recherche va évoluer constamment. Ainsi, on analyse à chaque coup le temps du coup joué précédemment afin de décider de la profondeur du coup que l'on va jouer.

$$Nouvelle\ profondeur = Ancienne\ profondeur + PartieEntière\left(1.5 - \frac{Temps_{précédent}}{Temps_{restant}} * 15\right)$$

Cette fonction présente plusieurs avantages dans la gestion du temps : elle nous permet d'augmenter la profondeur de recherche lorsque l'on se rend compte que le temps de calcul diminue (en raison d'une densité de graines qui évolue par exemple), et nous permet aussi de diminuer cette même profondeur de recherche si par exemple un autre processus lancé sur la machine de test monopolise les ressources. Dans ce dernier cas, une observation d'un ralentissement à un coup entraînera une limitation de la profondeur de recherche au coup suivant.

Mode dégradé

Lorsque le programme atteint sa dernière minute de jeu, il rentre en mode dégradé : tous les calculs se font à la profondeur trouvée par l'initialisation à laquelle on a ajouté 2. Le plateau de jeu étant sensé être beaucoup plus simple que celui testé à l'initialisation, on garantit ainsi une réponse en moins d'une seconde, ce qui nous offre une espérance de soixante coups en mode dégradé, au minimum.

Les bibliothèques

Comme il avait été autorisé dans le sujet, nous avons utilisé pour le départ de notre programme une bibliothèque de coups prédéfinie. Nous avons calculé deux bibliothèques différentes mais au final, seule celle présentant les coups d'entrée sera utilisée pour les raisons que nous expliquerons par la suite.

Bibliothèques d'entrée

Nous disposons de deux bibliothèques d'entrée qui correspondent chacune à une position de jeu (on joue en premier ou en deuxième).

Pour générer ces bibliothèques, nous utilisons le même programme que celui qui nous sert pour le jeu, en modifiant sa compilation. Pour différencier les versions, nous avons plusieurs directives de compilation :

```
#define ENTRYLIB_BUILD            1
#define ENTRYLIB_MAXPROF          8
#define ENTRYLIB_ALPHABETAPROF    18
#define ENTRYLIB_NOMOREALPHABETA  24
```

Le premier prend la valeur 0 ou 1 selon que l'on souhaite jouer ou générer la bibliothèque. Les deux suivants servent lors de la génération d'une bibliothèque et définissent :

- ENTRYLIB_MAXPROF : Le nombre de coups calculés dans la bibliothèque pour chaque joueur (8 coups correspondra par exemple à une visibilité à 16 coups en avant). La valeur de cette variable va avoir une influence sur la taille de la bibliothèque générée ainsi que sur le temps nécessaire à la génération. Plus cette valeur sera élevée, plus nous pourrons l'utiliser pendant un temps prolongé. Cela aura pour influence de nous laisser plus de temps pour les coups que nous aurons à calculer « en temps réel ».
- ENTRYLIB_ALPHABETAPROF : Cette valeur correspond à la profondeur utilisée pour l'alpha-béta destiné à évaluer un coup. Sa valeur aura une influence sur le temps de calcul et sur la qualité des coups choisis.
- ENTRYLIB_NOMOREALPHABETA : Cette valeur définit le seuil maximal des alpha-béta dont nous avons parlé dans la partie correspondante.

Ces trois variables ont comme nous l'avons dit une influence sur la qualité de la bibliothèque et par extension sur les performances de notre programme. Le choix des valeurs des constantes a été dicté par :

- Un compromis entre le temps de calcul et la qualité de la bibliothèque (pour chaque coup).
- Un compromis entre l'espace utilisé par la bibliothèque, le temps de calcul et son efficacité (en nombre de coups précalculés). La taille de la bibliothèque a une importance sur le disque dur (taille et portabilité du programme) et sur la mémoire vive (La bibliothèque est intégralement chargée en mémoire au début du programme).

Bibliothèques de fin

L'idée de la bibliothèque de fin était de nous permettre de connaître, pour une configuration de jeu donnée, les coups à jouer pour gagner.

Pourtant, lors des fins de parties il reste peu de graines et les recherches alpha-beta sont très rapides. De plus, l'un des problèmes de cette bibliothèque est que l'on doit vérifier, à chaque coup, que l'on est dans une configuration gérée.

Même si une bibliothèque de fin peut probablement économiser du temps de calcul, les gains potentiels nous ont parus insuffisants pour justifier sa création.

Description des fichiers source

Les fichiers source sont au nombre de 6 :

- **algos.c** qui définit 3 fonctions :
  - Une fonction **pense** qui cherche le prochain coup à jouer pour la configuration de jeu actuelle.
    - Si on est dans le domaine d'action de la bibliothèque d'entrée on y recherche le prochain coup.
    - Sinon on effectue un alpha-beta dont la profondeur et les conditions d'arrêt sont paramétrées par le module de gestion du temps.
  - Une fonction **joue** qui implémente l'algorithme permettant de gérer l'influence d'un coup sur le jeu.
    - On vide la case de départ.
    - On distribue les graines.
    - On prend les graines potentiellement gagnées.

    - On teste si la partie est finie.
  - Une fonction **alphabeta** qui implémente l'algorithme de même nom. Des optimisations ont été réalisées afin d'éviter des appels de calculs inutiles. Ainsi, si l'on a déjà gagné la partie (au moins 24 graines gagnées) l'algorithme stoppe ou continue la recherche en fonction d'un paramètre. Ceci permet d'éventuellement maximiser le score final si le temps restant le permet.
- **awele.c** qui définit le programme principal, et la fonction jeu dont nous avons parlé précédemment
- awele.h définit les directives de compilation du programme ainsi que la déclaration des différentes fonctions et structures de données.
- divers.c fournit des fonctions pour l'initialisation des awélés ainsi que pour l'affichage d'un jeu.
- entrylib.c fournit les fonctions pour l'interfaçage avec la bibliothèque d'entrée : chargement en mémoire ou génération.
- heuristique.c contient la fonction d'évaluation d'un jeu donné.

Compilation et mise en route

Une fois décompressé, le programme est compilé par la commande « make ». Il est à noter que celui-ci est optimisé pour une architecture de type Intel Pentium Pro ou ultérieur. Pour modifier ceci, il est possible de changer la variable « march » de compilation dans le Makefile.

Le programme est ensuite lancé par la commande « awele » avec comme paramètre le numéro du joueur (0 : on commence, 1 : l'adversaire commence).

Répartition du travail

Le tableau ci-dessous représente les différentes parties de la conception et du développement du projet. Nous n'avons pas présenté le travail de chacun sur des pages séparées mais avons regroupé ici la répartition dans ce tableau.

La répartition n'ayant en réalité pas été totale, une croix dans une case indique que la personne a effectué une partie significative du travail, mais pas forcément qu'elle a été la seule à travailler sur le point.

	Fichier(s) concerné(s)	***Thibault Blaiset***	***Olivier Hécart***	***Sylvain Machefert***	***Aurélien Méré***
Documentation préliminaire		**X**			**X**
Structure et E/S	`Awele.h, Awele.c`			**X**	**X**
Heuristique	`Heuristique.c`	**X**	**X**		
Algorithmes	`Algos.c`			**X**	**X**
Gestion du temps	`Algos.c, Awele.c, Init.c`	**X**	**X**		
Bibliothèques	`Entrylib.c`				**X**
Tests	`Tous`	**X**	**X**	**X**	**X**
Rapport		**X**	**X**	**X**	**X**

Conclusion

Nous sommes particulièrement satisfaits du comportement du programme aussi bien en terme de gestion du temps qu'en terme de performances globales. Par ailleurs, lors des premiers tests menés entre nous contre le programme de certains de nos camarades, les résultats nous ont semblé satisfaisants et nous attendons avec impatience le déroulement du tournoi !