
POLYTECHNIQUE MONTRÉAL

UNIVERSITÉ D'INGÉNIERIE

INF8175: Intelligence artif.: méthodes et algorithmes

**Projet - Agent intelligent pour le jeu Divercité**

**Poisson-Lanterne de l'Atlantide**

Baptiste Rosselet - 2403088

Alexis Lambert - 2403731

**Polytechnique Montréal**

# I Méthodologie

## A Algorithme de recherche

### 1 Minimax

Dans ce projet, nous avons fait le choix d'utiliser l'algorithme minimax, permettant une gestion adversarielle simple et efficace pour Divercité qui est un jeu à somme nulle, déterministe et à information parfaite. L'agent simule les coups possibles pour lui-même et pour l'adversaire sur une certaine profondeur, en cherchant à maximiser son propre score final tout en supposant que l'adversaire cherchera à le minimiser car les agents sont considérés comme rationnels. Cette approche permet d'explorer toutes les combinaisons possibles de coups jusqu'à une profondeur donnée dans l'arbre de jeu, garantissant ainsi une décision optimale à chaque tour. L'algorithme minimax est particulièrement adapté aux jeux comme Divercité où chaque joueur a une connaissance complète de l'état du jeu et où les actions des joueurs s'alternent de manière séquentielle.

### 2 Contrainte de minimax

L'algorithme minimax présente un point négatif principal : le coût de calcul de la recherche pour des jeux avec un fort facteur de branchement comme Divercite, qui, au tour 1, a 164 états possibles (16 cases 'cité' + 25 cases 'ressources' avec 4 couleurs possibles pour chaque).

Dans ce projet, la contrainte principale est le temps : nous n'avons que 15 minutes pour l'ensemble de nos coups. Il faut alors, pour chaque coup, répartir le temps et s'assurer d'optimiser la recherche en explorant le minimum d'états inutiles. Nous voulons pouvoir regarder le plus profondément possible pour chaque coup afin de s'assurer d'effectuer un "bon coup" et éviter l'effet de myopie au maximum.

### 3 Élagage Alpha-Bêta

Une première optimisation très courante avec un algorithme minimax est l'élagage alpha-bêta. Cette technique permet de réduire considérablement le nombre de nœuds à explorer dans l'arbre de jeu en ignorant les branches qui ne pourront pas influencer la décision finale, ce qui permet donc de réduire le coût de la recherche. En appliquant l'élagage alpha-bêta, on peut alors explorer plus d'états car on élimine des états jugés inintéressants. On effectue aussi un tri des coups en utilisant l'heuristique que nous avons implémentée pour favoriser l'élagage.

### 4 Gestion de la profondeur

Comme mentionné précédemment, l'objectif est d'explorer le plus profondément possible dans notre recherche pour trouver la meilleure action possible. Nous avons donc fait le choix d'une stratégie de type A, comme mentionné dans le cours, où l'on parcourt les profondeurs de manière horizontale. Plutôt que de choisir une profondeur fixe, l'agent utilise une approche de profondeur itérative dans la fonction *compute_action*. Il effectue d'abord une recherche complète à profondeur 1, puis à profondeur 2, et ainsi de suite. Il continue d'augmenter la profondeur (*current_depth += 1*) tant que le temps calculé pour le coup (*time_for_this_move*) n'est pas écoulé. La meilleure action trouvée lors de la dernière itération de profondeur complétée est conservée et finalement retournée. Ainsi, pour le temps alloué à chaque action, nous allons essayer d'aller le plus profondément possible. À chaque appel récursif de la fonction *minimax*, nous vérifions si nous avons encore du "budget" temps et, si ce n'est pas le cas, nous générons une exception *TimeoutError* qui sera attrapée par

*compute_action* et qui pourra ensuite retourner la meilleure action qui avait été explorée.

### 5 Table de transposition

Une autre technique efficace pour augmenter la profondeur à laquelle la recherche parvient est l'implémentation d'une table de transposition.

De plus, dans le jeu Divercité, on peut arriver à un même état en provenant de différentes branches, ce qui favorise l'utilisation de tables de transposition pour stocker des états déjà évalués et ainsi éviter de les réévaluer.

Ainsi, les tables de transposition réduisent grandement le nombre d'états à évaluer. Pour implémenter une table de transposition, nous utilisons les ensembles (sets) de Python, qui utilisent un hachage comme clé. Pour garantir que le hachage de deux états ayant les mêmes pièces soit similaire, nous avons implémenté une fonction *_get_state_hash* qui prend les pièces présentes sur le plateau, les trie et ensuite les stocke dans un tuple avec l'ID du prochain joueur qui doit jouer (car l'état n'est pas évalué de la même manière selon quel joueur est en train de jouer). Python hache ensuite automatiquement le tuple lorsqu'on le lui donne en tant que clé.

## B Gestion du temps

Pour faire fonctionner au mieux notre algorithme de recherche, il faut bien répartir le temps sur chacun de nos coups, chaque agent devant jouer 20 coups. Une répartition uniforme serait contre-productive, car les derniers coups permettent de descendre jusqu'en bas de l'arbre et représentent donc du temps perdu. Nous avons estimé de manière subjective que les premiers coups (les 5 premiers) n'avaient pas trop d'impact sur l'issue de la partie et que l'important était dans le milieu de la partie, étant donné que les coups vers la fin de la partie permettent d'atteindre la profondeur maximale. Pour les coups dont les étapes sont de 10 à 25, on attribue alors 15% du temps restant pour effectuer la recherche. Pour les 10 premiers coups, il s'agit du temps restant divisé par le nombre de coups restants, avec un maximum de 60 secondes. Pour les coups des étapes 25 à 40, il s'agit également du temps restant divisé par le nombre de coups restants, avec un maximum de 120 secondes.

## C Heuristique

### 1 Composition de l'heuristique

La partie heuristique est la principale spécificité de l'agent. Tout d'abord, lors du calcul d'évaluation d'un état, on alloue 50% de la note à la différence de score de l'état et les 50% restants à un calcul d'évaluation des cités et des ressources de l'état. Pour les 2 premiers coups, nous avons également ajouté un calcul pénalisant les états éloignés du centre, car nous avons estimé qu'il est plus propice de jouer au milieu lors des premiers coups. Ce "biais" est ajouté avec un facteur de 0,5 au calcul d'évaluation de l'état.

### 2 Évaluation des cités et ressources

Pour réaliser cette évaluation, on parcourt toutes les pièces du plateau et on ajoute la valeur évaluée si elle nous appartient, on soustrait sinon. Les évaluations se basent sur le score ainsi que sur des bonus et des malus en fonction de certains cas spécifiques.

Pour l'évaluation d'une ville, on peut différencier différents cas :

- S'il y a une diversité, le score est de 5.
- S'il y a 2 ressources de la même couleur, mais que cette couleur est différente de la couleur de la ville, on applique alors un malus de 0,5.
- Pour chaque couleur unique autour de la ville, si on n'a pas eu le cas précédent, on gagne un petit bonus de diversité qui vaut 0,1.
- On compte ensuite les ressources de la même couleur que la ville de la même manière que dans le score (+1).

Pour l'évaluation des ressources, l'idée est similaire, mais on évalue les villes autour de la ressource pour savoir comment on les impacte, et on va calculer un impact positif ou négatif en fonction du propriétaire de la ville.

- Si la ville voisine contient 2 ressources de la mauvaise couleur mais de la même couleur que la ressource que l'on est en train d'évaluer, on applique un malus pour la ressource.
- Pour chaque couleur unique autour de la ville voisine à la ressource, si on n'a pas eu le cas précédent, on applique un petit bonus de diversité qui vaut 0,1.
- Si la ressource est de la même couleur que la ville, on ajoute un bonus de 1 de la même manière que lorsqu'on calcule le score.

La construction de l'évaluation s'est faite de manière expérimentale en analysant les actions qui nous paraissaient bonnes ou non. L'idée est de bonifier légèrement le fait de se rapprocher de la diversité de manière à ce que les petits bonus s'accumulent et, en contrepartie, de punir le fait de casser une diversité en appliquant une double couleur. L'heuristique de l'état, étant calculée par une évaluation de chaque ville dont on additionne ou soustrait le score en fonction de l'appartenance, permet, grâce au malus, de bonifier l'action d'annuler une diversité (on applique un malus sur un score que l'on va soustraire à notre heuristique).

# II Résultats et évolution de l'agent

## A Agents utilisés

Pour les comparaisons dans cette section, nous allons utiliser quatre versions différentes de notre agent ainsi que les deux agents fournis, à savoir l'agent 'random' et l'agent 'greedy'. Les différentes versions sont les suivantes :

- **Version 0** : Cette version utilise simplement l'algorithme minimax avec un élagage alpha-bêta, sans tri préalable des actions, et utilise comme heuristique le score de l'état. Cette version explore une profondeur de 4 à toutes les étapes.
- **Version 1** : En plus des fonctionnalités de la version 0, cette version possède une profondeur variable en fonction du budget temps alloué. Il y a trois plages de budget temps :
  - Pour les étapes de 0 à 10, le budget est le temps restant divisé par le nombre d'étapes restantes, avec un maximum de 60 secondes.
  - Pour les étapes de 10 à 25, le budget est de 20% du temps restant.

– Pour les 15 dernières étapes, le budget est le temps restant divisé par le nombre d'étapes restantes, avec un maximum de 120 secondes.

- **Version 2** : Pour cette version, nous avons modifié la répartition du budget temps en supprimant la tranche du milieu pour l'inclure dans la tranche de "fin de partie". Le budget temps est maintenant calculé avec le nombre de coups restants plutôt que le nombre d'étapes restantes. Nous avons également introduit le biais de distance avec le centre pour les 2 premiers coups. De plus, dans cette version, nous avons introduit l'évaluation des états de la même manière que décrite dans la partie 2.
- **Version 3** : Cette version est la version finale. Nous avons ajouté les tables de transposition et avons trié les actions avec notre heuristique pour maximiser les chances d'élagage. Nous avons également réparti le budget temps en trois catégories, comme précisé dans la partie B.

## B Métrique de comparaison

Comme métrique de comparaison, nous avons choisi la profondeur de recherche maximale que notre agent atteint à chaque coup. Étant donné que nous avons opté pour une approche de type A, cette métrique nous semble la plus adaptée, car nous souhaitons explorer le plus profondément possible afin de sélectionner le meilleur coup.

Nous pouvons observer sur la Figure 1 que, au fil des différentes versions de notre agent, la profondeur de recherche a augmenté grâce aux diverses améliorations apportées. Plus précisément, l'Agent V0 maintient une profondeur de recherche constante mais limitée, tandis que l'Agent V1 commence à montrer des variations en fonction du budget temps alloué. Les agent V2 et V3 démontrent une capacité accrue à explorer des profondeurs plus importantes, en particulier vers la fin de la partie, ce qui reflète l'efficacité des optimisations mises en place.

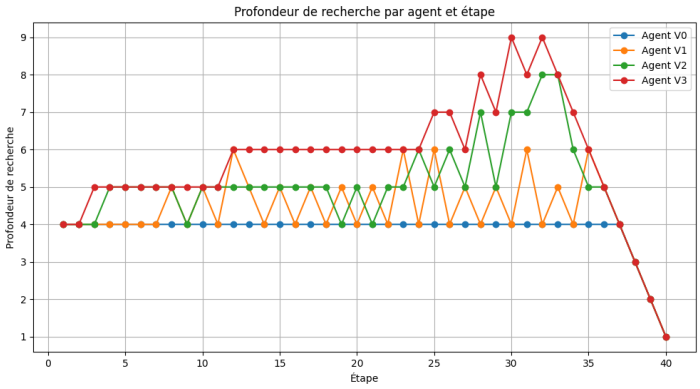


Figure 1: Graphique de comparaison de la profondeur de recherche entre agents

## C Résultats entre agents

On peut observer que, à partir de la version 1, les agents gagnent systématiquement contre l'agent "Greedy". De plus, pour les versions avancées des agents, le côté blanc semble avantagé dans notre implémentation, ce qui pourrait indiquer un biais dans la stratégie ou la gestion des coups. Ces résultats montrent l'efficacité des améliorations apportées aux versions successives de notre agent.

| Noir \ Blanc | V0 | V1 | V2 | V3 | Random | Greedy |
|---|---|---|---|---|---|---|
| V0 | - | V1 | V2 | V3 | V0 | V0 |
| V1 | V1 | - | V2 | V3 | V1 | V1 |
| V2 | V2 | V2 | - | V3 | V2 | V2 |
| V3 | V3 | V3 | V2 | - | V3 | V3 |
| Random | V0 | V1 | V2 | V3 | - | G |
| Greedy | G | V1 | V2 | V3 | G | - |

Table 1: Résultats des matchs entre les différents agents

# III Discussion

La version finale de notre agent présente plusieurs limites, notamment le fait qu'elle suit une stratégie de recherche de type A avec l'algorithme minimax, ce qui implique une forte dépendance à l'heuristique et peut mener à des effets de myopie. Nous pensons avoir une heuristique plutôt bonne, car malgré un agent assez simple, nous arrivons à obtenir de bons résultats sur Abyss (entre 1150 et 1200 elo au moment de l'écriture du rapport).

Pour nous, les principaux avantages de notre agent sont donc sa simplicité : il utilise les techniques vues en cours et une heuristique déterminée selon notre analyse du jeu et des combats qu'il a pu effectuer.

Cependant, il présente des limites et de nombreuses pistes d'améliorations. La valeur des poids pour notre heuristique ainsi que des bonus et des malus pourrait être ajustée avec de l'apprentissage machine en effectuant des tests sur un grand nombre de simulations. Nous pourrions aussi légèrement modifier l'heuristique pour prendre en compte le nombre de pièces restante.

Pour gagner du temps, nous aurions également pu définir une ouverture en définissant des coups fixes pour ne pas dépenser de temps.

De plus, une piste d'amélioration majeure pour réduire le coût de la recherche et ainsi explorer en profondeur serait l'utilisation d'un langage de programmation bas niveau (C/C++ ou Rust) pour la partie minimax, ce qui améliorerait grandement la vitesse en utilisant un langage compilé pour ces opérations.