ENSEIRB-MATMECA

Projet du semestre 5
Informatique

Quoridor

Auteurs :
Jean-Baptiste Zacchello
Florian Le Vern
Amine Cheknoun

Responsable :
M. Corentin Travers

Decembre 2014

Sommaire

Introduction

Le Quoridor est un jeu de société dans lequel s'affrontent deux joueurs sur un plateau de 81 tuiles carrées (9×9). Chaque joueur y est représenté par un pion qui part de la tuile centrale du côté du plateau situé face à lui. L'objectif est d'être le premier joueur à atteindre n'importe laquelle des neuf tuiles situées sur le bord opposé du plateau. Chaque joueur dispose de 10 murs de la longueur de 2 tuiles. Chacun à leur tour, ils doivent choisir entre bouger son pion dans une direction donnée ou placer un mur pour allonger le chemin de son adversaire.

L'objectif du projet est d'implémenté en langage C le jeu du Quoridor, des intelligences artificielles pour ce jeu et un serveur pour que les IA puissent s'affronter et vérifier qu'elles respectent les règles.

Par ordre chronologique, nous nous sommes d'abord intéressés à la modélisation du jeu et à l'implémentation de fonctions communes à tous les groupes. Il a fallu ensuite créer l'interface graphique pour pouvoir faire les tests de ces fonctions. L'implémentation du serveur s'est faite en même temps que le développement des premières IA pour pouvoir les tester. Nous verrons à la fin de ce rapport quelques axes d'améliorations que nous avons remarqué.

1 Modélisation du jeu

1.1 Le plateau et les coups

Pour la représentation de la grille, notre choix s'est porté sur un tableau à deux dimensions de taille 17×17. Les cases ayant l'ordonnée et l'abscisse paires peuvent recevoir un pion, les autres peuvent recevoir un mur. Nous avons fait ce choix pour sa simplicité d'utilisation (pour placer les murs) et par sa ressemblance visuelle avec le plateau de jeu.

Les coups sont représentés par un triplet de coordonnées (c, l, d), avec $0 \leq c \leq 8$ le numéro de la colonne et $0 \leq l \leq 8$ le numéro de la ligne. $d = -1$ si le coup est un déplacement, $d = 0$ si c'est un placement de mur horizontal, $d = 1$ si c'est un placement de mur vertical. Cette facon de représenter un coup s'est imposée à nous pour correspondre aux normes régissant les fonctions imposées de l'interface.

1.2 Les fonctions de l'interface

Pour que d'autres groupes puissent proposer leurs IA sur notre interface, des fonctions communes reposant sur des normes précises ont été imposées. Au fur et à mesure de l'avancée du projet, nous avons discuté de ces fonctions avec d'autres groupes ce qui a conduit à l'ajout ou à la modification de certaines fonctions. En effet, pour implémenter les IA, des fonctions donnant des informations sur l'état du jeu étaient nécessaires.

1.2.1 Discussion sur les "const struct board*"

De notre côté, nous avons choisi de supprimer la restriction "const" sur le plateau de jeu en entrée des fonctions "move_pawn", "is_blockable" et "place_wall".

Voici l'explication de ce choix :

Ces trois fonctions nécessitent de modifier l'état du jeu. Deux possibilités se présentaient. La première était de conserver ces "const struct board*" en entrées de ces fonctions et par conséquent de modifier, directement dans celles-ci, un "struct board" global. Ainsi, c'est seulement en utilisant ces fonctions que l'on peut modifier l'état du jeu puisqu'elles seules connaissent le nom du "struct board" global. L'IA ne pourra donc pas tricher à moins de connaitre ce nom. Néanmoins, cette facon de faire implique que l'IA est obligée d'avoir une représentation personnelle du jeu pour pouvoir tester ses coups et choisir le meilleur par exemple, sinon l'utilisation d'une fonction de l'interface sur une copie du plateau modifiera quand même le vrai plateau.

C'est pour cette raison que nous avons choisi la deuxième possibilité. Elle consiste à supprimer la restriction des "const struct board" en entrée. Ainsi l'IA peut effectuer des tests directement avec les fonctions de l'interface. Notre gestion de la sécurité se fait dans le serveur. Les IA des tous les autres groupes pourront donc jouer avec notre interface. L'inconvénient est que nous avons implémenter nos IA en suivant ce principe, donc les groupes qui auront choisi la première option ne pourront pas faire fonctionner nos IA avec leur interface.

1.2.2 Explication de l'implémentation

La plupart des fonctions de l'interface se font en temps constant par des accès directes au plateau de jeu "struct board* b". Intéressons-nous plus particulièrement aux fonctions qui

modifient le plateau de jeu.

void **place_wall**(struct board* b, char column, char line, char direction)
La norme choisie pour placer les murs est la suivante : à partir d'une position donnée, si on demande un mur horizotal, il sera placé au dessus et continuera vers la droite, si on demande un mur vertical, il sera placé à gauche et continuera vers le bas.

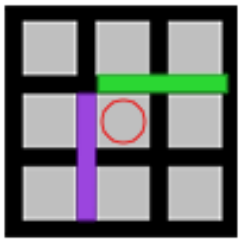

FIGURE 1 – Placement des murs à partir d'une position donnée (rond rouge)

La procédure place_wall modifie notre tableau 17×17. C'est à ce niveau que l'utilisation d'une telle méthode se révèle très pratique puisqu'il suffit d'écrire des 1 aux emplacements représentant les "goutières" pour les murs, si le coup est bien validé.

int **is_blockable**(struct board* b, char column, char line, char direction)
Cette fonction indique s'il est possible de placer un mur à une position donnée, dans une direction donnée. Il y a principalement deux vérifications à faire :

- Le nouveau mur ne doit pas couper un mur déjà posé
- Le nouveau mur ne doit empêcher aucun des joueurs de finir. Pour vérifier cette condition, on place d'abord le mur demandé pour tester. Puis on utilise un algorithme récursif qui, partant de la position d'un joueur remplit un tableau 9×9 de 0 avec des 1 à chaque case qu'il peut atteindre. Si la ligne gagnante est atteinte alors il existe bien un chemin pour la victoire. On oubliera pas de supprimer le mur qui a servi pour le test.

void **move_pawn**(struct board* b, char column, char line)
Pour le déplacement des pions, ceux-là ne sont pas représentés dans le tableau 17×17. La structure board contient deux structures player. Une structure player contient la position d'un pion ainsi que le nombre de mur qu'il reste à ce joueur. Avoir fait ce choix, nous permet d'accéder à la position d'un joueur en temps constant plutôt que de devoir trouver sa position dans le tableau ce qui aurait conduit à une complexité linéaire.

1.3 La vérification de la validité des coups

Un coup est vérifié quand il répond à toutes les règles du jeu.

Pour un placement d'un mur :

- vérifier si l'emplacement choisi est innoccupé
- si l'adversaire a toujours un chemin pour gagner.

Pour un déplacement, il faudra comparer la position actuelle du pion du joueur et celle désirée. Un déplacement "classique" implique un déplacement d'une case. Un déplacement "saut" est un déplacement de 2 cases : alignées ou en diagonale.
Il faudra alors tester 3 cas différents :

1. si le déplacement est "classique" :
   — vérifier si il n'y a pas de mur entre le pion et l'emplacement voulu
2. si le déplacement est un "saut" aligné :
   — vérifier si il y a un pion adverse à coté du pion du joueur, dans le sens du saut
   — vérifier si il n'y a pas de mur entre le pion du joueur et le pion adverse
   — vérifier si il n'y a pas de mur entre le pion adverse et l'emplacement voulu
3. si le déplacement est un "saut" diagonale :
   — vérifier si il y a un pion adverse à coté du pion du joueur
   — vérifier que le pion adverse soit adjacent à l'emplacement voulu
   — vérifier si il n'y a pas de mur entre le pion du joueur et le pion adverse
   — vérifier si il n'y a pas de mur entre le pion adverse et l'emplacement voulu
   — vérifier si il y a un mur derrière le pion adverse par rapport au pion du joueur

2 L'interface graphique

L'interface graphique a été réalisée à l'aide de la bibliothèque *Ncurses*. Nous avons essayé de représenter le plateau de jeu de la manière la plus lisible qu'il soit.

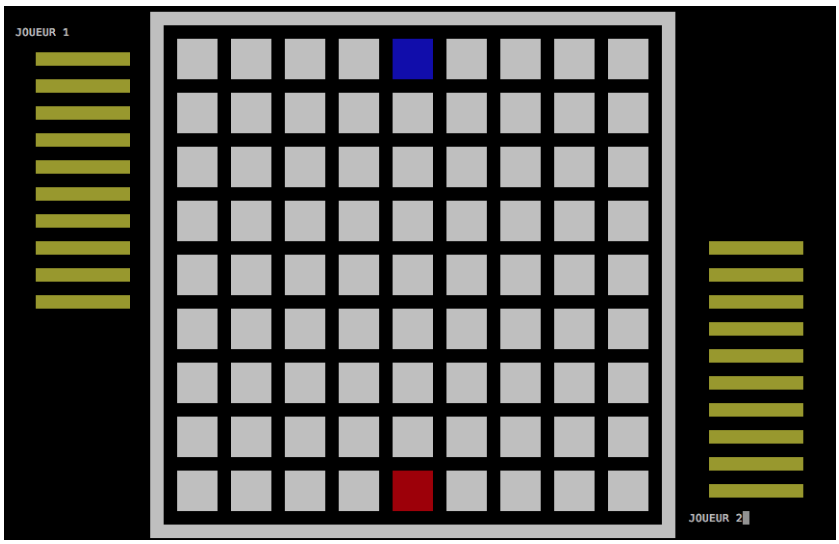

FIGURE 2 – Interface graphique

Le fait d'avoir un tableau de 17×17 a été un avantage pour la représentation de celui-ci : il suffit de lire les valeurs de chaque case pour savoir ce qu'il faut afficher. Nous avons donc créé des sous-fenêtres carrées pour représenter les cases jouables (tuiles). Ensuite, pour représenter les murs qui s'étendent sur deux cases, il était nécessaire de créer des barres verticales, horizontales ainsi que des carrés qui servent à lier ces barres. C'était pour nous la manière la plus simple de représenter l'état du jeu.

La représentation d'une barre se fait donc à l'aide de trois sous-fenêtres. Deux rectangles de même longueur que la taille d'une tuile ainsi qu'un carré de côté l'espacement entre deux tuiles. Ainsi on forme une barre qui s'entend sur deux tuiles.

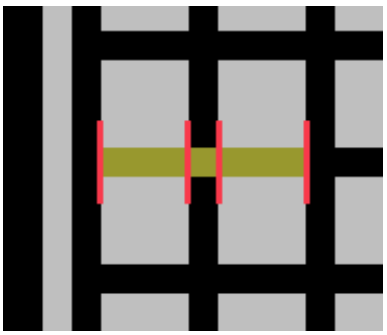

FIGURE 3 – Représentation des murs

On a ensuite affiché de nombre de murs restants pour chaque joueur. Là encore, il suffisait simplement d'affiche des rectangles à un intervalles réguliers à l'aide d'une boucle itérant jusqu'à remaining_bridges(b, player).

3 Le serveur

Le serveur permet aux intelligences artificielles de pouvoir s'affronter et vérifie que les coups sont conformes.

Notre serveur fait intervenir les IA par le biais du fichier "registry.h" fourni. Pour assurer qu'une IA ne triche pas, une copie du plateau de jeu (variable "struct board" globale) lui est transmise à chaque tour. Ainsi elle peut modifier à sa guise cette copie et faire tous les tests qu'elle veut même à partir de nos fonctions (ce n'est pas un "const"). A la fin du traitement de ce plateau de jeu par l'IA, le serveur compare celui-là avec le véritable plateau global. Il récupère ainsi le coup joué. Cela permet donc de s'assurer qu'un seul coup à été joué. Puis il rejoue ce coup sur le véritable plateau de jeu. En rejouant le coup, il utilise les fonctions de notre interface qui s'assurent de la validité de celui-là.

Bonus : l'historique.
Le fait de deviner le coup joué par l'IA et de le rejouer sur le vrai plateau permet d'archiver la partie dans un fichier texte. Cet historique ne sert pas de décoration : grâce à un main de tests, on peut lire ce fichier et donc revisualiser une partie. Cela permet de voir où sont les problèmes dans nos IA ou de pouvoir les améliorer lorsqu'il y a eu un cas intéressant dans une partie.

4 Les intelligences artificielles

4.1 Le PlayerBot

Avoir un joli plateau, c'est bien, mais pouvoir jouer dessus c'est mieux ! Ce bot n'en est en fait pas un puisqu'il permet à l'utilisateur de pouvoir jouer.

4.2 Le RandomBot

Pour commencer à implémenter des IA, il est bon de s'assurer que tout le reste fonctionne déjà bien. Le RandomBot est là pour cela. Il choisit aléatoirement s'il va jouer un mur ou se déplacer. Puis il choisit aléatoirement l'endroit où il va effectuer son action parmi tous les coups autorisés. On se rendra alors vite compte que tous les tests de validité d'un coup étaient mal cilbrés au début ...

4.3 Le MiroirBot

Cette IA n'a aucune chance de gagner ! Il serait tout de même étrange de perdre contre quelqu'un qui joue exactement le même coup que vous. Néanmoins, le défis est intéressant puisqu'il faut deviner où a joué son adversaire sans connaître l'implémentation de l'interface de jeu.

Ainsi, le MiroirBot conserve dans une variable "struct board" globale l'état du plateau de jeu à son tour précédent et le compare à celui qui lui est donné. Grâce aux fonctions communes de l'interface, on peut facilement savoir si le joueur a bougé son pion ou a placé un mur. En parcourant les tuiles aux alentours du pion adverse on peut aussi savoir comment il a bouger son pion. Par contre, pour trouver où il a placé un mur cela devient plus corsé. La solution proposée est la suivante :

```
Pour chaque position de mur horizontal
de la gauche vers la droite et de haut en bas Faire
   Si on peut placer le mur à l'état précédent
   et qu'on ne peut plus le placer maintenant Alors
      On retient ce coup
      (on a coupé le mur qui vient d'être posé)
   Fin Si
Fin Pour

De même pour les murs verticaux
```

Ces deux informations nous permettent de trouver la tuile où le mur a été joué et son sens.

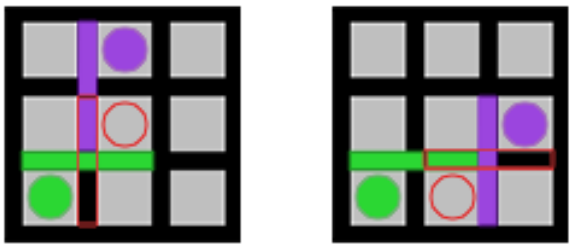

FIGURE 4 – Trouver la position du mur posé par le joueur adverse (rond rouge) et son sens dans MiroirBot

NB : on notera qu'il y a quelques cas particuliers au bords du plateau.

FIGURE 5 – Cas particuliers dans MiroirBot

La complexité de cet algorithme est donc de l'ordre du nombre de murs plaçables + la taille du tableau (9×9). Mais il ne semble pas possible de pouvoir faire mieux.

Le plateau étant totalement symétrique, il suffit de faire une rotation centrale du coup joué par l'adversaire pour jouer son coup. Cependant, il faudra ajuster cette position (on rappelle que les murs horizontaux sont placés au dessus des tuiles et vont vers la droite et les murs verticaux sont placés à gauche et vont vers le bas).

Pas besoin de s'occuper de la validité des coups à jouer puisque son adversaire l'aura fait pour lui ! cela n'est pas totalement vrai car un adversaire un peu futé pourra forcer MiroirBot à tricher en le faisant poser un mur à un endroit bloquant le jeu ou tout simplement en sautant par dessus son pion.

4.4 Le ShortestPathBot

Cette IA est tournée autour d'un algorithme du plus court chemin.
Si son plus court chemin est inférieur ou égal à celui adverse, le pion se déplace en suivant ce chemin. Sinon, il cherche à placer un mur tel que la différence entre le plus court chemin

adverse et son plus court chemin soit maximale.

Liste des algorithmes :

1. Fonction : PlusCourtChemin(Plateau, Tableau[9][9], Entier JoueurActuel) : Entier
2. Procédure : PlusCourtCheminRec(Plateau, Tableau[9][9] d'entiers, PositionActuelle)
3. Fonction : RemonterChemin(Plateau, Tableau[9][9] d'entiers, PositionActuelle) : Position
4. Procédure : Bouge(Plateau, Tableau[9][9] d'entiers, Entier JoueurActuel, Entier Distance)
5. Procédure : Main(Plateau)

81	3	2	3	4
81	2	1	2	5
2	1	0	1	4
3	2	1	2	3
3	81	2	3	4
4	3	3	4	5

FIGURE 6 – Algorithme Shortpath

PlusCourtChemin(Plateau, Tableau[9][9], Entier JoueurActuel) : Entier
L'algorithme du plus court chemin, calcule à partir de la position d'un pion, la distance en terme de tours pour qu'il puisse atteindre toutes les tuiles disponibles du plateau et renvoie un tableau contenant, pour chaque tuile, la distance minimale entre le pion du joueur actuel et celle-là.

Il initialise un tableau de taille 9×9.
Si la case contient le pion du JoueurActuel, elle prends la valeur 0, sinon 81. (La valeur 81 est la distance correspondant parcourir toutes les cases du tableau).
Puis lance la procédure auxiliaire PlusCourtCheminRec chargée de modifier les valeurs du tableau.
Enfin, il renvoie la plus petite valeur des tuiles de la ligne de victoire du JoueurActuel.

La complexité en espace de cette fonction est Θ(n) avec n=la taille du tableau car elle appelle la fonction PlusCourtCheminRec.
La complexité en temps est $\Theta(n\sqrt{n})$ car elle appelle la fonction PlusCourtCheminRec.

PlusCourtCheminRec(Plateau, Tableau[9][9] d'entiers, Position Actuelle)

```
Pour les 4 directions : Vérifier si il y a un mur
   Si oui, passer à la direction suivante
```

```
Si non, vérifier qu'il y a un pion (ce sera forcément celui du joueur adverse).
   Si oui, pour chaque déplacement de "saut" autorisé : faire
      Si la valeur de la case désirée est supérieur stricte à celle actuelle
      +1
         Si oui, cette case prend la valeur : celle actuelle +1 et relancer
         l'algorithme avec la position de cette nouvelle case.
   Si non, vérifier si la valeur de cette case est supérieur stricte à
   celle actuelle -1
      Si oui, cette case prend la valeur : celle actuelle +1 et relancer
      l'algorithme avec la position de cette nouvelle case
```

L'algorithme s'arrête une fois que toutes les plus courtes distances ont été trouvées.
Il restera des cases contenant 81 : celle du pion adverse et celles inatteignables.

La complexité en espace de cette fonction est Θ (n) avec n=la taille du tableau.
En effet, la pile maximale d'appel récursif est celle passant par toutes les cases du tableau.
La complexité en temps est $\Theta(n\sqrt{n})$.
Cette complexité a été trouvé en incrémantant un compteur pour chaque appel récursif.

Ce tableau donne toutes les informations pour le fonctionnement de l'IA :

1. La taille du plus court chemin est la valeur la plus petite parmi les cases de la ligne gagnante.
2. En partant de la case gagnante du plus court chemin, on peut déterminer sur quelle case le pion doit se déplacer.
3. Pour tester si un mur ne bloque pas le joueur adverse, si toutes ses cases gagnantes ont la valeur 81, alors ce mur n'est pas autorisé.

Toutes les autres fonctions utilisent ce tableau.

RemonterChemin(Plateau, Tableau[9][9] d'entiers, Position Actuelle) : Position
L'algorithme pour remonter le plus court chemin prends en entrée le tableau de l'algorithme précédent et le joueur actuel, et renvoie la position de la case précédente du chemin.

```
Pour les 4 directions : Vérifier si il y a un mur
   Si oui, passer à la direction suivante
   Si non, vérifier qu'il y a un pion (ce sera forcément celui du joueur adverse).
      Si oui, pour les 3 autres cases adjacentes au pion, vérifier si
      l'une d'entre elle a pour valeur : celle actuelle -1.
         Si oui, vérifier si un déplacement est possible entre
         cette case et celle actuelle.
            Si oui, renvoyer la position de cette case.
      Si non, vérifier si la valeur de cette case égale celle actuelle - 1.
         Si oui, renvoyer la position de cette case
         Si non, passer à la direction suivante
```

La complexité en espace de cette fonction est Θ (1) car elle ne crée pas de tableau et ne fait pas d'appels récursifs.
La complexité en temps est Θ (1) car elle ne fait pas de boucle ni d'appel récursif.

Bouge (Plateau, Tableau[9][9] d'entiers, Entier JoueurActuel, Entier distance)

La fonction cherche la case gagnante la plus proche pour le JoueurActuel.

```
Tant que la case n'a pas pour valeur 1:
      case = RemonterChemin(Board, Tableau, case);
```

Enfin, la procédure appelle move_pawn pour conclure le tour.
La complexité en espace de cette fonction est Θ (1) car elle ne crée pas de variables et ne fait pas d'appels récursifs.
La complexité en temps est Θ (distance) car elle fait une boucle de taille distance.

Main (Plateau)
Copie le plateau fourni par le serveur pour faire les tests.

```
Initialiste un tableau de 9 × 9.
Appelle de get_current_player(Plateau) pour connaître le joueur courant.
(Mon plus court chemin <= son plus court chemin)? OU je n'ai plus de mur?
      Si oui, on bouge le pion
      Si non, pour chaque case du plateau : faire
         Copier le plateau, placer un mur horizontal si possible,
         calculer (Son plus court chemin - mon plus court chemin)
         Copier le plateau, placer un mur vertical si possible,
         calculer (Son plus court chemin - mon plus court chemin)
      si il existe un mur qui augmente la différence, le placer
      sinon, on bouge le pion
```

La complexité en espace de cette fonction est Θ (n) avec n la taille du plateau.
En effet, cette fonction appel PlusCourtCheminRec.
La complexité en temps est $\Theta(n^2\sqrt{n})$.
En effet, elle appelle n fois la fonction PlusCourtChemin.

Le développement de cette IA fut tournée en grande partie sur le bon fonctionnement de l'algorithme PlusCourtCheminRec, où de nombreux travaux de réécriture ont permis de passer de 200 lignes de code à 77.
Le "printf" a souvent été utilisé pour le débuggage, notamment pour afficher les valeurs du tableau 9×9.
Cette IA étant l'équivalent d'un Minmax de profondeur 1, elle lui a logiquement servi de base pour son écriture.

4.5 Le MinimaxBot

Cette intelligence artificielle est basée sur une représentation du jeu en arbre. Les fils d'un noeud de l'arbre sont tous les états possibles du plateau de jeu au prochain coup. Chaque noeud contient un entier, c'est une note pour l'état du jeu donné. L'algorithme effectue des appels récursifs pour prédire les coups adverses en considérant que celui-là joue bien et ainsi trouver le meilleur coup. Le jeu correspond donc à une application donnant les valeurs des noeuds dans cet arbre. L'un des joueurs cherchera à maximiser les valeurs que va prendre cette fonction tandis que l'autre cherchera à les minimiser.

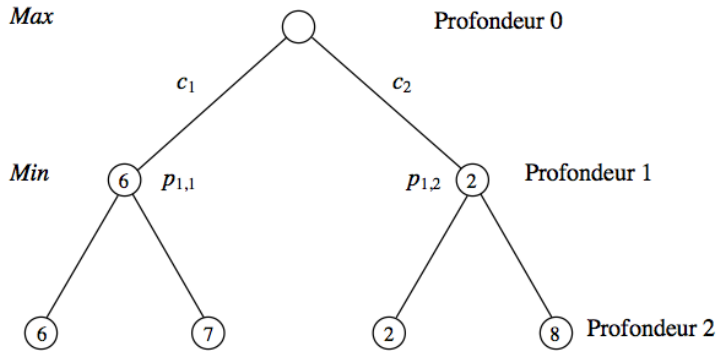

FIGURE 7 – Principe de l'algorithme Minimax

Dans notre cas, la note que nous affectons à un état du jeu est :
$plusCourtChemin(adversaire) - plusCourtChemin(joueurActuel)$.
Si elle est positive ou nulle c'est que le jeu est en faveur du joueur et qu'il doit avancer son pion, sinon il faut qu'il place un mur pour obtenir une meilleure note.

Pour trouver les mouvements possibles, deux solutions se présentaient :

- énumérer les 12 mouvements possibles, mais cela implique d'avoir une très grande quantité de tests pour vérifier la validité de ces mouvements (et à la relecture du code il est difficile de s'y retrouver).
- sélectionner les positions marquées d'un 1 dans le tableau des chemins, mais cela implique une complexité bien plus importante (polynomiale au lieu de constante).

D'autre part, pour les appels récursifs de l'algorithme, on ne peut évidemment pas prédire sur une infinité de coups. On limitera donc le parcours de l'arbre jusqu'à une profondeur maximale. Néanmoins, on considère quand même tous les coups possibles à partir d'un état donné du jeu. Cela implique une complexité en temps et en espace très importante (majorée par la taille du plateau au carré en temps et de l'ordre de $P \times N$ en espace où P est la profondeur maximale et N le nombre de placements de murs possibles + les 12 mouvements).

```
Fonction Minimax(p : Plateau, Profondeur : entier)
Déterminer les successeurs p1...pN de p;
Si Profondeur = 0
   retourner donnerUneNote(p);
Sinon
   Si p est un noeud où on maximise
      retourner Max(Minimax(p1, Profondeur - 1), ..., Minimax(pN, Profondeur - 1));
   Sinon
      retourner Min(Minimax(p1, Profondeur - 1), ..., Minimax(pN, Profondeur - 1));
```

Pour améliorer cela, nous avons d'abord fait appel à la correction alpha-beta. Elle permet d'élaguer l'arbre et ainsi de réduire la complexité. En effet, sachant que l'autre joueur cherche à jouer le pire coup pour nous on peut souvent affirmer qu'un sous-arbre ne pourra pas apporter un meilleur coup.

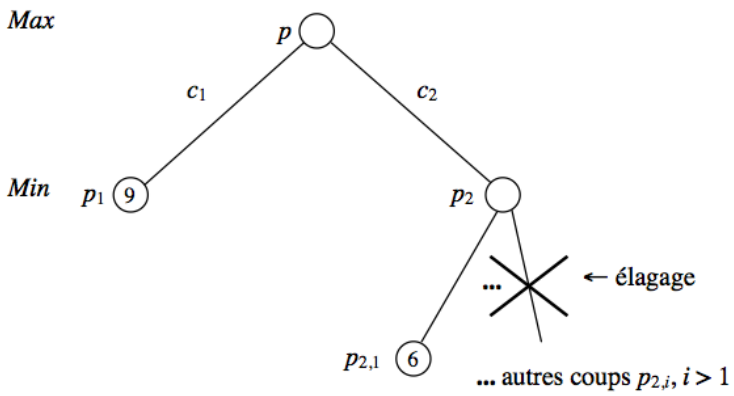

FIGURE 8 – Elagage dans l'algorithme du Minimax

Pour avoir encore de meilleures performances, on préfèrera tester les mouvements possibles les uns après les autres en ne conservant que le coup donnant la note maximale. La complexité en temps se réduit à $N \times Complexite(donnerUneNote)$ et en espace à P.

L'implémentation de cette IA fut particulièrement difficile car pour pouvoir prédire les coups adverses, il est nécessaire de pouvoir simuler un mouvement de sa part. Il a donc fallut implémenter une interface de jeu propre à cette intelligence artificielle avec des fonctions similaires à celles de l'interface commune mais permettant de faire jouer n'importe quel joueur. Le nombre de paramètres qui interviennent en même temps dans le calcul du meilleur coup rendent le débuggage difficile et c'est pourquoi notre MinimaxBot n'est pas totalement opérationnel.

5 Résultats et perspectives

5.1 Tests

Afin de tester les fonctions de l'interface, en plus des fprintf dans la sortie d'erreur, nous avons mis au point un *main()* qui permet de lire un fichier texte de coups. La norme utilisée est celle décrite dans le mémoire de Glendenning (p.16).

```
Exemple :
e8 e2
e7 e3
e7h e2h
```

Ainsi, en lisant des fichiers de coups illustrants des situations symptomatiques, on peut vérifier que les fonctions de l'interface réagissent correctement. De plus, cela nous permet de pouvoir revoir une partie qui a été jouée sur le serveur grâce à l'historique. Par exemple, les parties faisant intervenir le RandBot ont permi de déceler des failles dans nos conditions illustrant les règles.

5.2 Améliorations

Nous discernons plusieurs pistes d'améliorations pour notre projet :

— Une discussion préalable avec les autres groupes aurait permis de mettre tout le monde d'accord sur la forme des fonctions de l'interface et des normes communes.

— La représentation du plateau de jeu que nous avons mise en place est assez facile à utiliser mais dans les fonctions de l'interfaces de nombreux tests sont nécessaires et font appels aux tuilles adjacentes ou aux murs adjacents. Il aurait donc été judicieux d'écrire des fonctions permettant d'effectuer un même test sur les 4 positions adjacentes par exemple.

— Au niveau de l'interface graphique, le non-raffraichissement du plateau entier à chaque tour et une interraction plus poussée avec l'utilisateur auraient pu être envisagés.

— Dans notre projet, tous les Bots ne sont pas totalement opérationnels.

— Pour le MiroirBot, quand il est au contact du pion adverse, il pourrait sauter par dessus en suivant le plus court chemin. Puis, ayant un déplacement d'avance, il chercherait a copier les placements de mur adverse, ou à suivre son plus court chemin si l'adversaire se déplace. Si le bot joue en 1er, il faudrait qu'il place un mur à un endroit inutile puis ensuite copier les mouvement de l'adversaire, sauf si il pose son 10eme mur. Dans ce cas, suivre son plus court chemin.

— Une fois les IA opérationnelles, il aurait aussi pu être possible d'améliorer leurs performances. Notamment pour le ShortestPathBot et le MinimaxBot. Par exemple dans le MinimaxBot, nous pourrions faire une première sélection parmis les murs intéressants à placer au lieu de tous les tester.

Conclusion

Définir un modèle de jeu est indispensable avant toute forme de programmation. Cela permet d'écrire indépendamment chaque fonction et de ne pas les modifier pour qu'elles puissent fonctionner ensemble.
De même, les IA doivent respecter certaines normes pour fonctionner sur notre serveur.

L'interface graphique reproduit assez fidèlement un vrai plateau de jeu, l'ergonomie est donc améliorée.

La dernière partie de notre projet était de réaliser des IA pour ce jeu. D'abord basiques, nos premiers IA servaient essentiellement pour tester les fonctions de jeu. Une fois le jeu exempt de bug, nous avons pu nous tourner vers des IA plus complexes et efficaces.
Réfléchir à des stratégies inédites et complètement différentes peut être intéressant pour avoir une nouvelle vision du jeu.

Bibliographie

Wikipédia, 2014. *Quoridor* [en ligne].

GLENDENNING, Lisa, 2005. *Mastering Quoridor* [en ligne]. Bachelor thesis in Computer Science. University of New Mexico.

Wikipédia, 2014. *Minimax* [en ligne].

Epreuve commune de TIPE, 2013. *Preuve de la correction de l'algorithme alpha-bêta* [en ligne].

Documentation Ncurses, 2005. *Ncurses programming howto* [en ligne].