École Polytechnique de Montréal

INF8215 - IA : Méthodes et algorithmes
Session d'automne 2021

Rapport de projet
Agent intelligent pour le jeu Quoridor

Nom d'équipe Challonge : LesRetardataires

Membres de l'équipe :
Bathylle de la Grandière - 2166208
Augustin Barruol - 2161214

I. Introduction

Le présent rapport porte sur un projet réalisé dans le cours INF8215, qui consiste à réaliser un agent intelligent capable de jouer au jeu de plateau Quoridor, en respectant un temps imparti de 5 minutes. Nous présenterons ici la méthodologie suivie pour construire notre agent, puis les résultats et l'évolution constatée, et enfin nous discuterons ces résultats en évoquant nos difficultés et les pistes d'améliorations.

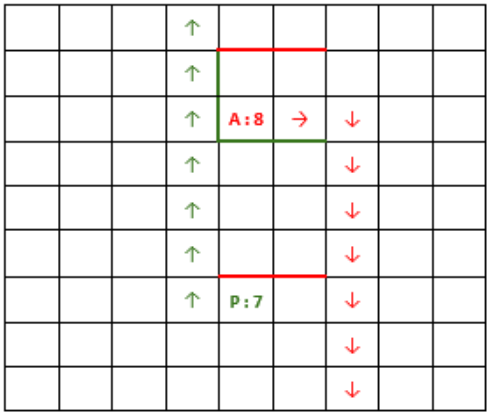
II. Méthodologie

A. Stratégie choisie

Notre agent intelligent repose sur la stratégie de recherche assez basique du MiniMax. Cependant, le nombre de possibilités d'actions à chaque tour de jeu du Quoridor étant tellement grand que nous avons dû réfléchir à des stratégies pour réduire drastiquement ce nombre en excluant d'office certaines actions qui sont peu pertinentes à explorer pour notre joueur.

La première modification évidente est d'ajouter un élagage de type Alpha-Bêta à notre MiniMax pour réduire le nombre de nœuds explorés. Cette modification est très simple à ajouter et n'a posé aucun souci d'implémentation.

Ensuite, nous nous sommes concentrés sur la réflexion suivante : comment trouver les actions les plus pertinentes à explorer pour notre joueur sans avoir à toutes les tester ? Nous avons donc créé une méthode *successors* qui permet de ne sélectionner que les actions qui nous intéressent pour ensuite les explorer dans le MiniMax, et ainsi mettre en place une stratégie de type . La question est : quels sont les nœuds intéressants à explorer pour un état donné ?



Prenons comme visualisation des états notre joueur en vert, l'adversaire en rouge, leurs murs, leur plus court chemin et la longueur de leur plus court chemin dans leur couleur respective. Si on prend l'exemple ci-dessus comme situation de réflexion, les actions les plus logiques à considérer sont :

- les mouvements de notre joueur : évidemment il faut toujours considérer les mouvements de notre joueur pour que celui-ci puisse avancer vers le but sur le plateau
- les murs pour bloquer l'adversaire :
  - Au départ, nous avons commencé par ne considérer que les murs directement autour de l'adversaire (soit 4 murs). Le résultat était correct mais cela faisait que l'IA n'avait pas de vision "long terme" : elle tentait uniquement de bloquer l'adversaire sur l'instant et non pas sur la durée.
  - Pour améliorer ce problème, nous avons ajouté à cela tous les murs dont les coordonnées

étaient communes avec les coordonnées présentes dans le plus court chemin de l'adversaire. Cela faisait déjà beaucoup plus de murs à considérer, mais les résultats étaient bien meilleurs.
    - Toutefois, dans certains cas, l'IA plaçait des murs qui étaient corrects pour ralentir l'adversaire mais pas optimaux, car on se limitait aux murs ayant exactement les mêmes coordonnées que celles présentes dans le plus court chemin de l'adversaire. Pour améliorer cela, nous avons choisi de considérer les 4 murs autour de chaque coordonnées présentes dans le plus court chemin de l'adversaire. Cela fait grandement augmenter le nombre de murs à considérer mais le résultat obtenu était largement meilleur que pour les deux points précédents.

Toutefois, même avec ces restrictions sur les actions à considérer, le nombre d'actions reste grand à chaque itération, c'est nous avons réfléchi à d'autres optimisations pour réduire le temps de calcul.

Tout d'abord, nous avons commencé par utiliser des listes sauf que nous nous sommes rendus compte que les actions étaient parfois dédoublées dans nos successeurs considérés, ce qui faisait exploser d'autant plus les temps de calcul. Pour résoudre ce problème, nous avons utilisé des ensembles (set en Python), qui permettent de nous assurer que les actions possibles ne sont pas considérées plusieurs fois par itération. Cette amélioration a grandement réduit le temps de calcul et nous permis de considérer des profondeurs de recherches plus importantes (nous verrons cela par la suite).

Puis, nous avons ajouté des conditions sur les successeurs à considérer :

- Nous avons décidé que pour les 3 premiers tours de jeu, le plus intéressant était d'avancer le pion, plutôt que de mettre des murs. On ne considère donc aucun placement de mur dans les successeurs.
- Dans le cas où l'adversaire n'a plus de mur et qu'il est plus loin de l'arrivée que le joueur, alors les actions à considérer ne sont que les déplacements du pion (et même uniquement le mouvement vers le plus court chemin). En effet, si l'adversaire n'a plus de mur, alors il ne peut plus nous ralentir dans notre course et si en plus, il est en retard, alors, il n'a aucune chance de gagner.
- Nous avons aussi décidé que si l'IA est en avance sur l'adversaire, alors elle ne considère pas de poser des murs.
- Nous ne considérons que les murs qui nous donnent un avantage par rapport à l'état actuel. S'il n'y en a aucun, alors on se déplace.

Avec ces modifications, le temps de calcul a été grandement réduit tout en explorant une grande partie des états qui peuvent être intéressants pour notre joueur. Évidemment, certaines actions ne sont pas considérées mais il a fallu faire des compromis entre le temps de calcul et le nombre d'actions possibles. Selon nous, cette méthode *successors* est une bonne solution à ce problème car réduit grandement l'exploration tout en se concentrant sur les actions les plus prometteuses.

B. Heuristique appliquée

Afin de choisir une bonne heuristique pour notre fonction d'évaluation, nous avons effectué quelques recherches et exploré plusieurs pistes.

Nous avons d'abord considéré la fonction get_score fournie par la classe Board. Cette fonction nous paraissait très pertinente, mais coûteuse en temps de calcul. Pour réduire ce temps, nous avons cherché à remplacer l'appel à cette fonction par des combinaisons linéaires de plusieurs facteurs. Nous avons donc effectué des recherches pour se documenter sur des travaux déjà réalisés sur le sujet (cf paragraphe références), ce qui nous a mené à considérer des combinaisons linéaires des facteurs suivants :

- distance de notre joueur à sa ligne d'arrivée (coût unitaire)
- différence entre la distance de notre joueur et sa ligne d'arrivée et celle de l'adversaire et sa ligne d'arrivée (coût unitaire)
- nombre minimal de déplacement nécessaire à notre joueur pour arriver à la ligne suivante (recherche en largeur)

- différence entre ce nombre de pas et celui de l'adversaire (recherche en largeur)

Cependant le gain de temps était trop peu important pour la perte de performances en comparaison avec notre agent utilisant get_score.

De plus, notre agent avait tendance à dépenser ses murs très rapidement au cours d'une partie, et la fonction get_score ne considère la différence de murs restants qu'en cas d'égalité de longueur de chemin. Nous nous sommes donc renseignés sur une manière d'intégrer la différence des murs au calcul de l'heuristique. Après plusieurs essais nous avons choisi de régler ce problème directement dans le choix de nos successeurs (fonction *successors*) plutôt que dans la fonction d'évaluation.

Par ailleurs, ce choix ne conduisait pas forcément à bloquer l'adversaire lorsqu'il était à un pas de sa ligne d'arrivée. Pour remédier à ce problème, nous avons affecté les valeurs respectives de ∞ et -∞ pour la ligne d'arrivée de notre joueur et de celle de l'adversaire.

C. Prise en compte du time-out

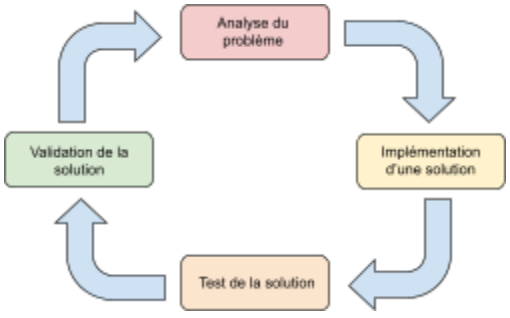
Afin de prendre en compte au mieux le temps imparti, nous avons adapté la profondeur de la recherche MiniMax en fonction du temps restant. Cette décision s'effectue dans la fonction is_terminal, où nous avons segmenté temporellement la partie de cette façon :

Légende : profondeur 1 profondeur 2 profondeur 3 profondeur 4

Nous avons été confronté à un problème de temps en ajoutant aux successeurs à considérer le placement des murs autour de chaque position constituant le plus court chemin de l'adversaire. Cela était dû à une grande redondance des murs considérés, puisque la structure utilisée (une liste) autorise les doublons. Nous avons donc converti cette liste en set, afin d'éliminer tous les doublons. Ce gain de temps nous a permis de passer d'une profondeur d'exploration maximale de 3 à une profondeur de 4.

III. Résultat et évolution de l'agent

Les résultats obtenus ont été corrects dès le départ puisque qu'on a tout de suite utilisé un algorithme de recherche MiniMax avec une bonne heuristique. Les modifications apportées au fur et à mesure du projet (choix des successeurs à considérer, utilisation d'un set, etc.) sont venues améliorer progressivement les performances de notre agent.



À chaque nouvelle implémentation, nous faisions jouer le nouvel agent contre son ancienne version, en premier et en deuxième joueur. Cela nous a permis d'évaluer la qualité de nos nouvelles idées à chaque solution. Nous avons fait en sorte de suffisamment comparer chaque nouvelle version à une ancienne avant de garder l'idée implémentée dans la nouvelle version. Cette méthode d'implémentation est courante en développement informatique puisqu'elle se rapproche du modèle de gestion de projet SDLC (Software Development Life Cycle) qui permet une organisation agile d'un projet.

D'autre part, nous avons aussi fait jouer notre agent contre nous-même, afin de pouvoir orienter la partie comme souhaité (par exemple en choisissant de dépenser beaucoup de murs dès le début afin de bloquer l'agent, ou alors au contraire en priorisant les déplacements de pions). C'est ainsi que nous

avons pu cibler certaines comportements à corriger, comme la dépense prématurée de tous les murs de notre agent.

IV. Discussions

A. Difficultés rencontrées

La mise en place de l'algorithme MiniMax est venue assez naturellement et n'a pas été difficile à implémenter, tout comme les idées d'améliorations progressives.

Les difficultés rencontrées au fil du projet étaient principalement dues à des erreurs comme les levées d'exception NoPath et les soumissions d'action invalides, dont l'origine était parfois difficile à trouver. De plus, la gestion des exceptions NoPath avec des blocs try/except a modifié les performances de notre agent, sans qu'on sache d'où venaient ces changements de comportement puisque ces exceptions ne sont pas fréquentes.

Selon nous, notre IA a une logique correcte et qui permet d'obtenir des résultats convenables. Cependant, les levées d'exception sont vraiment le point négatif de notre implémentation puisque l'on a pas réussi à déterminer leur origine pour chacune des situations rencontrées. Ainsi, si l'on avait plus de temps à consacrer à ce projet, c'est surtout sur le débuggage de ces erreurs qu'on passerait du temps afin que notre agent puisse exploiter ses pleines performances sans imprévu dû aux exceptions. Nous sommes tout de même satisfait du résultat produit par notre agent lors de ce projet, surtout en voyant l'évolution depuis le début de l'implémentation jusqu'à l'ajout de chacune des idées supplémentaires.

B. Pistes d'améliorations

En dehors des exceptions, si nous avions plus de temps, certaines améliorations de l'agent seraient envisageables. Tout d'abord, même si nous avons passé beaucoup de temps à chercher une fonction d'évaluation du plateau plus poussée que get_score, cela paraît surprenant qu'il n'y en ait pas de plus efficace. Donc la première amélioration pourrait être de pousser la réflexion pour la recherche de l'heuristique.

Ensuite, afin de réduire le temps de calcul de notre algorithme à chaque coup, il serait possible de garder une mémoire des parties déjà jouées pour que l'IA puisse aller chercher directement les coups les plus intéressants à jouer. Avec une base de données assez consistante, cela permettrait d'augmenter la profondeur dans notre arbre de recherche pour les situations qui n'ont pas encore été rencontrées.

Avec notre implémentation actuelle, notre agent se contente d'utiliser ses murs pour bloquer l'adversaire, c'est-à-dire de les utiliser de manière offensive. Il est également possible d'utiliser ses murs afin de tracer notre propre chemin, et ainsi empêcher l'adversaire de nous bloquer. L'implémentation de cette idée ne paraît pas compliquée mais il semble que la difficulté réside surtout dans la combinaison de ces deux styles de jeu : complexifier notre heuristique devrait pouvoir nous permettre de bien évaluer quand il est plus important de placer un mur offensif ou un mur défensif. Il pourrait également être pertinent de favoriser l'un ou l'autre des styles de jeu en fonction de si l'agent joue en premier ou en second.

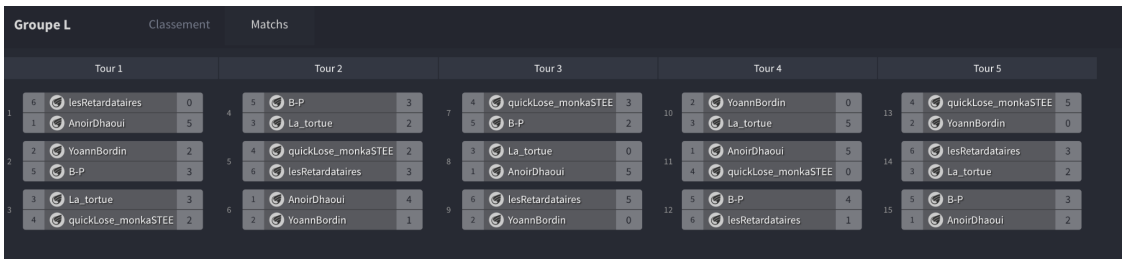
D'autre part, nous avons réfléchi à intégrer des ouvertures ou des fins de partie classiques, comme on peut retrouver aux échecs. Après nous être renseignés sur les différentes ouvertures existantes et en avoir implémenté deux, il s'est avéré que notre agent ne suivait pas la logique mise en place par l'ouverture lorsqu'il revenait à l'algorithme minimax pour le milieu de jeu. Les résultats de cette implémentation étaient décevants (perte inutile de murs par exemple). Pour les situations de fin de partie, nous n'en avons pas envisagé puisque le nombre de situations différentes est gigantesque et donc difficile à prévoir et à implémenter.

Pour conclure, nous sommes satisfait de notre agent, qui arrive parfois à nous battre (bien que nous ne soyons pas des joueurs experts). Nous aurions voulu implémenter plus d'idées, notamment celles évoquées précédemment, mais nous avons passé beaucoup de temps à gérer des problèmes de comportement inexpliqué de notre agent.

V. Résultat tournoi Challonge

Au tournoi Challonge, nous étions satisfaits de gagner plusieurs matchs lors des phases de poules, même si nous n'avons pas été sélectionnés pour la suite de la compétition.

Rang	Participant	1. Points	2. TB	3. Diff Pts	4. Match V-D-N	Historique du match
1	Avancé AnoirDhaoui	21	0	17	0 - 0 - 5	T T T T T
2	Avancé B-P (anenaryon)	15	0	5	0 - 0 - 5	T T T T T
3	Avancé La_tortue	12	0	-1	0 - 0 - 5	T T T T T
4	quickLose_monkaSTEER	12	0	-1	0 - 0 - 5	T T T T T
5	lesRetardataires	12	0	-1	0 - 0 - 5	T T T T T
6	YoannBordin	3	0	-19	0 - 0 - 5	T T T T T



Phases de poules

Notons tout de même qu'il y a eu une égalité parfaite, en nombre de points et en différentiel, entre trois équipes à la troisième place. Alors que nous avons gagné 3 de nos 5 matches (contre 2 seulement pour les deux autres équipes à égalité) et que nous avons battu 3 à 2 l'équipe La_tortue, qui a joué les phases finales, notre équipe lesRetardataires n'a pas passé les poules. L'aléatoire a joué contre nous mais nous avons été ravis de voir notre agent gagner quelques matches !

VI. Annexes : références

- Mertens, P.. (2006). A Quoridor-playing Agent.
  - Notamment pour les différentes heuristiques proposées
- Glendenning, Lisa. “Mastering Quoridor.” (2002).
  - Pour certaines stratégies poussées et diverses heuristiques
- https://fr.wikipedia.org/wiki/Quoridor // https://en.wikipedia.org/wiki/Quoridor
  - Pour la compréhension du jeu et la stratégie de base
- https://quoridorstrats.wordpress.com
  - De nombreuses stratégies sont proposées, notamment pour les ouvertures et autres situations classiques (match miroir, fin de partie, …)