Pierre-André Roy – 08 359 625

Jean-François Bourget – 08 357 636

Projet de session – Connect5

IFT630 - Processus concurrents et parallélisme

Travail présenté

À

M. Gabriel Girard

26 avril 2012

Université de Sherbrooke

Table des matières

Introduction au projet

Le présent document a pour but de décrire notre projet dans le cadre du cours « IFT630 – Processus concurrents et parallélisme » et de détailler son développement.

Nous avons décidé de reproduire le jeu Connect5. Il s'agit d'un jeu à 2 joueurs qui se joue tour par tour. Le but est de placer 5 billes de la même couleur collées les unes aux autres, soit en forme de ligne verticale, horizontale, ou diagonale. Le premier joueur à atteindre cet objectif gagne la partie.

Nous avons créé un joueur artificiel afin de jouer contre la personne utilisant notre application. C'est à cet endroit que la majorité du travail dans le projet a été fait. En effet, nous avons créé un joueur qui prend des décisions par lui-même selon l'avancement de la partie, soit les coup joués jusqu'à maintenant, et selon son rôle à jouer dans la partie.

Une certaine base de ce projet avait déjà été réalisée par Pierre-André Roy lors de son cours d'intelligence Artificielle « ift615 » avec Monsieur Éric Beaudry. Cependant, l'algorithme de recherche, qui sera présenté ci-dessous, était implémenté de façon séquentielle, et le tout prenait un temps exponentiel plus on augmentait la taille de la grille ou la profondeur de recherche dans l'arbre d'états. Nous avons donc décidé de reprendre en main le projet, de le paralléliser à l'aide de différentes méthodes l'algorithme de recherche, et d'apporter certaines modifications à l'interface pour faciliter la manipulation, l'affichage, ainsi que le choix des différents modes et « tests ».

Le projet, en sa totalité, est fait avec le langage Java, avec Eclipse comme environnement de développement. Les outils utilisés pour la programmation multi-fils seront expliqués plus loin dans le rapport.

Algorithme de recherche

Afin de rendre notre joueur intelligent, nous avons utilisé un algorithme d'intelligence artificielle pour la prise de décision. Nous avons utilisé l'algorithme « Minimax avec élagage Alpha-beta ».

Dans une partie de connect5 utilisant cet algorithme, il y a deux joueurs avec chacun leur rôle à jouer dans la partie. Le joueur 1, max, cherche à maximiser le pointage en tout temps. Peut importe l'état du jeu (un état étant une grille dans laquelle chaque case est une variable et a, ou non, une valeur (jeton noir ou blanc) associée), le joueur 1 cherche à terminer la partie le plus rapidement possible, alors il tentera de placer ses jetons de façon à former une ligne par le moins de coups possibles. Le joueur 2, min, cherche à empêcher l'autre de former des lignes en bloquant ses tentatives dès qu'il le peut. En gros, chaque état à une valeur plus ou moins grande en fonction du joueur à qui c'est le tour et des « combinaisons de jetons » sur la grille. La recherche d'un coup optimal peut être vue comme la recherche d'un état ayant la plus grande valeur dans un arbre d'état.

Plus la profondeur de recherche est grande, plus le résultat sera précis puisqu'il sera possible de prévoir un plus grand nombre de coups à l'avance, mais plus la recherche sera longue.

La partie « élagage alpha-beta » de l'algorithme, quant à elle, accélère le processus de recherche en coupant des branches de l'arbre dans lesquelles on est assuré de ne pas trouver de meilleurs résultats que le meilleur actuel.

Nous n'entrerons pas plus en détails dans l'explication de cet algorithme puisque cela n'est pas le but du cours. Il est cependant essentiel de comprendre le rôle de chacun des 2 joueurs, ainsi que le rôle de la variable « profondeur » pour pouvoir effectuer et comprendre des tests de performance précis sur notre jeu connect5.

Si notre algorithme devait être utilisé dans une partie réelle, il serait essentiel d'implémenter le principe de « iterative deepening ». Bref, si notre joueur artificiel avait 10 secondes pour jouer son tour, voici ce qu'il ferait pendant ces 10 secondes :

1. PROFONDEUR = 1.
2. Recherche dans l'arbre.
3. Retour du meilleur résultat.
4. S'il reste du temps, recommencer la recherche avec PROFONDEUR+1.

Cela permettrait de trouver une réponse correcte rapidement, tout en laissant la possibilité de chercher une réponse plus juste si le temps le permet.

Cependant, comme notre but était simplement de faire des comparaisons de temps entre l'algorithme séquentiel et parallèle, cette technique nous importait peu.

Parallélisation

Comme nous l'avons mentionné plus haut, la recherche dans l'arbre prenait un temps irréaliste moindrement que les configurations du jeu devenaient lourdes. Voici un exemple concret de la grandeur du domaine à explorer par l'algorithme.

En début de partie avec 0 coup joué, avec une grille de 10x10 comprenant donc 100 cases,

Une recherche avec une profondeur de 1 :

→ doit simuler un coup dans chacune des 100 cases, et retourner le coup ayant la meilleure valeur. Au total, 100 états explorés.

Une recherche avec une profondeur de 2 :

→ doit simuler un 1er coup dans chacune des 100 cases, un 2^{e} coup dans chacune des 99 restantes. Au total, 100*99= 9900 explorés.

Une recherche avec une profondeur de 4 :

→ doit simuler un 1er coup dans chacune des 100 cases, un 2^{e} coup dans chacune des 99 cases restantes, un 3^{e} dans les 98 restantes, et un 4^{e} dans les 97 restantes.

Au total, 100*99*98*97 = 94109400 états explorés.

Il est facile de voir que le temps pris par la recherche explose rapidement.

Il était donc très intéressant pour nous de tenter de paralléliser cet algorithme, autant dans un but d'amélioration de performance que par curiosité personnelle. À quel point sera-t-il possible d'accélérer la recherche?

Au niveau de la communication « entre fils » et « entre fil et créateur » , la technique utilisée est la plus simple qui soit. Il n'était pas nécessaire pour nos fils de communiquer entre eux puisque l'état à la plus grande valeur (le meilleur coup à jouer) peut être trouvé à tout moment lors de la recherche. Il n'y a aucune circonstance dans laquelle un fil peut trouver un état qui pourrait impliquer l'arrêt de la recherche faite par les autres fils.

Il ne restait donc qu'à assurer un système de communication fiable entre les fils et le créateur des fils. Un outil offert par java, soit le « CompletionService » nous a permis de résoudre ce problème de façon élégante. Le tout sera décrit plus en détails un peu plus loin.

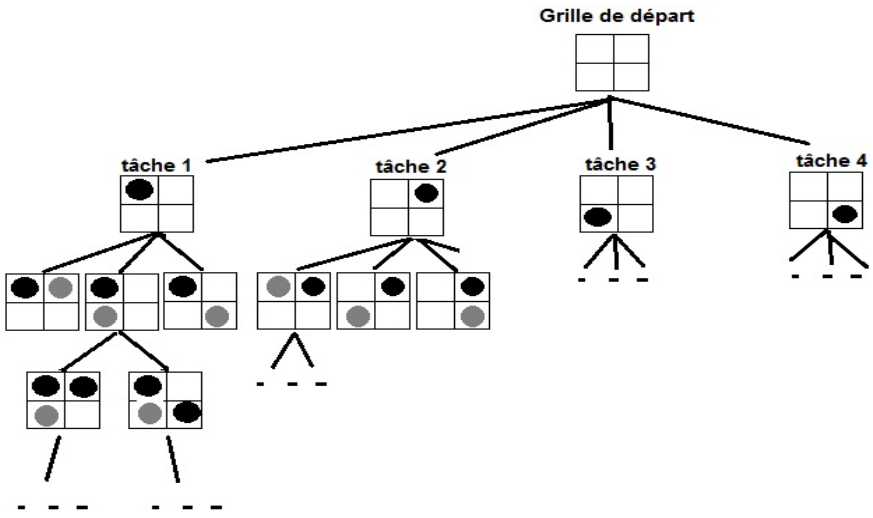
Division de l'algorithme de recherche pour le « multithreading »

Il fallait trouver une façon efficace de diviser le travail entre les fils créés. Nous avons passé beaucoup de temps à trouver le bon point d'entrée pour la parallélisation. Les premières tentatives n'ont pas tout-à-fait porté fruit puisque l'*overhead* et le travail requis lors de la création des fils et de l'exécution des tâches était, jusqu'à un certain point, autant ou plus lourd que le travail effectué en séquentiel. Nous avons finalement trouvé une façon que nous croyons correcte.

Bref, lorsque le programme fait appel à notre mécanisme de calcul du prochain coup à jouer, la grille possède un certains nombre de cases vide restantes. Lorsque vient le temps de partager le travail, chaque tâche correspondra à la recherche (d'une profondeur P-1) dans le sous-arbre correspondant à l'un de ces coups restants.

Par exemple, dans le cas d'une grille 2x2 (simplicité visuelle), en début de partie, avec une profondeur P :

Les cases vides (coup à jouer restants) seront {0,1,2,3}. Chaque tâche recevra la même grille, avec son propre coup à jouer. Il y aura donc 4 tâches au total. La 1ere recevra le coup 0, et devra rechercher, avec une profondeur P-1, le meilleur coup à jouer dans les coups {1,2,3} et leur sous-arbre respectif. Lorsqu'une tâche se termine, elle retourne au créateur la valeur la plus grande qu'elle a trouvé dans son arbre. Le créateur devra donc comparer toutes les valeurs reçues, et retourner l'action correspondant à la plus grande valeur.



Techniques et outils utilisés pour le « multithreading »

Pour réussir à paralléliser l'algorithme de recherche, nous avons utilisé des outils standards fournis par le langage java.

Lors des diverses tentatives de parallélisation de l'algorithme, nous nous sommes retrouvés devant un nombre plus ou moins important de fils à créer et détruire, à chaque coup. Dans certaines situation, la parallélisation se faisait sur des tâches plus courtes, ce qui impliquait un plus grand nombre de fils créés à chaque coup. Dans d'autre cas, comme dans le cas final, il fallait créer un nombre de fils correspondant au nombre de cases restantes. Bref, comme le nombre de fils à créer variait souvent, mais qu'il demeurait grand, nous avons finalement opté pour un «thread pool ». Après avoir tenté d'implémenter nous même notre thread pool léger avec seulement les fonctionnalités de base requises, nous nous sommes rendus compte que le temps épargné était négligeable, comparativement à l'utilisation de thread pools plus fiable offerts par java.

Dans notre cas, il fallait non seulement que chaque fil effectue un travail, mais il devait aussi

```
ExecutorService threadPool = Executors.newFixedThreadPool(8);
CompletionService<int[]> pool = new ExecutorCompletionService<int[]>(threadPool);
                        .........
// créer la tâche qui sera éxécutée par le thread
AlphabetaTask task = new AlphabetaTask(workerGrille, coup, alpha, beta, p-1, joueur);
// envoyer la tâche dans le thread pool
pool.submit(task);
                        .........
// recevoir un résultat
results[i] = pool.take().get();
// fermer le threadpool et tuer les threads à l'intérieur
threadPool.shutdown();
```

retourner un résultat à la fin de sa recherche.

Un objet ExecutorService offre des méthodes pour contrôler la création d'un pool de fils ainsi que sa destruction. Nous créons ici un threadpool de taille fixe (8).

Au départ, pour recevoir la réponse des fils éventuellement, il fallait créer un objet de type Future<T> pour chaque réponse prévue. Cependant, suite à quelques recherches, nous sommes tombés sur la classe CompletionService, qui offre une façon plus élégante pour envoyer une tâche au pool et pour recevoir le résultat.

Puis, pour chaque tâche à exécuter, nous créons un objet AlphabetaTask (classe que nous nous sommes créée) en lui passant les paramètres correspondant à sa recherche à effectuer, et envoyons cet objet dans le pool grâce à la méthode submit(task). L'ExecutorService se chargera de distribuer chaque tâche à un fil libre, ou de la mettre dans une task-queue si tous les fils sont occupés. Lorsqu'un thread redevient libre, il se voit attribuer une tâche. Lorsqu'une tâche Alphabeta est enfin exécutée, sa méthode call() est appelée.

Par la suite, une fois tout le travail terminé et les résultats récoltés grâce à la méthode .take().get() du CompletionService, il faut fermer le pool avec la méthode shutdown() du ExecutorService, ce qui a pour effet de tuer tous les fil à l'intérieur.

Problèmes rencontrés

Quoi que la parallélisation d'un algorithme peut sembler facile, nous avons fait face à plusieurs problèmes.

Les problèmes reliés à la division des tâches ont déjà été abordés alors nous n'en parlerons pas ici. Il est bon cependant d'ajouter que la base de l'implémentation de l'algorithme a dû être modifiée puisque la façon dont il était appelée récursivement s'adaptait mal à une implémentation parallèle.

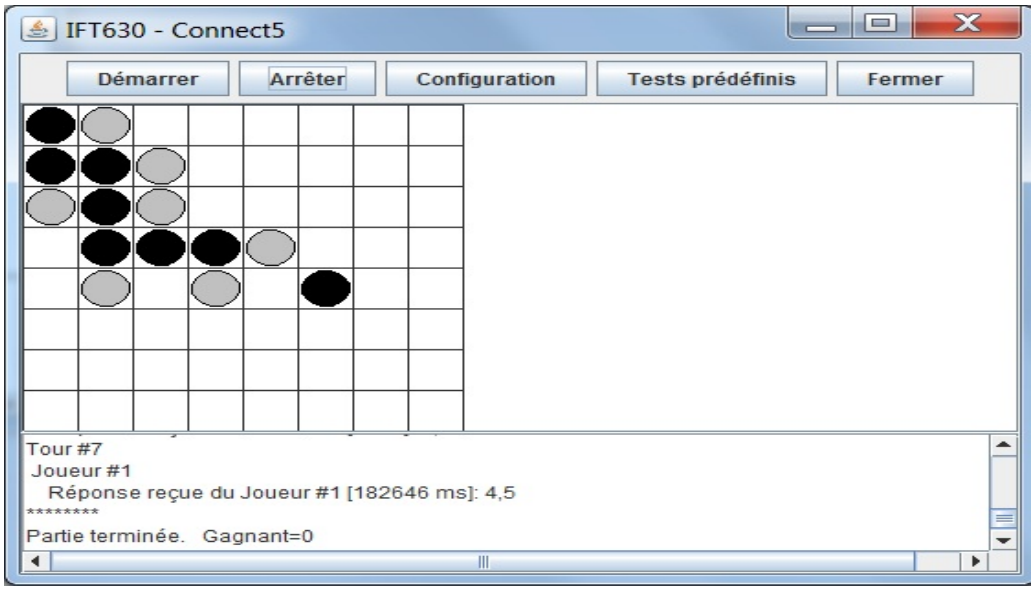
Au niveau des outils, ce ne sont pas tant des problèmes que nous avons rencontrés mais plutôt des déceptions quant à l'amélioration faible des performances. *L'overhead* était souvent trop lourd pour le gain de performance.

Finalement, nous avons eu des problèmes de concurrence indésirables à cause d'erreurs de notre part dans l'utilisation du langage java. Lors du passage des paramètres aux fils (plus précisément à leur tâche Alphabeta respective), nous nous sommes rendu compte que nous passions certaine valeurs par pointeur plutôt que par valeur. La grille du jeu en est le meilleur exemple. Nous faisions pourtant des clones() de nos variables passées, croyant bêtement que chaque fil travaillerait sur sa propre copie de ces objets. Nos fil manipulaient donc parfois la même grille, ce qui entraînait des énormes erreurs dans le calcul des valeurs de grille, et ces erreurs étaient parfois difficiles à repérer. Cependant, des tests approfondis nous ont permis de cibler nos erreurs, et nous avons pu remédier à la situation.

Interface

Afin de jouer au jeu, nous avons créé une interface simple et facile d'utilisation. Nous sommes partis d'une base qui avait été développée par Éric Beaudry pour le cours d'intelligence artificielle et y avons apporté plusieurs modifications afin de l'adapter aux fonctionnalités de notre projet et à nos goûts personnels.

Plateforme de jeu



La plateforme de jeu est très simple. Il s'agit de la grille sur laquelle les coups sont joués, d'une zone de texte où sont affichées les informations des coups joués et des boutons pour démarrer, arrêter, configurer, tester ou fermer la partie.

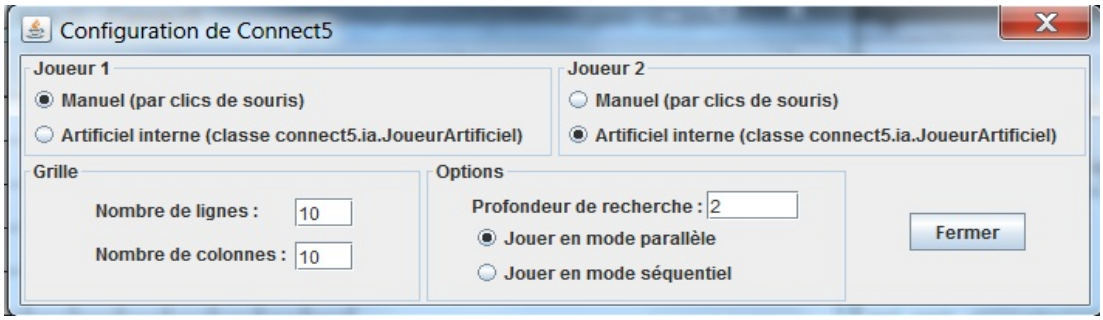
Configuration

Afin de permettre à l'utilisateur de configurer manuellement les différents paramètres du jeu et de la recherche du coup à jouer par l'intelligence artificielle, nous avons intégré un menu de configurations.

D'abord, il est possible de choisir le type de joueur désiré, et ce, pour les deux joueurs. Les choix disponibles sont le joueur manuel, c'est-à-dire que la personne joue elle-même les coups, et le joueur artificiel.

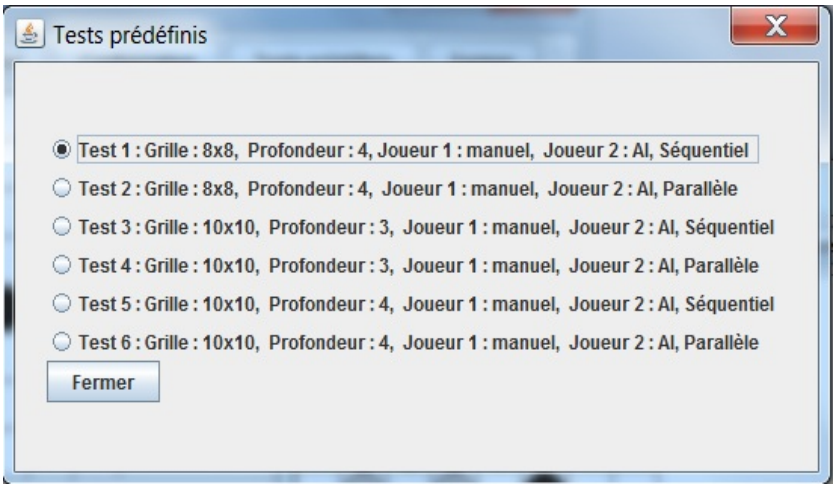
Il est aussi possible de définir la dimension de la grille de jeu en entrant les deux valeurs (hauteur et largeur) et de choisir la profondeur de recherche pour le joueur artificiel, c'est-à-dire le niveau de précision des coups joués.

Enfin, l'utilisateur a le choix entre un algorithme de recherche séquentiel et parallèle (multi-fils).



Tests prédéfinis

Nous avons ajouté à l'interface une section dans laquelle sont définis plusieurs tests sous forme de boutons radios, permettant à l'utilisateur de choisir en un clic le test à effectuer. Elles offrent un choix quant à la dimension de la grille, la profondeur de la recherche dans l'arbre, le rôle joué par l'intelligence artificielle (soit joueur 1 ou joueur 2), et finalement le type d'algorithme utilisé (séquentiel ou parallèle). Cela a grandement accéléré le processus de test et de déboguage, tout en assurant une cohérence une niveau des choix d'un test à l'autre.



Rapport de tests

Voici une liste des résultats obtenus suite à une série de tests effectués avec différentes configurations du jeu (tests prédéfinis), joueur 1 humain et joueur 2 AI, en début de partie (grille vide).

Tableau 1 : Intel® Core™ i7 2670QM @ 2.20GHz

Test	Taille de la grille	Profondeur	Séquentiel	Parallèle 8 fils
1	6x6	2	4 ms	6 ms
2	10x10	3	3078 ms	886 ms
3	12x12	3	12730 ms	3620 ms
4	10x10	4	284986 ms	87008 ms

Tableau 2 : Intel®Pentium® Dual-Core CPU T4200 @ 2.00GHz

Test	Taille de la grille	Profondeur	Séquentiel	Parallèle 8 fils
1	6x6	2	78 ms	93 ms
2	10x10	3	4772 ms	2764 ms
3	12x12	3	19868 ms	10950 ms
4	10x10	4	454676 ms	241640 ms

Suite à ces tests, nous pouvons conclure qu'il y a une nette amélioration de la vitesse du calcul plus la profondeur de recherche est grande. Cependant, lorsque la recherche est très courte (premier test) on remarque que *l'overhead* associé à la création des fils et distribution des tâches ralenti le processus.

Conclusion

Pour terminer, les objectifs du projet ont été atteints. Il est clair que l'amélioration de la performance engendrée par la parallélisation de l'algorithme de recherche est très marquée. Plus le temps pris par l'algorithme séquentiel est grand, plus l'algorithme parallèle montre ses avantages. Si nous implémentions la technique « iterative-deepening » pour permettre à notre algorithme d'être utilisé dans le cadre d'une partie réelle contre un adversaire quelconque, la réponse trouvée dans le temps permis serait plus précise car l'algorithme aurait le temps de parcourir l'arbre avec une profondeur plus grande. Notre joueur artificiel est donc beaucoup plus performant et intelligent qu'il l'était lorsque nous avons pris le projet en charge.

Plusieurs autres techniques de parallélisation pourraient possiblement être utilisées, ainsi que plusieurs autres algorithmes de recherche reliés au jeux à deux adversaires. Cependant, pour le temps que nous avions, et tenant compte du fait que le projet est dans le cadre du cours IFT630 et non d'intelligence artificielle, nous croyons nos résultats suffisants.

Ce projet a servit de travail d'intégration de plusieurs connaissances apprises lors du cours et nous a permis de mieux comprendre un grand nombre de concepts. Quoi que la programmation multi-fils présente ici est plutôt simple au niveau de la communication, elle nous a permis de réfléchir à une bonne façon de paralléliser un algorithme et est malgré tout fiable et efficace.

Remise du projet

Nous vous remettons le projet en format .jar. Comme nous utilisons simplement la technologie multifil de base intégrée au langage java, vous n'aurez aucune librairie(ou autre) à installer et configurer.