**INF 8175 : Projet Divercité**

# Rapport de projet

Décembre 2024

Alexandre DRÉAN 2408681
Julien SEGONNE 2409827

Nom d'équipe Challonge : theboys


POLYTECHNIQUE
MONTRÉAL

# Table des matières

# Introduction

Ce rapport présente notre travail sur le projet Divercité du cours INF1875. Il comprend la présentation de notre algorithme, une brève analyse de ses résultats et des idées d'améliorations.

# 1 Implémentation de l'agent intelligent

## 1.1 MiniMax

### 1.1.1 Choix de l'algorithme de base

L'algorithme MiniMax étant reconnue dans le domaine de la recherche adversarielle, ayant fait ses preuves avec des agents qui performent très bien aux échecs ou au jeu de go, nous avons décidé d'adopter cet algorithme comme base de notre agent pour le jeu "Divercité".

Nous avons arbitrairement privilégié cet algorithme à l'algorithme de Monte-Carlo comme cela était suggéré dans l'énoncé. On peut tout de même souligner qu'ayant fait et observer de nombreuses parties, nous avons identifié certaines stratégies ce qui nous a permis d'élaborer une heuristique assez tôt dans le projet et qui semblaît être assez prometteuse, il aurait donc été regrettable de choisir un MCTS alors que nous avions déjà des idées de stratégie pour une heuristique.

Concernant l'apprentissage par renforcement qui était aussi proposé dans l'énoncé, nous avons tout de suite écarté cette méthode car elle nécessitait selon nous une logistique importante de par la nécessité d'entrainer un modèle sur la plateforme abyss. Il aurait fallu upload un nombre important de versions du modèle sur abyss car à chaque itération de modèle il faut le faire jouer, récupérer les données des parties pour faire la backpropagation et refaire des parties avec le nouveau modèle, etc...

### 1.1.2 Limitations de l'algorithme MiniMax

Le choix de l'algorithme de base étant fait, nous allons rappeler ses grandes limites qui ont orientées tout le reste de notre travail. Tout d'abord le facteur de branchement est très important, il varie bien-sûr en fonction de l'avancement de la partie mais, par exemple pour le premier coup, il y a **164** ([4 couleurs de cité] × [16 emplacements de cité] + [4 couleurs de ressource] × [25 emplacements de ressource]) possibilités. La profondeur étant de 40, on obtient un nombre d'état (surestimé certes car le branchement ne reste pas à 164) d'environ $164^{40} = 3,9 \times 10^{88}$, il est donc inconcevable d'appliquer l'algorithme MiniMax en l'état. Le reste du travail a donc consisté à réduire le facteur de branchement et bien choisir la profondeur.

## 1.2 Alpha-Beta pruning

La première stratégie utilisée pour limiter le branchement est l'Alpha-Beta pruning. C'est l'algorithme que nous avons directement implémenté exactement comme il est présenté dans le cours. Nous avons cependant ajouté dans cette algorithme récursif le compte du nombre d'états explorés, cela nous a permis dans la suite d'avancer dans notre travail de diminution du nombre d'états explorés. Par ailleurs, nous avons fixé une profondeur maximale à notre arbre de recherche, c'est à dire que nous avons utilisé la stratégie A du cours, on s'arrête à une profondeur donnée et à cette profondeur on choisit comme score l'utilité, qui est égale à la différence des scores des deux joueurs.

En théorie, l'Alpha-Beta pruning permet d'élaguer certaines branches et donc de considérablement diminuer le facteur de branchement, cependant, pour que l'élaguage soit efficace, il est primordiale de commencer en explorant les branches qui sont les plus vraissemblablement jouées. Pour ce faire, nous savons quel coup va jouer notre agent, cela est dicté par notre heuristique (décrite plus bas), et on suppose que l'adversaire veut minimiser notre heuristique. On trie donc les actions à explorer dans l'arbre en fonction de notre heuristique.

## 1.3 Borner le facteur de branchement

Comme nous venons de la voir, à chaque état de l'arbre il faut effectuer un tri en fonction de notre heuristique. Cela est très coûteux et pour cette étape de tri on ne peut pas se permettre de trier seulement la moitié des actions possible sous peine de raté la meilleure action. Le tri est très coûteux mais aussi nécessaire, notre seule possiblité est d'essayer de moins l'utiliser, pour cela on va réduire encore d'avantage les états explorés en explorant seulement les 'num_branch_to_explore' premières branches obtenus dans le trie. Bien que cela ne permet par d'explorer l'intégralité des branches, cela nous spécialise (c'est la stratégie B) dans les actions bonnes selon l'heuristique, et cela nous permet de conserver le tri même s'il est coûteux.

Le choix de la borne 'num_branch_to_explore' a été fait de manière empirique jusqu'à ce que le temps de calcul soit raisonnable et que les performances semblent convenables.

| On a finalement abouti à num_branch_to_explore' = 18. |
|---|

## 1.4 Choix de la profondeur

Par choix de la profondeur on fait référence à la stratégie de type A du cours où on coupe l'arbre minimax à une certaine profondeur maximale et on remplace les noeuds par des feuilles en prenant comme score la différence entre le score des deux joueurs, nous avons égalemment testé en mettant l'heuristique ici mais les résultats étaient moins bons.

Notons $n \in [1,40]$ le numéro du coup où on est rendu dans la partie. Le temps de calcul nécessaire pour effectuer le coup n est proportionnel à la profondeur de l'arbre de recherche ayant pour racine l'état de la partie avant le coup n. Il est donc important de choisir cette profondeur en fonction de n, en choisissant bien à quel moment de la partie on veut allouer plus de temps de calcul pour permettre d'augmenter la profondeur et donc de faire les meilleurs coups. Encore une fois après de nombreux tests, en faisant jouer notre agent contre d'autres agents similaires avec des profondeurs différentes, nous avons constaté que faire augmenter la profondeur au cours de la partie donnait de meilleurs résultats. Plus précisément nous avons les paliers suivants :

1. Pour les coups 1 à 21, on explore à **une profondeur de 5**, comme 'num_branch_to_explore' vaut 18, théoriquement cela représente $18^5 = 1889568$ états. Mais grâce au prunning on explore dans les faits environ entre 12000 et 40000 états (ce chiffre varie en fonction des parties et des coups) ce qui nous évite un time-out, inévitable avec le nombre précédent répété à chacun des 20 premiers coups.
2. Pour les coups 22 à 29, on explore à **une profondeur de 6**, cela représente $18^6 = 34012224$ états mais apès prunning on descend entre 16000 et 19000 états.
3. Pour les coups 30 à 32 , on explore à **une profondeur de 7**, ce sont les coups les plus coûteux, cela représente $18^7 = 612220032$ états mais après prunning on descend sous les 70000 états.
4. Pour les coups après 32 on va jusqu'**au bout de l'arbre** qui est d'une profondeur inférieure à 7 et dont le facteur de branchement finit par être inférieur à la borne fixée 'num_branch_to_explore'. Ces étapes représentent un temps de calcul négligeable : à ce stade, la partie est déjà jouée.

Les différentes profondeurs et les nombre d'états explorés sont illustrés dans la Figure 1 avec notre agent "myPlayer", ils permettent de mieux se rendre compte de l'évolution de la profondeur et de l'aléatoire lié au pruning.

## 1.5 Heuristique

Cette section présente l'ensemble des stratégies permettant de calculer l'heuristique. Les stratégies ont été élaborées en jouant des parties entre nous et en regardant les parties des meilleurs agents sur abyss. L'heuristique est calculée pour un état de jeu donné en additionnant toutes les valeurs calculées présentées par la suite.

### 1.5.1 Delta score

La différence de score entre les deux joueurs est intégrée dans l'heuristique car elle reflète en partie l'état du jeu. Mais elle ne se suffit pas à elle-même car seule, elle ne permet pas de détecter des coups à fort potentiel qui amèneront plus de points par la suite.

### 1.5.2 Diversité de couleurs

A force de jouer, nous nous sommes rendus compte qu'il était primordial d'avoir le plus de couleurs de ressource disponibles dans toutes les situations afin de ne pas être bloqué pour compléter une divercité ou afin de mettre l'adversaire dans la difficulté et le piéger. Ainsi, l'heuristique intègre un malus qui dépend du nombre de couleurs manquantes pour notre agent et un bonus s'il manque des couleurs à l'adversaire. Cette stratégie fonctionne assez bien car il existe beaucoup de coups que l'agent fait par défaut avec la même couleur alors qu'il pourrait le faire avec d'autres, cela a pour effet d'équilibré l'utilisation de nos couleurs.

### 1.5.3 Potentielles divercités

Notre heuristique comporte une détection des potentielles divercités qui s'inspire de l'algorithme pour compter les divercités à la fin du jeu. Ici, nous comptons seulement nos cités entourées de 3 couleurs différentes et une case vide et dont la couleur manquante est encore jouable (nous nous servons de l'algorithme utilisé dans la sous-section précédente). Il fait de même pour l'adversaire (afin d'éviter les états de jeu qui amène l'adversaire à des potentielles divercités) en donnant un malus pour chacune d'elles.
Une illustration marquante de cette partie de l'heuristique est l'apparition de coups menant à deux potentielles divercités côte-à-côte, l'adversaire n'a donc plus que le choix de nous laisser au moins une divercité.

### 1.5.4 Nombre de cités posées (stratégie évolutive)

Une stratégie intéressante est de poser en majorité des cités pour les premiers coups afin de jouer autour par la suite avec nos ressources et faire de bonnes comnbinaisons. Ainsi, les cités posées en début de partie apportent un bonus à l'heuristique. Cette stratégie est en suite désactivée après le début de partie car elle n'est plus intéressante. De plus, elle fait concurrence avec le delta score de l'heuristique (si l'on se fie au delta score, en début de partie, nous jouerons des ressources pour gagner des points sur les cités des adversaires et augmenter le delta) qui n'a pas vraiment de potentiel, ainsi le delta score est désactivé en début de partie.

### 1.5.5 Pondération

Chacune des valeurs calculées (présentées dans les sous-sections précédentes) est pondérée par une constante que l'on modifiait pour contrôler les choix de notre agent et obtenir un équilibre le plus performant possible. Les valeurs suivantes sont celles avec lesquelles nous avons obtenu les meilleurs résultats.

| Stratégie | Pondération |
|---|---|
| Delta score | 1 |
| Diversité de couleur | 2 |
| Potentielles divercités | 3 |
| Nombre de cités posées | 2 puis 0 |

## 2 Résultats et analyse du tri

Comme nous l'avons déjà expliqué, notre agent parcourt l'arbre de recherche à partir d'un état de la façon suivante :

1. Il récupère l'ensemble des états fils possibles à partir de la racine
2. Il trie ces états fils en fonction de l'heuristique
3. Il garde ***seulement les 'num_branch_to_explore' = 18 meilleurs*** états fils
4. Il explore ces état fils (en appliquant les étapes précédentes récursivement) en commençant par le meilleur et en allant à une profondeur 'depth' qui dépend de l'avancement de la partie
5. Le parcourt se fait toujours avec pruning

**Observation** : le tri est appliqué récursivement sur chaque état, cela rajoute un nombre de calculs considérable ce qui fait qu'on ne peut pas explorer un grand nombre d'états.

Le MiniMax le plus basique qu'on pouvait faire sans appliquer de tri ni d'heuristique pourrait être décrit par l'exploration à partir d'un état suivante :

1. Il récupère l'ensemble des états fils possibles à partir de la racine
2. Il parcourt ***tous*** les fils dans un ordre quelconque et il les explore récursivement en allant à une profondeur 'depth_noSort' qui dépend de l'avancement de la partie.
3. Le parcours se fait toujours avec pruning

**Observation** : Il n'y a pas de tri, donc on peut se permettre d'explorer beaucoup plus d'états, cependant, la limite est atteinte extrêmement vite car on ne borne pas le facteur de branchement.

Dans la figure 1 ci-dessous, "Depth" est à comprendre comme 'depth_myPlayer' : la profondeur de notre agent en fonction de la partie. La variable 'depth-noSort' - qui correspond à la profondeur des arbres de l'algorithme noSort - est toujours égale à 3 sauf à la fin de la partie. Cela signifie que l'algorithme noSort utilise des arbres peu profonds, cela est contraint par le fait que le facteur de branchement n'est pas borné (contrairement à myPlayer) ce qui fait beaucoup d'état même pour une faible profondeur.

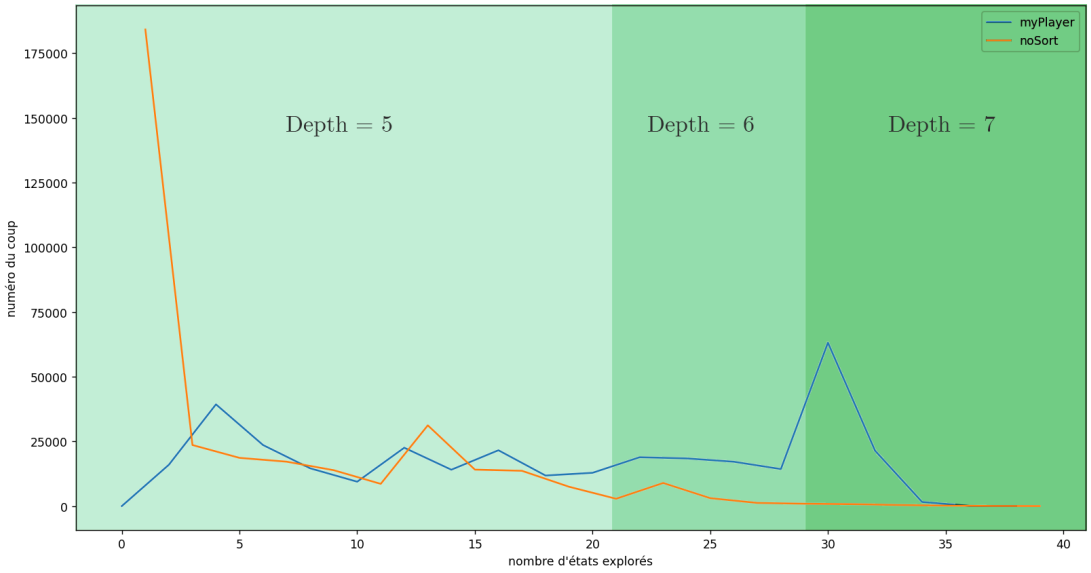


FIGURE 1 – Nombre d'états visités en fonction de l'avancement de la partie. myPlayer correspond à notre agent, noSort correspond à un autre agent qui est un minimax basique avec seulement du pruning et une faible profondeur. Les depths correspondent à la profondeur de l'arbre de recherche de myPlayer

Cette figure montre le nombre d'états explorés en fonction de l'avancement d'une partie entre myPlayer (notre agent) et noSort (un agent de test miniMax basique). L'une des observations importantes liée à cette image est le nombre d'états de myPlayer qui augmente considérablement quand on passe à Depth=7, il s'agit ici des coups qui font la différence, nous aurions pu essayer de rentrer un peu plus tôt dans cette plage pour avoir de meilleurs résultats, mais cela aurait nécessité d'être très prudent sur le temps de calcul pour ne pas faire de TimeOut.

# 3 Idées non implémentées

Cette section présente les idées que nous n'avons pas pu implémenter par manque de temps ou parce que celles-ci étaient trop complexes.

## 3.1 Table de transposition

Lorsque notre agent joue en premier, le coup qu'il joue est déjà implémenté et sera toujours le même, ce qui nous fait gagner une étape de profondeur (non négligeable car l'heuristique peut drastiquement changer avec un coup supplémentaire). Nous aurions pu créer une table de transposition et coder plus de coups à l'avance pour le début et la fin de partie sachant que ces périodes de jeu se ressemblent beaucoup entre les parties. Ceci nous aurait fait gagner plus de temps de calcul (précieux pour augmenter les profondeurs de recherche ou conserver plus d'états prometteurs dans le tri). Nous n'avons pas mis en place cette stratégie car elle obligeait à définir une clé de hash pour chaque état possible du plateau de jeu ce qui forçait à modifié un fichier de code autre que myPlayer et était gênant pour uploader l'agent sur Abyss.

## 3.2 Amélioration heuristique

A l'instar des échecs, jouer au centre du plateau semblent être une bonne stratégie. Ainsi, intégrer à l'heuristique un bonus lorsque des cités sont jouées au centre du plateau aurait pu être une bonne idée. En effet, jouer au centre permet d'offrir plus de combinaisons entre nos cités (comme avec l'exemple des deux potentielles divercités) car elle peut être relié à quatre autres cités, tandis qu'une cité dans le coin ne peut être liée qu'à deux cités. Cela aurait aussi pour effet d'empêcher l'adversaire de mettre en place son jeu.

## 3.3 Symétries du plateau

Encore une fois, pour gagner du temps de calcul, nous aurions pu exploiter certaines symétries du plateau pour que certains états symétriques ne soient pas explorés à plusieurs reprises.

# Conclusion

L'idée générale de notre agent était de se libérer le plus de temps de calcul pour pouvoir améliorer les profondeurs de l'algorithme MiniMax et du tri et d'avoir une bonne estimation du potentiel de chaque coup avec l'heuristique. Avec notre implémentation, notre dernier agent a réussi à vaincre les agents préexistants (random et greedy) et le premier MiniMax que nous avions implémenté. En revanche, contre les autres agents, il n'a pas été aussi efficace (111V - 93D | 1180 elo).