IPB
ENSEIRB
MATMECA
BORDEAUX

Rapport de Projet : Amazons the game

Équipe : projetss6-amaz-19048

Auteurs : Youri CHANCRIN, Théo DESCOMPS, Samuel KHALIFA, Jean-Baptiste MARTINEZ

Responsable de projet : David Renault

Enseignant référent : Mihail Popov

Première année, filière informatique
Date : 12 mai 2023

Version actuelle du document : draft

Table des matières

1 Présentation du sujet

L'objectif de ce projet était la création d'un serveur [1] sur lequel des joueurs (*clients*) peuvent jouer au jeu des *Amazones*. Il nous a aussi été demandé d'implémenter quelques clients même si cela était plutôt secondaire. Le tout en utilisant le langage de programmation *C*.

1.1 Le jeu des Amazones

Le jeu des amazons est un jeu de plateau où deux joueurs s'affrontent en jouant tour à tour. Ils disposent chacun de reines [2], qui peuvent se déplacer comme des reines au jeu des échecs. Les reines doivent être espacés d'au moins 2 espaces ou d'un espace en diagonale. Le jeu requiert que les reines soient symmétriques entre les joueurs et qu'il y ait autant de reines sur la ligne externe que sur les colonnes externes.

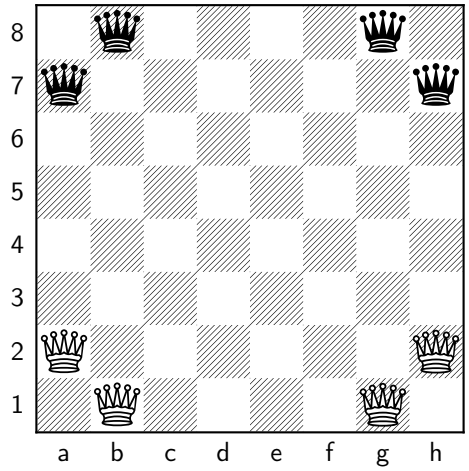


FIGURE 1 – Un jeu des Amazones.

Chaque joueur joue une reine puis tire une flêche depuis la position atteinte par la reine. Les flêches agissent comme une reine dans le jeu des échecs mais ne peuvent traverser de reines ou d'autres flêches. Les reines ne peuvent pas se déplacer sur des flèches ou d'autres reines. Le premier joueur qui ne peut pas jouer un coup valide perd.

Pour répondre à ce besoin, nous avons suivis les spécifications du sujet, mis en place une infrastructure client-serveur, et réalisé plusieurs clients capable de jouer une partie, et souvent de la gagner.

1.2 Spécifications du sujet

Le sujet nous demandait d'implémenter un modèle client-serveur et nous a contraint à utiliser la librairie GSL.

1.2.1 Structure Client Serveur

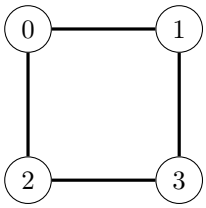
La structure client-serveur, autrement appelée "monolithe", permet de séparer la responsabilité des joueurs et du jeu en lui même. Le serveur représente le jeu. Il organise le déroulement du jeu, retransmet l'information d'un client x aux autres clients et vérifie que chaque coup joué est valide.

1.2.2 Utilisation de GSL

Il nous a été demandé de représenter le plateau de jeu comme un graphe. Cela permet quelques folies pour implémenter moultes plateaux intéressant. Dans les règles des échecs, une case aura au plus

1. Un serveur au sens figuré du terme. Nous n'avons pas eu a faire de réseau, heureusement.
2. Plus exactement $\texttt{floor}(4 * (m/10 + 1))$ où m est la largeur du plateau

8 voisins, cela veut dire que dans le graphe, un noeud aura au maximum 8 voisins, alors qu'un graphe fait au moins 25 noeuds. Pour représenter ce graphe, il nous a été demandé de le représenter par sa matrice d'adjacence. Plus exactement, on stocke la direction entre le sommet i et le sommet j dans la matrice d'adjacence comme montré dans la Figure 2. Pour stocker cette matrice, nous étions contraints d'utiliser la GNU Scientific Library (GSL). Cette librairie permet le stockage et traitement de matrices optimisées autrement appelées *matrices creuses*. Il aurait peut-être été plus judicieux de stocker une liste d'adjacence directement, mais les matrices creuses sous format Compressed Sparse Row (CSR) peuvent s'apparenter à cela.



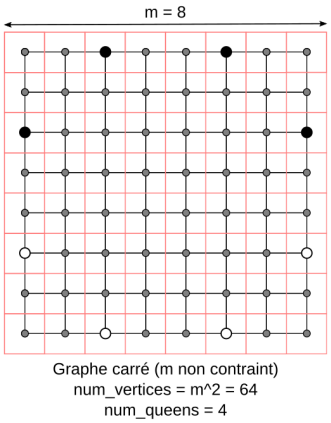
(a) Un graphe 2x2

$$\begin{bmatrix} NO\_DIR & EAST & SOUTH & NO\_DIR \\ WEST & NO\_DIR & NO\_DIR & SOUTH \\ NORTH & NO\_DIR & NO\_DIR & EAST \\ NO\_DIR & NORTH & WEST & NO\_DIR \end{bmatrix}$$

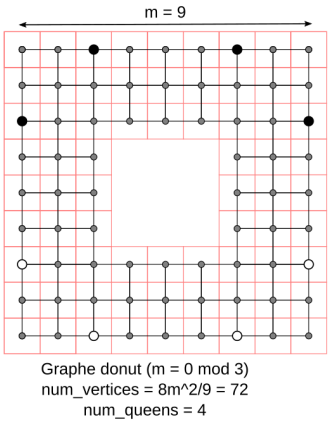
(b) Sa matrice d'adjacence

FIGURE 2 – Exemple de graphe et sa représentation en matrice d'adjacence

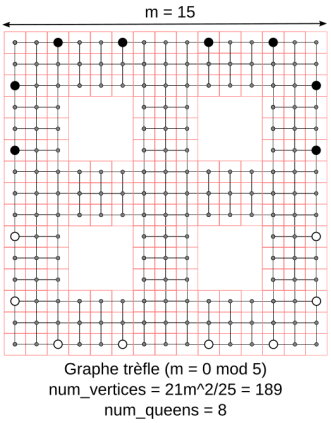
Enfin, le sujet demandait à implémenter les plateaux présenté en figure 3



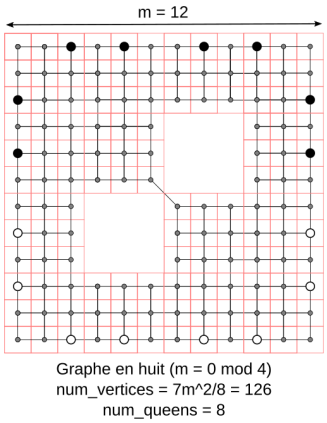
(a) Plateau classique



(b) Plateau donut



(c) Plateau trèfle



(d) Plateau huit

FIGURE 3 – Les différents plateaux à implémenter et leurs graphes

1.2.3 Utilisation de libdl

Chaque client devait être compilé en tant que librairie dynamique. Le serveur devait lancer les clients en tant que librairies dynamiques ouvertes au *runtime* en utilisant la fonction `dlopen` de la librairie `libdl`.

2 Modèle Client-Serveur

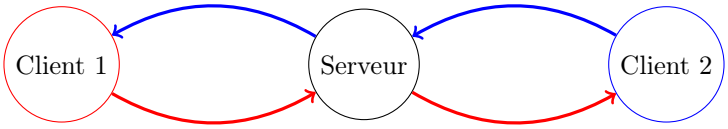


FIGURE 4 – Une représentation abstraite du modèle client-serveur

Comme le montre le graphe dans la figure 4, chaque client x communique au serveur, puis ce que le client x communique au serveur est retranscrit aux autres clients. Cela implique deux choses. Premièrement, un isolement des responsabilités. En effet, le serveur et les clients ne font pas du tout la même chose. Le serveur fait le principal du travail tandis que les clients jouent. Deuxièmement, il faut une langue commune. En effet, si le client 1 parle anglais, tandis que le serveur parle espagnol et le client 2 russe, ils ne se comprendront pas [3]. Cette idée de langue commune s'appelle une *interface* en programmation. Il faut une interface commune à chaque client entre clients et serveurs. Cette interface nous a été fourni par le sujet. Cela permet de faire jouer les clients des différentes équipes dans un tournoi.

2.1 Isolation des responsabilités

Le serveur s'occupe de lancer la partie, de la dérouler, et de déterminer la terminaison de la partie. Pour avoir un bon déroulement de la partie et permettre aux clients de jouer, il doit informer le prochain client à jouer du coup joué par le client précédent. Les clients, eux, doivent *correctement* jouer à la partie.

On remarque que le serveur et les clients ont besoin de garder en mémoire l'évolution d'une partie. Le serveur en a besoin pour savoir si un joueur fait un coup invalide. Les clients en ont besoin pour suivre une stratégie cohérente.

On décide donc de créer un troisième module nommé `common` qui implémentera toutes les fonctionnalités communes aux clients et au serveur.

2.2 Interface entre client et serveur

Les clients doivent disposer de quatres fonctions.

— **get_player_name()** : Retourne le nom du joueur.
— **initialize(plateau)** : Permet au joueur de se préparer pour la partie à venir.
— **play(coup_precedent)** : Indique au joueur quel coup l'autre client a joué et demande au joueur de jouer un coup.
— **finalize()** : Permet au joueur de libérer les ressources allouées.

Cette interface établie une généricité entre tous les clients et permet de les faire s'affronter. Cela rend possible la création d'une compétition pour établir quelle est la meilleur stratégie.

3 Module Serveur

Le serveur s'occupe de gérer une partie jouée par deux clients.

D'une part, il s'occupe de l'isolation de la mémoire lors de l'initialisation. D'autre part, il doit être capable de déterminer le gagnant de la partie.

3. Sauf coup de chance extrême

3.1 Isolation de la mémoire

Il nous a été nécessaire d'isoler la mémoire entre chaque clients. En effet, le C donnant un accès complet à la mémoire, et vu que le serveur et les clients sont dans le même processus, il est extrêmement risqué de donner le même pointeur au plateau à chaque client. Pour éviter cela, le serveur doit copier les données d'initialisation de la partie pour chaque client.

Dans le cas où les clients et le serveur seraient sur des machines séparées, la question ne se poserait pas et les clients seraient contraints d'allouer eux-mêmes la mémoire pour cette tâche.

3.2 Détermination du vainqueur

Au fur et à mesure de la partie, le serveur reçoit les coups des deux joueurs. Il applique les modifications engendrées par ces coups sur un plateau dont seul lui a l'accès. Ce plateau est fourni par le module `common`. Lorsque que le serveur obtient le coup du joueur courrant, il effectue un test à l'aide la fonction **is_move_legal()** de l'interface board.h. S'il reçoit la valeur false, le joueur courrant à perdu, l'autre joueur est donc le vainqueur.

4 Module Common

Le module `common` contient un ensemble de fonctionnalités qui sont utilisées à la fois par le serveur et par les clients.

4.1 Structures de données

Le module `common` dispose de plusieurs structures de données :
- **array_list** : Un tableau dynamique pouvant contenir n'importe quel objet.
- **position_set** : Un ensemble de positions.
- **tree** : Un arbre d'objets quelconque.
- **zobrist_hash** : Permet de *hasher*[4] un plateau de jeu.
- **board** : Une représentation du plateu de jeu.

Ces structures offrent une diversité de complexités dont les clients peuvent se servir pour minimiser leurs temps d'éxecutions.

Les avoir à disposition de tous les clients simplifie le prototypage de nouvelles stratégies. Par exemple, la structure `tree` permet de modéliser un arbre des parties possibles pour les stratégies qui gardent en mémoire une prévision des prochains tours.

4.2 Plateau de jeu

Le plateau de jeu enveloppe le graphe fournit par GSL. Ce-dernier étant dans un format optimisé, on a choisit de ne jamais le modifier car c'est une opération couteuse et complexe. Pour enregistrer les évolutions de la partie, nous avons décidé de stocker les flèches posées sur le plateau comme un tableau de booléens indiquant si la case c est bloquée par une flèche ou non. Pour les reines, nous avons décidé de d'abord stocker la position des reines pour chaque joueur, puis après quelques étapes de *profilage* pour optimiser nos clients, nous avons rajouté un autre moyen d'accéder aux reines sur le plateau.

5 Module Client

Un client doit retourner un coup à jouer lorsque le serveur lui demande. Ce coup doit être valide, si cela est possible.

Tous les clients disposent des fonctions de l'interface imposées. La différence principale se situe au niveau de la fonction **play()**.

4. Un hash consiste à calculer un nombre le plus unique possible d'une structure de données quelconque

5.1 Client Random

Avant de commencer les autres clients plus avancés, il nous a d'abord fallu implémenter un client de base permettant de "comparer". Ce client de base consiste à jouer aléatoirement jusqu'à ce qu'il ne puisse plus jouer aucune reines. Son algorithme (1) est simple et rapide, ce qui permet de rapidement mesurer l'efficacité de nos nouvelles stratégies.

Entrées: *plateau* : un plateau de jeu
début
  $coup \leftarrow \{coup\ invalide\}$;
  si $reines\_non\_bloquees(plateau) = \emptyset$ **alors** **retourner** *coup* ;
  $reine \leftarrow element\_aleatoire(reines\_non\_bloquees(plateau))$;
  $coup \leftarrow element\_aleatoire(tous\_coups\_possibles(reine))$;
  retourner *coup*;
fin

Algorithm 1: Algorithme du joueur aléatoire

5.2 Client Alphabeta pruning

Lorsque deux joueurs jouent à un jeu comme celui des amazones ou même des échecs, les joueurs ont un but distinct. Imaginons une fonction $f(p, j)$ qui donne le score d'un plateau p pour le joueur j. Le joueur rouge tente de maximiser $f(p, rouge)$ tandis que le joueur bleu tente de minimiser $f(p, rouge)$ et vice-versa.

C'est à partir de cette observation que l'algorithme minmax est conçu.

5.2.1 L'algorithme minmax

On peut représenter toutes les parties possibles comme un arbre de jeu.

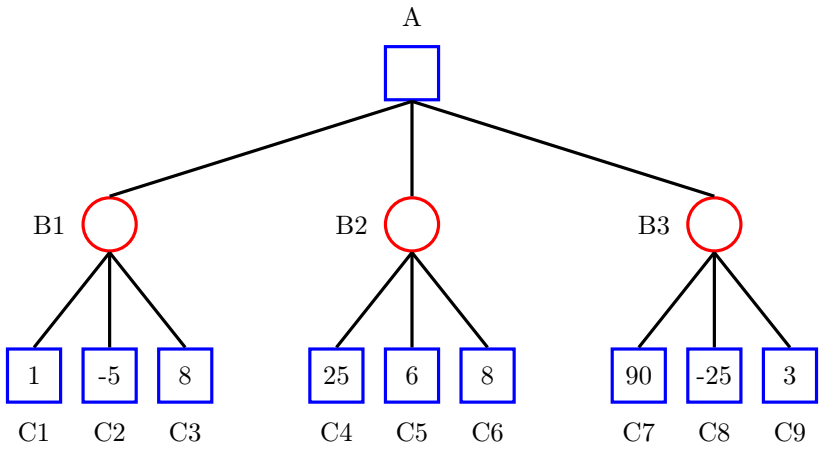


FIGURE 5 – Exemple d'arbre minmax pour un jeu quelconque

La figure 5 montre un exemple de minmax à profondeur 3 sur un jeu où on peut jouer 3 coups différents sur un seul état. Les carrés bleus représentent le joueur bleu, tandis que les cercles rouges représentent le joueur rouge. Les feuilles seront toujours les seuls états évalués. Si on déroule l'algorithme, on obtient la figure 6.

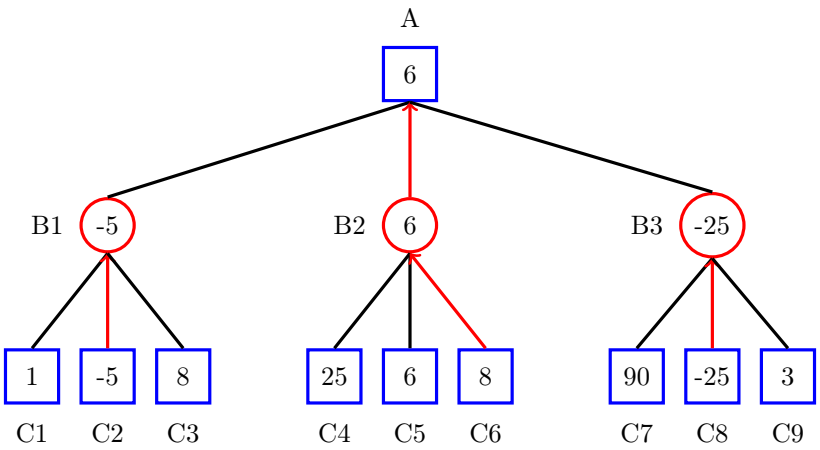


FIGURE 6 – Arbre minmax final après exécution de l'algorithme sur un jeu quelconque

Les noeuds rouges prennent la valeur minimale, tandis que le noeud bleu prend la valeur maximale. D'après minmax, on devrait donc jouer le coup qui donne l'état B2. Cet algorithme est exact. Cela veut dire que si on peut dérouler l'arbre complet des parties, on gagnera à coup sûr [5]. Mais, il est très improbable qu'on arrive à dérouler l'arbre des parties. En effet, pour un simple jeu où l'on ne peut jouer que 3 coups par partie, à profondeur 3 l'algorithme évalue 9 noeuds. Pour un jeu où il y a n coups possibles par état, à une profondeur d, l'algorithme minmax calcule n^{d-1} valeurs. Cela peut très vite exploser, surtout quand le jeu en question est le jeu des amazones avec une cinquantaine de coups possibles selon le plateau.

Cependant, il existe une amélioration de minmax qui évite d'explorer des noeuds inutiles, l'élagage alpha-bêta.

5.2.2 Amélioration de minmax : élagage alpha-beta

L'élagage alpha-beta consiste à ne pas visiter de noeuds inutiles. En effet, cet algorithme profite du fait que le joueur rouge cherche à maximiser tandis que le joueur bleu cherche à minimiser. Reprenons l'exemple en Figure 6. Lorsque l'algorithme évalue B3, on cherche à maximiser, et le maximum courant est égal à 6. Lorsqu'on évalue B3, on évalue d'abord C7 ce qui donne le score de 90, jusque là alphabeta ne fait rien. Or, lorsqu'on évalue C8, la valeur est égale à -25. Cela veut-dire que la valeur de B3 est au moins inférieure ou égale à -25. Or, le joueur bleu cherche à maximiser et possède un maximum courant égal à 6. On sait donc que la valeur de A est au moins supérieure ou égale à 6. On peut en conclure que le noeud B3 sera de toute façon moins bon que le noeud B2. On peut donc arrêter de l'explorer. Cet algorithme permet d'éviter d'explorer tous les noeuds. Malheureusement, la complexité reste toujours exponentielle, mais elle reste meilleure que celle de minmax.

Théoriquement la complexité ne causerait pas de problèmes si nous avions des ressources illimitées et du temps illimité. Or, nous n'avons pas ce luxe. Sur le ladder [6], chaque joueur doit jouer tous ses coups en moins de 10 secondes. Si on voulait passer largement en dessous, il faudrait utiliser l'algorithme à profondeur 1. Or, cela n'est pas intéressant. On peut tout de même prendre avantage du fait que l'arbre réduit massivement en taille plus la partie avance. En effet, à chaque tour, une case en plus devient inaccessible. C'est pour cela qu'on peut tout simplement choisir d'aller de plus en plus loin dans l'arbre des possibilités au fur et à mesure de la partie.

5.2.3 Profondeur dynamique

Comme énoncé précédemment, on prend avantage du fait que le jeu se "simplifie" au cours du temps. On peut même en déduire que la partie devient exponentiellement plus simple. Suite à beaucoup d'expérimentations, nous avons déduit 2 méthodes principales pour déterminer la profondeur à jouer. Une basée sur le temps passé à calculer un arbre alphabeta, et une basée sur le nombre de tours.

5. Sauf si c'est un jeu non déterministe bien entendu
6. La plateforme où les différents clients des différentes équipes s'affrontent

Fonction de profondeur Comme fonction, nous avons décidé d'utiliser la fonction décrite dans l'équation 1 où la fonction $depth(t)$ retourne la profondeur voulue en fonction du nombre de tours écoulés t et de la largeur du plateau m.

$$depth(t) = \lfloor \exp\left(\frac{1.15 * t}{\exp(0.05 * m) * \sqrt{m} * m}\right) \rfloor \tag{1}$$

Les paramètres ont été choisi relativement arbitrairement, mais nous avons tout de même utilisé l'application Desmos [7] qui nous a permis de mesurer l'évolution de notre profondeur en fonction du nombre de tours. De plus, nous avons implémenté le système de *timeout* sur notre serveur pour être sûr que notre algorithme ne dépasse pas le temps imparti.

Mesure du temps Ensuite, pour la deuxième façon d'augmenter la profondeur, le programme calcule le temps maximal pour 1 tour ($\frac{timeout}{nombre\_cases}$), mesure le temps pris pour une exécution de l'algorithme alphabeta, et si l'inégalité définie dans l'équation 2 est vraie, on peut augmenter la profondeur de 1.

$$\frac{1}{\sqrt{nombre\_cases}} * temps\_pris < \frac{max\_temps\_par\_tour}{nombre\_sommets} \tag{2}$$

De plus, si le temps pris par l'algorithme alphabeta est supérieur au temps maximal par tour, on réduit la profondeur de 1.

La deuxième approche est relativement meilleure que la première approche, mais la deuxième approche a plus de chance de résulter en un *timeout*. L'idéal, ce serait de calculer exactement la complexité de notre alphabeta en fonction de l'avancement de la partie, et si cette complexité, même à profondeur supérieure, est en dessous d'un certain seuil, on peut aller plus profondément.

Enfin, aller profondément, c'est efficace. Mais explorer plus profondément ne signifie rien si notre heuristique n'est pas bonne.

5.2.4 Heuristiques et théorie des jeux

Le jeu des Amazones est un jeu relativement récent [8], donc il n'y a pas beaucoup de recherche à propos de la théorie derrière ce jeu. De plus, le nombre d'ouvertures est bien plus grande que celles aux échecs. Cependant, les premiers coups du jeu ne sont pas si intéressants que ça. On peut séparer le jeu des amazones en 3 parties distinctes

- L'ouverture. C'est le début de la partie, les coups ont peu d'importance vu le domaine jouable.
- Le remplissage. C'est de loin la partie la plus importante du jeu. C'est dans cette partie que le reste de la partie va se décider.
- La fin de partie. Normalement, toutes les reines sont isolées. Le joueur qui a le mieux isolé l'adversaire l'emporte.

On remarque donc que la partie cruciale est le remplissage. Durant cette partie, on cherche à bloquer le plus les reines adverses tout en essayant de ne pas se faire isoler. La dessus, plusieurs heuristiques peuvent représenter ces situations.

Mouvements possibles Un bon plateau pour un joueur, c'est celui où l'ennemi peut effectuer le moins de mouvements et où on peut jouer le plus de mouvements. Dans cette heuristique, on mesure la mobilité de chaque joueur. Le joueur avec le plus de mobilité est celui avec le plus gros avantage. L'équation 3 décrit la fonction heuristique. On voit même qu'on pourrait modifier les poids de chaque mobilité, pour définir la priorité sur bloquer l'ennemi, ou éviter de se faire bloquer. Cet heuristique est bonne, car à un niveau superficiel, on cherche effectivement à limiter la mobilité du joueur adverse tout en évitant de se bloquer. Rien qu'avec cette heuristique et toutes les optimisations précédentes, notre joueur alphabeta gagne à 100% contre le joueur aléatoire [9]. Cependant, cette heuristique n'est pas suffisante. En réalité, le jeu des amazones n'est pas une question de mobilité, mais de *territoire*.

$$evaluate(p, joueur) = mobilite(p, joueur) - mobilite(p, ennemi) \tag{3}$$

7. Une calculatrice graphique
8. Ca dépend si 1988 est considéré comme récent...
9. Bien heureusement, sinon cela signifirait soit que notre heuristique n'est pas bonne, soit qu'alphabeta ne peut pas aller assez profondément pour jouer pertinemment

Territoire Comme deuxième heuristique, nous avons optés pour l'heuristique du territoire. Cette heuristique consiste à calculer une table de distances pour chaque joueur. Un exemple de table des distances est montré dans la figure 7. Un territoire est défini par l'ensemble des cases sur lesquelles on peut arriver plus rapidement que l'adversaire. Cette heuristique est déjà plus intelligente, car elle inclut le concept de mobilité mais rajoute aussi une autre notion, celle d'espace. L'alphabeta muni de cette heuristique bat 100% du temps notre ancienne heuristique [10].

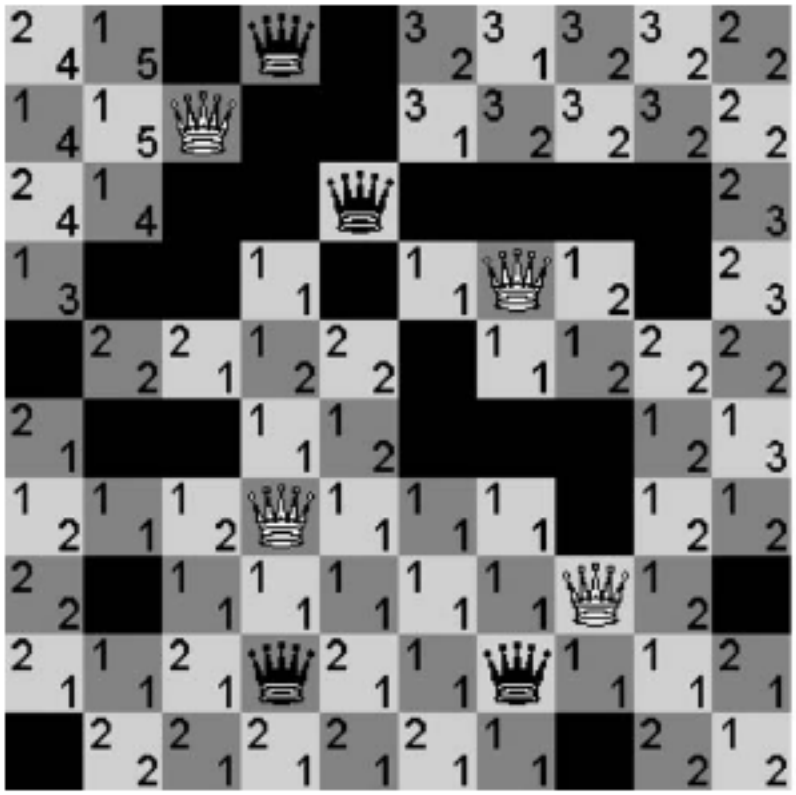


FIGURE 7 – Exemple de table des distances (Fig. 2 source)

5.2.5 Optimisations supplémentaires

On pourrait effectuer d'autres optimisations. Par exemple, on pourrait utiliser une table de transposition, celle-ci s'occuperait de stocker l'évaluation connue pour un plateau. Cela permettrait d'implémenter de l'*Iterative Deepening* de façon optimisée. L'*Iterative Deepening* consiste à commencer à profondeur 1, et tant qu'il nous reste du temps, on passe à la profondeur suivante jusqu'à qu'on le temps soit écoulé. Cependant, l'*Iterative Deepening* sans table de transposition revient à recalculer l'arbre à profondeur 1, ce qui est juste stupide. On pourrait commencer l'exploration à partir du meilleur chemin trouvé. Au départ, nous avions comme intention d'implémenter une table de transposition, mais par manque de temps nous avons dû abandonner.

De plus, nos heuristiques ne sont pas forcément les meilleures. Nous pourrions utiliser un Réseau de Neurones Convolutionnel (CNN) couplé à un algorithme d'apprentissage par renforcement pour évaluer un plateau. Cependant, cela poserait des problèmes dans nos cas où nous pouvons avoir des plateaux alambiqués. De plus, le CNN prends beaucoup de mémoire, et risque de tourner très mal sur un CPU.

Enfin, nous pourrions aussi tenter une parallélisation de l'algorithme, même si nous perdrons l'avantage de l'algorithme alphabeta...

10. Si ce n'était pas le cas, cela contredirait tout ce que j'ai dit précédemment, donc heureusement que c'est meilleur en pratique

Malgré toutes ces propositions, l'élagage alphabeta reste un algorithme relativement peu efficace dans notre cas. En effet, le jeu des Amazones possède tellement de combinaisons, que l'algorithme ne peut pas explorer très profondément. Il y a un autre algorithme qui vise à palier ce problème, le Monte Carlo Tree Search (MCTS).

5.3 Client Monte Carlo Tree Search (MCTS)

L'algorithme Monte Carlo Tree Search défini en 2 est un algorithme d'apprentissage par renforcement. Il stocke en mémoire le *modèle* du jeu sous forme d'arbre, et joue en fonction de ce modèle. Dans le modèle de jeu, chaque noeud représente un état. La racine est l'état courant, et les enfants de la racines sont les prochains états. L'algorithme stocke 2 informations essentielles par noeud : N le nombre de visites du noeud, et W le nombre de victoires du joueur (et non de l'ennemi) en partant de ce noeud.

Il y a deux cas d'utilisations de l'algorithme : entraînement et exploitation. Pour l'entraînement, on sélectionne une feuille de notre modèle de jeu en suivant une stratégie spécifique. Ensuite, on regarde tous les enfants possibles depuis cet état et on en choisit un. On déroule la partie jusqu'à arriver à un état *terminal*. Le déroulement de la partie doit être le plus rapide possible. Dans le cas de notre projet, nous avons décidé de choisir un déroulement aléatoire, même si nous aurions pû choisir une manière plus intelligente de jouer [11] qui permettrait à l'algorithme de converger plus rapidement à une solution efficace. Un exemple du déroulement d'une itération de l'algorithme est représenté dans la Figure 8.

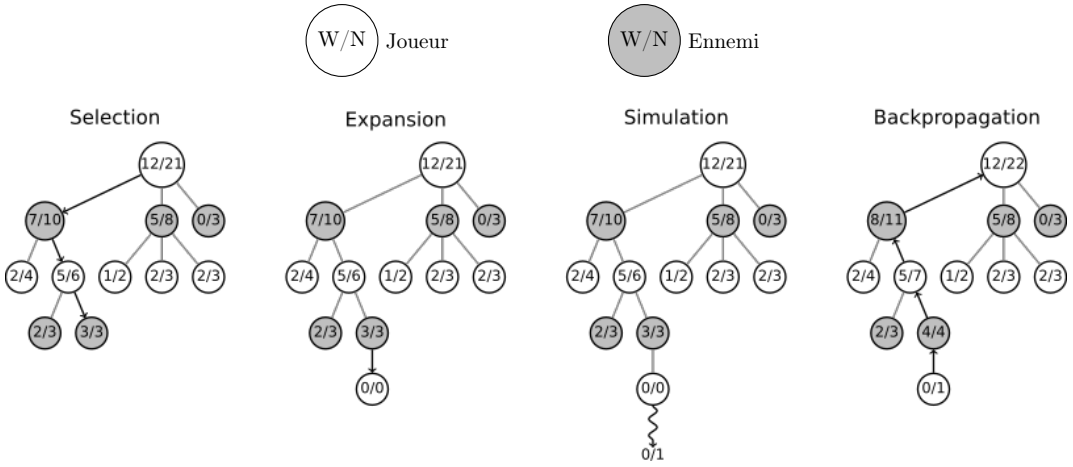


FIGURE 8 – Exemple d'itération de l'algorithme MCTS (Wikimedia)

Même si cet algorithme converge vers minmax [12], il nécessite un nombre d'itérations conséquent. De plus, il faut bien choisir notre heuristique grâce à laquelle on choisit notre noeud à l'étape de sélection. Enfin, cet algorithme est en théorie meilleur que alphabeta, mais nous n'avons pas réussi à obtenir de résultats concluants. Peut-être cela est-ce dû à une complexité de nos algorithmes trop élevée. Ou dû au fait que durant l'expansion, nous simulons tous les enfants.

Une amélioration possible serait d'évaluer les enfants du noeud sélectionné de façon parallèle. En effet, les zones mémoires sont séparées, nous pouvons donc plutôt facilement paralléliser le problème. Même après moultes profilages et optimisations, l'algorithme du MCTS n'a malheureuseent pas été concluant, malgré toutes les promesses à son propos...

6 Optimisations

Durant le développement de nos différents clients, nous nous sommes rendus compte que nous étions très limité par le temps de calcul. En effet, la plupart de nos algorithmes n'étaient pas optimisés. Nous avons donc eu recourt à du profilage

11. Tel que jouer le coup qui maximise le nombre de coups possibles
12. Bouzy, Bruno. "Old-fashioned Computer Go vs Monte-Carlo Go". IEEE Symposium on Computational Intelligence and Games, April 1–5, 2007, Hilton Hawaiian Village, Honolulu, Hawaii.

Entrées: p : plateau, A : arbre de jeu
début
  $to\_expand \leftarrow selection(A)$;
  $expansion = expand(to\_expand)$;
  $result = simulate(expandison)$;
  $backpropagate(result, A, expansion)$;
fin

Algorithm 2: Algorithme Monte Carlo Tree Search

6.1 Profilage et identification de zone critiques

Après un premier appel au profileur *gprof* montré dans la Figure 9 , nous voyons déjà une première zone critique, `find_neighbor_in_direction`.

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 63.70      1.72     1.72 22147930     0.00     0.00  find_neighbor_in_direct
 16.30      2.16     0.44 18578520     0.00     0.00  is_cell_empty
 11.11      2.46     0.30                             _init
  6.67      2.64     0.18  1646152     0.00     0.00  get_reachable_positions_
  0.74      2.66     0.02 18578520     0.00     0.00  is_on_board
  0.74      2.68     0.02     4525     0.00     0.00  rollout
```

FIGURE 9 – Résultat du profileur *gprof* après exécution d'une itération MCTS

On remarque que la fonction `find_neighbor_in_direction` nous mange toute la puissance de calcul. Cette fonction est appelée plus de 22 millions de fois, elle occupe 63.70% du temps CPU et la somme de la durée prise par tous ses appels est égale à 1.72 secondes, ce qui est gigantesque pour une seule itération de MCTS. Au moment du *profiling*, la fonction `find_neighbor_in_direction` avait le fonctionnement défini dans l'algorithme 3. Sa complexité était au moins de $O(nombre\_sommets)$ soit $O(n)$. Nous voudrions une complexité $O(1)$ idéalement vu le nombre d'appels à cette fonction.

Entrées: M : matrice d'adjacence, d : direction, i : $\mathbb{N} < nbre\_sommets$
début
  pour tous les $0 \leq j < nbre\_sommets$ **faire**
    si $M_{i,j} = dir$ **alors retourner** j ;
  fin
  retourner *-1*;
fin

Algorithm 3: Algorithme peu efficace pour trouver le voisin dans une direction

2 idées ont été implémentées pour améliorer cette méthode. Premièrement, on stocke le résultat de l'algorithme dans un cache, donnant une complexité moyenne $O(1)$. Deuxièmement, la première approche était très naïve, elle ne prenait pas en compte l'avantage du format CSR. En effet, avec le format CSR, on peut tout simplement itérer sur les valeurs non zéros à la ligne i, ce qui équivaut à itérer sur tous les voisins de i. Cette deuxième approche est de complexité $O(NUM\_DIRS)$, donc est aussi $O(1)$. En fusionnant les deux approches, on peut obtenir l'algorithme 4 qui est très efficace et prends avantage de nos structures de données.

Après application de ces résultats et un appel à *gprof* affiché dans la Figure 10, une nouvelle zone critique apparait : *is_cell_empty*.

Entrées: M : matrice d'adjacence, d : direction, i : $\mathbb{N} < nbre\_sommets$
début
  si *is_neighbor_cached(dir, i)* **alors retourner** *cached_neighbor(dir, i)* ;
  pour chaque $j \in voisins(i)$ **faire**
    si $M_{i,j} = dir$ **alors retourner** j ;
  fin
  retourner *-1*;
fin

Algorithm 4: Algorithme peu efficace pour trouver le voisin dans une direction

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 60.61      0.40     0.40 18578520     0.00     0.00  is_cell_empty
 15.15      0.50     0.10  1646152     0.00     0.00  get_reachable_positions_generic
  9.09      0.56     0.06 22147930     0.00     0.00  find_neighbor_in_direction
  3.03      0.58     0.02 22147930     0.00     0.00  find_neighbor_in_cache
  3.03      0.60     0.02 18578520     0.00     0.00  is_on_board
```

FIGURE 10 – Résultat du profileur *gprof* après exécution d'une itération MCTS et optimisation *find_neighbor_in_direction*

Cette fois-ci, c'est *is_cell_empty* qui occupe toutes les ressources CPU. La raison est simple, nous stockons actuellement les positions des reines par joueur. Pour savoir si une case est vide, il faut donc itérer sur toutes les reines de tous les joueurs pour savoir s'il y a une reine, et regarder s'il y a une flèche à cette position. Là encore, nous avons un vilain $O(n)$ couplé à un nombre d'appels colossal. Cependant, la solution à ce problème n'est pas aussi élégante que celle pour *find_neighbor_in_direction*. En effet, nous devons dupliquer de l'information, car stocker les reines par joueur est intéressant pour d'autres algorithmes. Nous avons donc décidé d'avoir une redondance de l'information, avec le champ `queens` pour avoir les positions des reines par joueur, et le champ `queens_on_board` qui indique quelle reine est à une position donnée en temps constant. Malheureusement, cela rajoute une complexité algorithmique aux fonctions `apply_move` et `cancel_move` qui applique (resp. annule) un coup sur un plateau. Après optimisations et après changement de gprof à oprofile [13], nous obtenons la figure 11, qui montre que nous avons encore quelques fonctions critiques, mais malheureusement nous n'avons pas trouvé d'optimisations pertinentes à réaliser sur ces fonctions.

```
36761    26.0592  (no location information)   alphabeta.so   get_reachable_positions_generic
26449    18.7492  (no location information)   alphabeta.so   is_cell_empty
25191    17.8575  (no location information)   alphabeta.so   find_neighbor_in_direction
12687     8.9936  (no location information)   alphabeta.so   add_position
12488     8.8525  (no location information)   alphabeta.so   find_neighbor_in_cache
9906      7.0222  (no location information)   alphabeta.so   is_on_board
```

FIGURE 11 – Bilan oprofile d'une exécution d'une partie en utilisant la stratégie *Function Deepening* de notre alphabeta

7 Conclusion

Pour conclure, nous dirions que le choix de simuler un serveur par un programme C qui charge d'autres librairies dynamiques est relativement étrange. En effet, charger les clients comme des librairies dynamiques ouvre la porte à beaucoup de failles, vu que les clients partagent le même espace mémoire que le serveur. Certaines stratégies malicieuses consisteraient à détecter si nous jouons contre un autre de nos clients, et si cela n'est pas le cas, on déclenche un crash chez l'autre client, le pénalisant de 10 points sur le ladder. Une faille imparable serait tout simplement de réserver toute la mémoire disponible, pour empêcher tout appel de *malloc* par l'adversaire. Cela aurait de très grandes chances de résulter en un `segmentation fault (core dumped)`.

13. gprof n'arrive pas à profile les librairies dynamiques, ce qui rendait le profiling pénible à effectuer. Nous avons donc opté pour oprofile qui permettait cela.

On aurait pu créer un processus par client et serveur, où le serveur et les clients communiquent avec un système de *pipes* ou de *socket*. Cependant, cela explose la complexité du projet sur des points peu intéressants comme la transmission de données entre clients et serveurs, ouvrant la porte à multiples bugs obscurs.