
Projet Amazons

Tanguy Daponte - Abdurahman El Califa Kan It Bensaidi - Louis
Peyrondet - Maximilien Vidiani

EI6PR105 Projet de programmation impérative

Responsable : David Renault

Encadrant : Jean-François David

Département Informatique

S6 - Année 2022/2023



Contents

1	Introduction	2
2	Structure	3
2.1	Serveurs	3
2.2	Clients	3
2.3	Tests unitaires	4
2.4	Graphe des dépendances	4
3	Client aléatoire	5
3.1	Première itération	5
3.2	Deuxième itération	5
3.3	Optimisation potentielle	5
4	Client intelligent	5
4.1	Les réseaux de neurones	5
4.2	La recherche en profondeur	6
5	La fonction d'évaluation	9
5.1	La liberté des reines	9
5.2	L'accessibilité des reines	10
5.3	La répartition des reines dans le tableau	11
5.4	Optimisations	12
6	Conclusion	15

1 Introduction

Le sujet de ce projet porte sur la réalisation d'une implémentation du jeu des Amazones, puis de la réalisation de clients utilisant des stratégies optimales.

Chaque joueur possède un nombre prédéfini de reines qui dépend de la taille de l'échiquier sur lequel prend place la partie. L'échiquier prend une des formes suivantes "Carré", "Donut" ou encore "Trèfle"

Chaque reine peut se déplacer de la même manière qu'une reine dans le jeu d'échecs, puis tire une flèche qui condamne la case ciblée. Le but du jeu est de bloquer toutes les reines adverses grâce aux flèches tout en protégeant ses reines. La partie se termine lorsque l'un des deux joueurs n'a plus aucun coup valide à jouer. Il est impossible d'avoir un match nul.

2 Structure

Le projet utilise une architecture client/serveur. Leurs rôles ainsi que leurs spécificités sont détaillés ci-dessous.

2.1 Serveurs

Le rôle du serveur est dans un premier temps, de charger les différentes bibliothèques dynamiques représentant les joueurs, et dans un second temps de superviser la partie en vérifiant que les coups joués sont bien valides et que la partie n'est pas arrivée à son terme. Le serveur a pour but de faire la liaison entre les deux clients en leur communiquant le dernier coup joué par leur adversaire.

Notre projet comporte deux serveurs différents. Le fichier *server.c* implémente un serveur classique permettant de faire jouer deux clients entre eux. Le fichier *arena.c* quant à lui, permet de jouer un panel de parties et d'en tirer un certain nombre d'informations afin de tester les différentes stratégies de nos clients.

2.2 Clients

Les clients ont la forme de bibliothèques dynamiques. Chaque client possède la même interface afin de pouvoir communiquer avec le serveur mais étant donné que leur stratégie diffère, ils possèdent une implémentation unique. Le plateau sur lequel la partie se déroule n'étant pas public, chaque client doit se charger de gérer sa propre version du plateau en le mettant à jour à chaque coup de l'adversaire, la liaison étant faite via le serveur. Chaque client possède une fonction *play* qui renvoie au serveur son prochain coup.

Nous avons créé plusieurs clients, chacun possédant une stratégie différente, ce qui implique que chaque client utilise une structure qui lui est propre. Ils partagent néanmoins la caractéristique d'utiliser leur structure comme une variable globale. Ce dernier point peut être problématique dans le cas où l'on souhaite faire jouer le client contre lui-même. Dans ce cas précis, lorsque la bibliothèque dynamique est chargée par le serveur, la variable globale du client est chargée en mémoire, plus particulièrement dans le segment de données¹. Lorsque le second client est chargé, la bibliothèque étant déjà chargée en mémoire, les fonctions sont partagées sans avoir à allouer plus de mémoire cependant le segment de données est commun donc la variable globale est partagée entre les deux clients ayant pour conséquence l'impossibilité de l'exécution.

Une des solutions trouvée pour traiter ce problème a été de faire une copie de la bibliothèque compilée avec un nom différent ce qui permet lors du chargement du second client d'avoir sa propre zone mémoire avec son propre segment de données.

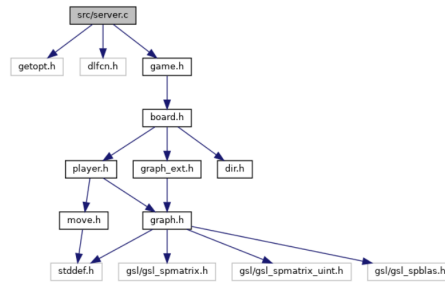
¹https://en.wikipedia.org/wiki/Data_segment

2.3 Tests unitaires

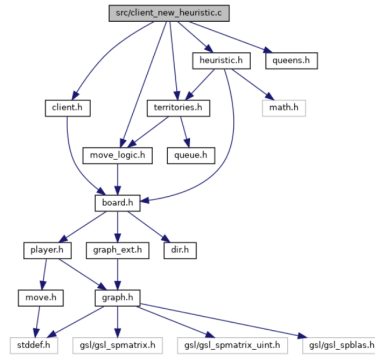
Afin d'assurer le coverage, nous avons inclus un système de test automatisé. Le fait d'avoir une option permettant d'exécuter rapidement tous les tests, nous permet après chaque modification, de réduire les erreurs et de maintenir la qualité du code sur le long terme, notamment pendant les phases d'optimisation. D'un autre côté notre exécutable `alltest` nous permet aussi d'exécuter les tests un par un en donnant le nom du test en argument de l'exécutable. On pouvait ainsi isoler le lancement d'un test, ce qui facilitait le débogage.

2.4 Graphe des dépendances

Les graphes de dépendances du serveur et d'un client montrent les différents modules et niveaux d'abstractions du projet. Le module *graph* est une couche d'abstraction de la bibliothèque `GSL`² qui offre une interface simplifiée pour l'utilisation de graphes implémentés via des matrices d'adjacence créées à l'aide de `GSL`. Toute cette notion de graphe est par la suite également abstraite par le module *board* qui sépare le projet afin de fournir au reste des modules une interface de manipulation de plateau de jeu plus intuitive.



(a) Server



(b) Client

Figure 1: Graphes des dépendances

²<https://www.gnu.org/software/gsl/>

3 Client aléatoire

Le premier client que nous avons implémenté, jouait des coups aléatoires, il avait pour objectif de tester notre serveur.

3.1 Première itération

Lors de la première itération du client, nous avons implémenté une méthode naïve qui consistait à générer des coups complètement aléatoires puis à vérifier si le coup est valide ou non. Cette première approche fonctionnait convenablement sur des plateaux d'une dimension de l'ordre de 10 par 10 mais dès lors que la dimension du plateau augmentait, le calcul des coups en fin de partie avait tendance à prendre trop de temps.

3.2 Deuxième itération

La deuxième itération du client implémentait une nouvelle structure *queen_moves* qui permet de stocker les coups possibles pour une reine à un indice donné. Le client choisit alors aléatoirement une reine parmi celles du joueur, utilise une fonction permettant d'obtenir les coups possibles pour cette reine et choisit aléatoirement un de ces coups. Dans le cas où la reine choisie ne peut plus se déplacer, le client choisit une autre reine aléatoirement.

3.3 Optimisation potentielle

Une optimisation pour cette dernière itération consisterait à utiliser un système de mémorisation permettant de savoir quelles reines sont bloquées afin de ne pas avoir à calculer une nouvelle fois les coups d'une reine bloquée. Cette optimisation serait surtout intéressante dans le cas où le nombre de reines deviendrait important.

4 Client intelligent

Une fois notre environnement bien testé, le but était maintenant de créer un client pouvant jouer, dans l'objectif de remporter les parties contre ses adversaires. Il doit, en fonction de la situation du plateau, jouer un coup particulier, ce qui représente une certaine forme d'intelligence.

Face à la multitude de stratégies qui s'offraient à nous, nous avons dû faire des choix.

Comme indiqué au début de la page du sujet, la résolution du jeu des Amazones a été un sujet de recherche qui a intrigué dans le domaine de la théorie des jeux combinatoire. Après quelques recherches nous avons remarqué que la majorité des joueurs intelligents étaient implémentés grâce à 3 méthodes.

4.1 Les réseaux de neurones

Les réseaux de neurones sont des modèles de machine learning utilisés pour apprendre des stratégies à partir de données d'entraînement. Étant donné nos

compétences dans le domaine et la durée du projet assez limitée nous avons assez vite abandonné cette idée.

Cependant nous avons remarqué que cette technique aurait pu être redoutable. Étant donné que le machine learning se nourrit de données, nous aurions pu attendre que toutes les équipes présentent leurs clients, puis récupérer le comportement de leurs clients, pour entraîner le nôtre.

Tout ça grâce à la section "moves" du visionneur web du ladder, qui nous présentait ces données sur un plateau d'argent ;

Moves	<pre>[{"qsrc":4,"qdst":22,"adst":19}, {"qsrc":29,"qdst":17,"adst":3}, {"qsrc":1,"qdst":15,"adst":13}, {"qsrc":24,"qdst":12,"adst":2}, {"qsrc":11,"qdst":26,"adst":27}, {"qsrc":12,"qdst":7,"adst":0}, {"qsrc":15,"qdst":21,"adst":11}, {"qsrc":34,"qdst":28,"adst":23}, {"qsrc":26,"qdst":32,"adst":8}, {"qsrc":7,"qdst":1,"adst":7}, {"qsrc":21,"qdst":16,"adst":21}, {"qsrc":31,"qdst":24,"adst":12}, {"qsrc":32,"qdst":31,"adst":25}, {"qsrc":28,"qdst":29,"adst":28}, {"qsrc":16,"qdst":10,"adst":16}, {"qsrc":29,"qdst":34,"adst":29}, {"qsrc":31,"qdst":33,"adst":30}, {"qsrc":34,"qdst":35,"adst":34}, {"qsrc":33,"qdst":32,"adst":31}, {"qsrc":24,"qdst":18,"adst":24}, {"qsrc":10,"qdst":4,"adst":10}]</pre>
--------------	---

4.2 La recherche en profondeur

La recherche en profondeur ou recherche alpha-bêta, consiste à explorer de manière exhaustive, dans un arbre minmax, l'ensemble des coups possibles à partir de l'état de jeu actuel, en utilisant une fonction d'évaluation pour évaluer la qualité des positions atteintes par chaque coup.

Après avoir discuté avec les étudiants de deuxième année, nous avons pensé que ce serait la technique la plus adaptée au problème. L'algorithme est relativement simple à implémenter et nous possédions une fonction d'évaluation, qui paraissait prometteuse.

Pour tirer le plus grand avantage de cette technique, il faut pouvoir explorer la partie sur un certain nombre de coups, et donc avoir une profondeur d'arbre la plus grande possible.

Cependant, la quantité de coups jouables en début de partie, représente une explosion combinatoire, qui rend la mise en pratique de cette méthode assez compliquée: Pour un plateau de taille 10, avec donc 8 reines, environ 30 cases accessibles par reine, donc 900 coups possibles par reine, 7200 coups par joueur, et donc $7200 \times 7200 =$ plus des 51 millions de coups pour un arbre de hauteur 2.

Une recherche en profondeur exhaustive paraissait donc compliquée étant donnée la limite de temps d'une partie. De plus le temps de calcul de notre fonction d'évaluation était non négligeable 5.4. Dans ce contexte nous avons décidé d'utiliser 2 optimisations, mais toujours en essayant de se rapprocher de l'exhaustivité du min_max.

Réduire la profondeur de jeu

Le problème d'un algorithme minmax, comme expliqué précédemment, est que le nombre de noeuds augmente de manière exponentielle, en fonction du nombre

de coups joués. Ayant confiance en la pertinence de la fonction d'évaluation nous avons décidé de jouer sur une seule hauteur, sous forme d'une double boucle for qui calcule l'heuristique pour chaque coup jouable, à partir d'un board donné.

La première boucle calcule chaque coup possible pour chaque reine, et la seconde calcule chaque flèche qui peut être tirée à partir d'un coup de reine joué.

Un problème que nous avons rencontré est que lorsque l'heuristique est égale pour 2 coups (nous gardons le meilleur coup actuel), les coups calculés en premier ont tendance à être plus joués que ceux en dernier. Pour y remédier nous avons changé la condition de changement de la valeur de l'heuristique:

```
if (board_heuristic > best_move_heuristic ||
    (board_heuristic == best_move_heuristic && rand()%3==0))
```

En rajoutant de l'aléatoire on évite que les coups que nous jouons soient orientés en fonction de l'ordre des coups possibles, lors de leur calcul.

Réduire le nombre de coups calculés par reine

Suite à quelques tests nous avons remarqué que le placement de la reine avait un impact assez important. C'est pour ça qu'au lieu de lancer une évaluation sur tous les coups de reine, nous lançons, le calcul des tirs de flèche sur uniquement les coups de reine les plus prometteurs.

Algorithme de Monte Carlo

Le Monte Carlo Tree Search (MCTS) est un algorithme de recherche arborescente largement utilisé dans les jeux et autres applications de prise de décision. L'idée principale de l'algorithme est de simuler des parties aléatoires de la situation à résoudre pour construire un arbre de recherche qui reflète les différentes actions possibles et leurs conséquences.

L'algorithme MCTS commence par construire un arbre de recherche à partir de l'état initial de la situation à résoudre. Il utilise ensuite une politique d'exploration appelée UCT (Upper Confidence Bound) pour sélectionner une action à considérer en tenant compte à la fois des connaissances actuelles sur l'état de la situation et de l'arbre de recherche. Si le noeud sélectionné n'est pas un noeud terminal, l'algorithme crée alors ses descendants auxquels sont associés des coups aléatoires. Ensuite, l'algorithme simule un certain nombre de parties aléatoire à partir de chaque fils créé. Ces simulations sont d'abord effectuées de manière totalement aléatoire.

Enfin, les résultats sont remontés jusqu'à la racine de l'arbre, ce qui met à jour les valeurs UCT, et une nouvelle phase de sélection est entamée. Ce processus est répété jusqu'à ce qu'un certain critère d'arrêt soit atteint, tel qu'un nombre maximal d'itérations ou une limite de temps.

L'algorithme MCTS est particulièrement utile pour les situations dans lesquelles l'espace de recherche est trop grand pour être exploré en utilisant des méthodes de recherche complètes telles que l'algorithme Minimax. En utilisant des simulations aléatoires pour explorer l'espace de recherche, l'algorithme

MCTS peut rapidement identifier les actions les plus prometteuses et fournir une bonne approximation de la solution optimale.

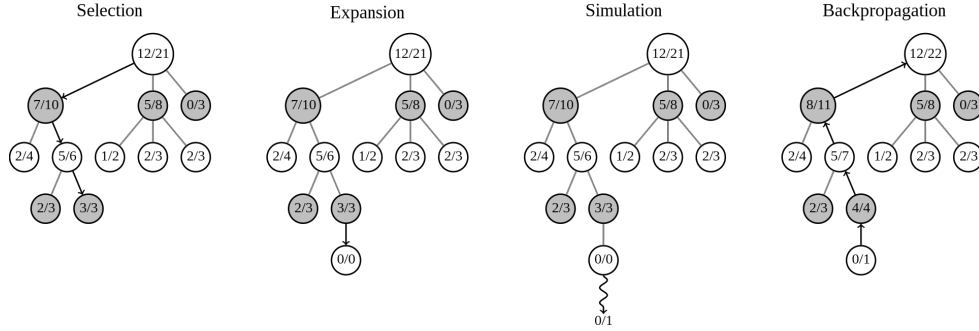


Figure 2: Exemple d'application de la recherche arborescente Monte-Carlo (MCTS) en utilisant l'algorithme UCT. [2].

Pour créer un client MCTS performant, nous avons développé une structure d'arbre qui alloue dynamiquement les nœuds nécessaires en fonction des étapes précédentes du MCTS. Cependant, étant donné que le nombre de coups possible est exponentiel, notre MCTS doit générer un grand nombre de nœuds et effectuer de nombreuses simulations, ce qui ralentit considérablement l'exécution. Afin d'optimiser notre structure d'arbre, nous avons adopté une stratégie de pointeur unique pour tous les nœuds, pointant vers le même plateau de jeu, ce qui évite de copier le plateau dans chaque nœud.

Nous avons également mis en place plusieurs stratégies pour optimiser l'algorithme MCTS. Tout d'abord, nous avons utilisé une heuristique pour la phase d'expansion plutôt que de générer les coups aléatoirement, ce qui permet des recherches plus efficaces dans l'arbre, malgré le coût en temps de l'évaluation du coup. Ensuite, étant donné le grand nombre de possibilités pour le jeu des amazones, nous avons limité la profondeur de l'arbre en fonction de l'avancement de la partie, car les probabilités de suivre un certain ordre de coups aléatoire à de grandes profondeurs sont proches de zéro. Enfin, nous avons implémenté une stratégie de recyclage de l'arbre, ou si le dernier coup de l'adversaire avait été généré pendant l'expansion, nous pouvions recycler l'arbre précédemment généré pour obtenir de meilleurs résultats.

Cependant, malgré toutes ces optimisations, nous avons constaté que MCTS restait trop lent pour des plateaux de taille supérieurs à 10. Pour remédier à cela, nous aurions opté pour une solution de locked-tree, où un arbre avec une structure préétablie est pré-généré. Les coups associés à chaque nœud sont modifiés à chaque itération, puis des simulations aléatoires ou avec heuristique sont effectuées pour chaque feuille, et la backpropagation est appliquée. Cette solution permet d'éviter l'allocation dynamique et la phase de sélection, mais elle s'éloigne de la philosophie de l'algorithme MCTS pour se rapprocher de celle de l'algorithme min-max.

5 La fonction d'évaluation

La fonction d'évaluation est une fonction qui prend en entrée un plateau et qui nous donne en sortie une valeur qui quantifie l'avantage que ce plateau présente pour notre joueur. Plusieurs paramètres sont pris en compte pour ce calcul.

5.1 La liberté des reines

Suite à une discussion avec M. Renault, la première idée qui nous est venue en tête est que la mobilité d'une reine est un facteur clé au cours d'une partie. En effet plus nos reines seront mobiles plus nous aurons de l'impact sur le plateau. L'idée de départ consistait à définir la mobilité d'une reine, comme le nombre de cases disponibles au tour d'une reine. Cependant cette technique a fini par nous apparaître comme trop restrictive, l'analyse se fait localement autour de chaque reine, alors que chaque reine peut avoir un impact sur les autres et sur tout le plateau.

Cela nous a poussé à penser chaque case vide comme un chemin pouvant être emprunté par n'importe quelle reine, et de lui trouver une valeur lui désignant une "puissance". Cette puissance représentant, dans le cas où la case soit bloquée (par une reine ou une flèche), le nombre de cases qui ne seraient plus disponibles pour les reines pour une équipe donnée.



							
				7			
1	2	3	4	11	6	7	
				5			
				4			
				3			
				2			
				1			

Figure 3: Valeur des cases pour l'heuristique de puissance des cases. La case de valeur 11 est la somme de la valeur de puissance des 2 reines (6+5).

Cette figure représente la puissance de certaines cases, pour ces 2 reines, que l'on va supposer ennemies. Ici, dans l'optique de réduire la mobilité, le coup le plus judicieux paraît être d'occuper la case de puissance 11, ce qui empêcherait les reines de se déplacer aux cases disponibles derrière ces reines.

Cette idée à bien fonctionner pour notre premier client intelligent, cependant en voyant que les thèmes abordés en suivant paraissaient plus intéressants nous avons laissé cette idée de coté.

5.2 L'accessibilité des reines

Une seconde variable très importante est la possession des cases pour chaque reine. Cette possession est définie, pour chaque case vide X et pour chaque joueur J , comme le nombre de coups minimaux devant être joué par J pour atteindre X .^[1]

Voici une figure tirée de la première publication [1] dans la page du sujet, avec les distances de la reine noire en bas des cases et les distances des rois (d'échecs) en haut des cases:

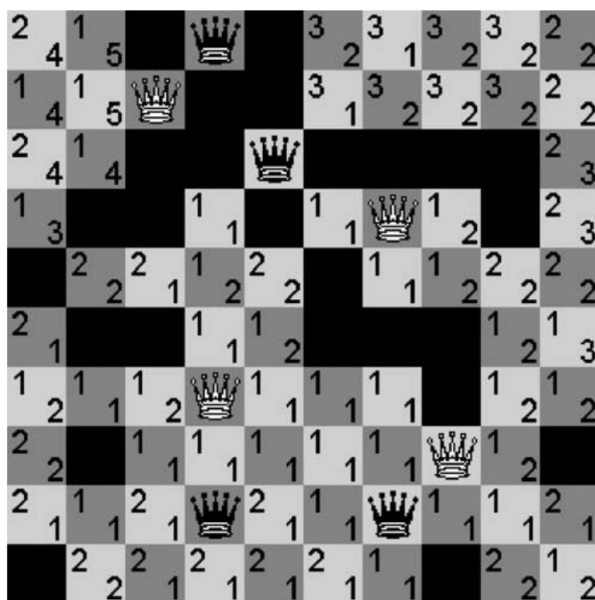


Figure 4: Territoires avec mouvements de reine, en milieu de partie

Afin de représenter ces distances, nous avons décidé d'avoir un tableau de la même taille que le board, pour chaque joueur, avec dans chaque case du tableau, le nombre de coups minimaux à jouer pour accéder à cette case par le joueur, si elle est accessible et `UINT_MAX` sinon.

La création du tableau

Pour pouvoir cartographier ces distances il nous fallait parcourir au moins une fois toutes les cases du tableau, accessibles à partir des reines. Pour cela nous avons choisi un parcours en largeur qui suit les règles de déplacement des reines, ainsi, en partant de chaque reine à l'aide d'une file on met à jour les cases, étant initialisées à `UINT_MAX`.

Le parcours en profondeur était aussi une solution, mais beaucoup moins efficace. Le principal problème est qu'il peut explorer les chemins plus longs et inutiles avant de découvrir les chemins les plus courts, pour une même itération de reine. Cela augmente le temps de calcul et les ressources nécessaires pour résoudre le problème.

5.3 La répartition des reines dans le tableau

Pour finir le dernier point sur lequel nous sommes concentrés, c'est la répartition des reines sur le tableau. Nous avons remarqué que certaines parties se terminaient avec des reines ayant à peine bougé, jouer une partie avec un nombre réduit de reines paraît bien moins avantageux que de jouer avec toutes les reines.

Afin d'y remédier l'article [1], proposait de calculer les tableaux des territoires avec des mouvements de roi. Cependant les résultats ne nous paraissaient pas assez concluants. Même tableau pour les distances de rois :

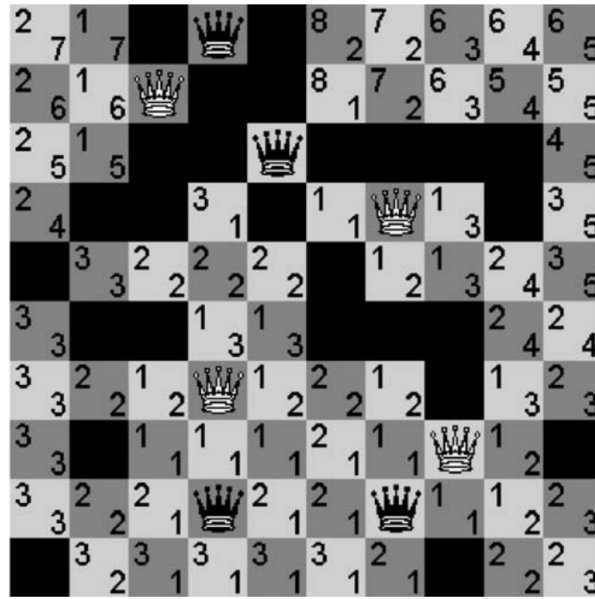


Figure 5: Territoires avec mouvements de roi, en milieu de partie

La solution que nous avons trouvée, a été de créer une structure `territory_cell`, contenant une distance, et un indice de reine. Ainsi au lieu de renvoyer un tableau des distances minimales, on renvoie un tableau de cette structure, contenant ces mêmes distances minimales et l'indice de la reine à laquelle cette distance correspond.

Ce qui permet ensuite, à la fin, de parcourir le tableau créé, et de compter la quantité de cases que chaque reine possède, afin de calculer la variance de ces quantités. Le but étant de minimiser cette variance, et donc d'avoir un nombre de cases possédées similaire pour chaque reine. On trouve ainsi une manière de répartir les reines de manière homogène sans augmenter le temps de calcul.

Dans cette partie les tests ont été réalisés grâce aux figures de l'article, qui présentent un exemple des valeurs des cases en fonction d'un placement des reines, qui paraît être en milieu de partie. Nous les comparons simplement aux valeurs calculées par nos fonctions.

5.4 Optimisations

Malgré le fait que nous avons choisi un algorithme avantageux, un parcours en largeur en partant de chaque reine, il y avait toujours des problèmes de timeout dans le ladder étant donné que le calcul de l'heuristique est lourd. En effet le premier parcours en largeur engendré par la première reine parcourt tout le plateau vu que toutes distances sont initialisées à `UINT_MAX`. Le parcours de la deuxième reine va par la suite écraser une majorité des valeurs, et ainsi de suite pour toutes les reines.

Avec ce type de parcours 5.2, beaucoup de cases étaient visitées plusieurs fois, ainsi on avait, avec R le nombre de reines et N le nombre de cases libres dans le tableau, une complexité dans le pire des cas de $O(R*M)$.

Parcours en largeur

Pour remédier à cette complexité nous avons changé notre algorithme, au lieu de parcourir le plateau en partant de chaque reine, nous partons de toutes les reines au même temps.

Ainsi toutes les cases pouvant être jouées après en X coups par un joueur, sont marquées avant de marquer les cases pouvant être jouées en $X+1$ coups.

Après l'optimisation, comme indiqué les cases sont rarement modifiées plus d'une fois, on a donc une complexité proche de N , soit presque du linéaire en nombre de cases du plateau.

Tableau de booléen pour les positions des reines

Après avoir généré et étudié l'arbre d'appel de notre programme en utilisant `KCacheGrind`³, il est apparu que l'une des fonctions les plus appelées est *queens_occupy* qui vérifie si l'indice donné est occupé par une reine ou non. En effet dès lors qu'un calcul est effectué sur le plateau il faut s'assurer, ou non, que la case est libre. Le principe de la fonction est simple, l'indice est comparé à un tableau où est stockée la position de chaque reine d'un joueur. Ainsi sa complexité est linéaire de l'ordre de $\theta(n)$, n étant le nombre de reines d'un joueur, donc pour vérifier si une case est libre sur un plateau 10×10 , la complexité est $2 * \theta(8)$. Pour réduire cette complexité linéaire en temps constant, l'utilisation d'un tableau de booléen est possible où chaque indice du tableau indique si une reine se trouve à cet indice sur le plateau. Le revers est une complexité en espace un peu plus importante, de $2 * \theta(nb_reines)$ à $\theta(taille_plateau)$, mais qui vaut largement le compromis.

³<https://kcachegrind.github.io/html/Home.html>

Flag de compilation

Lors du développement du projet nous utilisons le flag de debug de gcc `-g3` qui permet d’avoir plus de précisions lors de l’utilisation, par exemple, de gdb. Cependant ces options de compilations ne sont utiles que pour la phase de développement, ainsi après avoir discuté avec M. Renault, nous avons pu faire ajouter sur le ladder une variable d’environnement `TURBO` initialisé à `true` lors de la compilation pour le ladder ce qui permet de modifier les règles de compilation du Makefile en conséquence et de supprimer les options de debug et d’ajouter les optimisations du compilateur gcc avec le flag `-O3`.

Pré-calcul des directions et mouvements

Afin de pouvoir continuer l’optimisation du programme, nous avons surmonté les limitations de *gprof*⁴, un profiler pour programme n’utilisant pas de bibliothèque dynamique, et *sprof*⁵, qui est adapté aux bibliothèques dynamiques mais trop peu maintenu donc inutilisable, en convertissant ces dernières en bibliothèques statiques sur une branche de développement secondaire. Les données de gprof ont révélé que 60% du temps d’exécution était consacrés aux fonctions de calcul de directions et de mouvements possibles d’une reine dans le jeu d’Amazons.

Pour la fonction de calcul de direction, nous avons précalculé et mémorisé les directions entre toutes les cases, réduisant ainsi le temps de calcul. La précalculation des mouvements des reines était plus complexe, étant donné la nature combinatoire du jeu. Notre solution a été de précalculer les mouvements possibles sur un plateau vide pour chaque case, puis d’itérer sur ces positions durant une partie pour éliminer les mouvements bloqués par l’état actuel du plateau (flèches ou reines). Cette approche combine l’efficacité du précalcul avec une flexibilité adaptative à l’évolution du plateau de jeu.

⁴https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html

⁵<https://man7.org/linux/man-pages/man1/sprof.1.html>

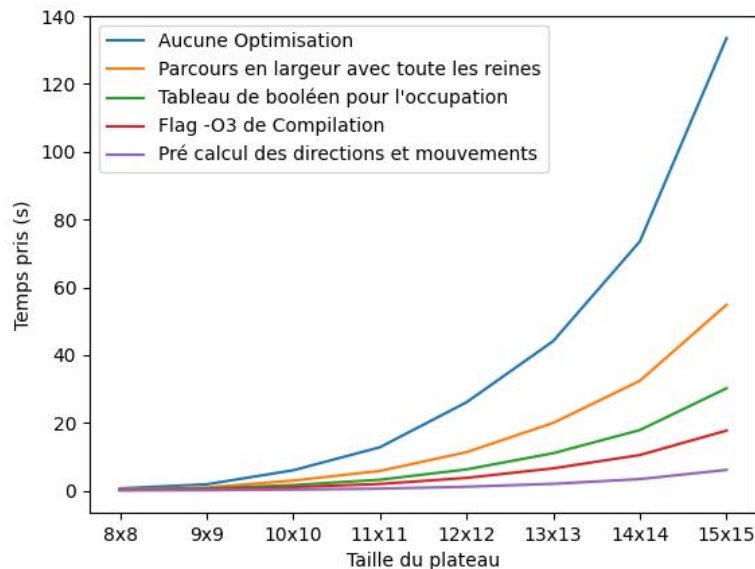


Figure 6: Graphique représentant les temps pris d'une partie pour différentes techniques d'optimisation sur différentes tailles de plateaux carrés pour un client. Le client commence toujours en premier. Le temps est la moyenne sur 25 exécutions indépendantes. Les optimisations sont cumulées à chaque courbe.

En combinant ces optimisations, les clients intelligents peuvent évaluer rapidement tous les prochains mouvements possibles en utilisant nos fonctions heuristiques complexes, leur permettant ainsi de choisir l'action optimale.

Multithreading

Une version multithreaded a également été développée, la répartition des charges s'effectue naïvement où un thread est attribué à chaque reine. Ainsi les calculs de tous les coups possibles pour chaque reine s'effectuent en parallèle. Cependant les tests de performances du projet montrent une dégradation conséquente du temps de calcul. La création et la gestion des threads ne sont pas optimisées, ce qui entraîne des coûts élevés et un déséquilibre dans la répartition des tâches. Pour remédier à cette situation, nous proposons d'implémenter une pool de threads. Cette approche nous permettrait de créer les threads uniquement au début du programme, évitant ainsi les coûts répétés de création et de destruction des threads. De plus, en utilisant une pool de threads, nous pourrions équilibrer de manière plus efficace les tâches en les répartissant entre les threads disponibles dans la pool, améliorant ainsi les performances de la version multithread.

6 Conclusion

La variété des défis et des techniques utilisées ont fait de ce projet un réel défi pour l'équipe. Grâce aux graphes notamment nous avons pu mettre en pratique des connaissances apprises dans d'autres cours.

La création des clients intelligents dans un cadre de compétition où la difficulté du problème dépendait aussi des autres équipes, a été une épreuve très enrichissante techniquement parlant, et nous a fait aborder bien plus de sujets que la consigne prévoyait au départ 5.4.

Au final, nos efforts de réflexion, de développement et d'optimisations ont payé, puisque nous avons souvent terminé en tête du ladder. Cela fait que nous sommes assez fiers de ce projet autant dans ses performances que dans sa qualité de code et de l'expérience qu'il nous a apportée.

References

- [1] Jens Lieberum. An evaluation function for the game of amazons. *Theoretical computer science*, 349(2):230–244, 2005.
- [2] Wikipédia. Recherche arborescente monte-carlo — wikipédia, l'encyclopédie libre, 2022.