ECOLE NATIONALE
SUPÉRIEURE
D'INFORMATIQUE

المدرسة الوطنية العليا للإعلام الآلي
(المعهد الوطني للتكوين في الإعلام الآلي سابقا)
Ecole nationale Supérieure d'Informatique
ex. INI (Institut National de formation en Informatique)

Rapport TP2

MinMax avec Iterative Deepening

Réalisé par

- CHENNI Nidhal Eddine
- BENDAHO Sarra

CRP - 2022/2023

Table des matières

Introduction

L'automatisation du jeu d'échecs est l'un des domaines les plus étudiés en intelligence artificielle. Les algorithmes de recherche exhaustive, tels que MinMax avec coupes alpha/bêta, sont souvent utilisés pour implémenter des programmes de jeu d'échecs performants. Ces algorithmes permettent à l'ordinateur d'explorer efficacement l'ensemble des coups possibles en prévoyant les réponses de l'adversaire. La recherche est effectuée jusqu'à une certaine profondeur, en utilisant **trois** fonctions d'estimation pour évaluer la qualité de chaque configuration du jeu. Cependant, la recherche exhaustive peut être très coûteuse en termes de temps et de mémoire, ce qui peut limiter les performances de l'IA en temps réel. C'est pourquoi des techniques telles que l'Iterative Deepening sont souvent utilisées pour optimiser la recherche.

L'objectif de ce tp est d'étudier l'influence de l'Iterative deepening sur le comportement du jeu d'échecs, et comment la fonction d'estimation des configurations peut améliorer ou détériorer les performances du jeu.

Dans ce qui suit, nous allons présenter l'algorithme de l'IDS et son implémentation, proposer une nouvelle fonction d'estimation et finalement, tester et comparer les résultats obtenus après les modifications apportées à l'algorithme d'origine.

L'implémentation de la fonction iterative deepening

Iterative deepening est une stratégie de recherche qui consiste à exécuter une recherche limitée en profondeur plusieurs fois, en augmentant la profondeur de la recherche à chaque fois jusqu'à ce qu'une solution soit trouvée. Dans notre cas, la profondeur de la recherche correspond au nombre de coups que l'algorithme considère comme à venir, mais le temps de la recherche est limité par le temps alloué pour chaque joueur, raison de plus pour l'utilisation d'un approfondissement itératif.

L'utilisation de l'IDS avec MinMax dans ce cas va garantir :

- Une efficacité temporelle : IDS permet à l'algorithme d'effectuer le meilleur coup possible dans le temps imparti. La recherche trouvera toujours le meilleur coup pour la profondeur donnée avant de passer à une profondeur supérieure. Cela réduit le temps nécessaire à la recherche d'une solution et garantit que l'algorithme peut effectuer un déplacement dans une limite de temps donnée.
- Une efficacité spatiale : Étant donné que l'IDS avec Minimax ne conserve que la trace du meilleur déplacement trouvé jusqu'à présent, il nécessite moins de mémoire que les autres algorithmes de recherche. L'algorithme peut rejeter les données de la profondeur précédente et ne conserver que le meilleur mouvement de la profondeur actuelle.
- Une solution optimale : L'algorithme Minimax garantit que la solution optimale sera trouvée en supposant un jeu parfait des deux joueurs. L'utilisation de l'IDS garantit que l'algorithme trouvera la solution optimale en respectant les contraintes de temps et de mémoire.

- Une amélioration de l'ordre des coups : L'approfondissement itératif avec Minimax améliore l'ordre des coups puisqu'il génère l'arbre de recherche de manière extensive. Cela garantit que les meilleurs mouvements sont évalués en premier et réduit ainsi le nombre de nœuds à explorer.

Le programme principal a été modifié pour remplacer les appels à la fonction "**minmax_ab**" par un appel à la fonction "**iterative_deepening**". Cette dernière est l'implémentation de l'algorithme standard de l'IDS et est définie dans le programme comme suit :

```
int iterative_deepening(struct config* conf, int largeur, int mode, int estM) {

    // Initialiser les variables pour stocker le meilleur score et coup
    int best_score = mode == MAX ? -INFINI : INFINI;
    int best_move;

    // Initialiser la profondeur de recherche à 1.
    int depth = 1;

    // Stocker le temps de début de la recherche.
    clock_t start_time = clock();

    // Répéter jusqu'à la limite soit atteinte ou que le temps soit écoulé.
    while ((clock() - start_time) < TIME_LIMIT && depth <= MAX_DEPTH_FOR_IDS) {

       // Effectuer Minimax Alpha-Bêta pour la profondeur actuelle.
       int score=minmax_ab(conf, mode, depth, -INFINI, INFINI, largeur, estM,
       npieces(conf));

       // MAJ le meilleur score et le meilleur coup en fonction du mode actuel.
       if (mode == MAX && score > best_score) {
           best_score = score; // mettre à jour le meilleur score.
       }
       if (mode == MIN && score < best_score) {
           best_score = score; // mettre à jour le meilleur score.
       }

       // Augmenter la profondeur de recherche pour la prochaine itération.
       depth++;
     }

    // Retourner le meilleur score trouvé.
    return best_score;
}
```

Proposition de nouvelles fonctions d'estimation

Dans cette partie, nous allons présenter quelques fonctions d'estimation que nous avons pu développer et intégrer dans le code en nous basant sur des articles de recherche antérieurs. Nous discuterons également de leurs avantages et de leurs inconvénients.

Première proposition : Régression du réseau de neurones

Compte tenu des avancées considérables réalisées par l'apprentissage automatique dans le domaine des jeux, notre première proposition était d'appliquer ces méthodes à la construction d'une fonction d'estimation pour le jeu d'échecs. Cette fonction pourrait ensuite être améliorée en utilisant des données provenant de tournois de jeu d'échecs. Nous nous sommes confiés à un dataset de jeux d'échecs : ChessDB qui est une base de données gratuite. Elle contient un nombre énorme de statistiques sur des parties jouées du jeu d'échecs d'une manière qui permet de trouver rapidement le bon coup. Nous sommes partis de l'hypothèse qu'en utilisant ce dataset, on pourra aboutir à une méthode d'étude très efficace en termes de temps, qui donnera le maximum d'amélioration en un minimum de temps.

En effet, les données de ce dataset ont été utilisées, après les avoir traitées, pour développer un modèle de régression qui prédit la probabilité qu'un joueur gagne en fonction de la configuration du jeu. Nous avons opté pour l'utilisation des réseaux de neurones en se servant de FANN, une bibliothèque C open-source pour la manipulation des réseaux neuronaux, pour la mise en œuvre de la fonction **"estim7"**.

Le réseau de neurones que nous avons utilisé avait deux configurations différentes : 64*20*1 et 64*50*40*30*20*20*1, où l'entrée est la linéarisation de la configuration du jeu (64 octets), et la sortie est soit une victoire, soit une défaite avec une valeur de 100/-100.

- **Avantages :**
    - La précision du modèle peut être améliorée.
    - Une bonne stratégie en fin de partie.
- **Inconvénients :**
    - Pas de stratégie en début de partie.
    - Lenteur par rapport aux estimations statiques.

Deuxième proposition : Tables de score

Aux échecs, il est généralement préférable d'avoir une bonne stratégie dès le début de la partie, plutôt que de s'appuyer sur une bonne stratégie vers la fin. Une stratégie d'ouverture solide jette les bases du reste de la partie, en donnant au joueur le contrôle du centre de l'échiquier et en établissant un plan de développement clair. Il est ainsi plus facile de contrôler le rythme de la partie et de mettre la pression sur l'adversaire dès le début.

Ce qui manquait justement dans la fonction proposée ci-dessus, et ce qui nous a fait penser à considérer une solution qui peut garantir une bonne stratégie dès le début de la partie,et ce en s'appuyant sur la position de la pièce plutôt que sur l'état de la partie.

La nouvelle fonction d'estimation, notée **"estim8",** contrairement au méthode qui utilise l'apprentissage automatique, vise à améliorer l'approche qui attribue des valeurs fixes aux pièces d'échecs. La méthode améliore cette approche en considérant à la fois la pièce évaluée et sa position sur l'échiquier. Il est essentiel de noter que la valeur d'une pièce sur un échiquier dépend fortement de sa position. Par exemple, la valeur d'un pion dans sa position initiale diffère de sa valeur sur l'avant-dernière rangée. La valeur d'une pièce est également influencée par le stade de la partie, c'est-à-dire le début, le milieu ou la fin de la partie.

Pour simplifier la fonction d'estimation, la solution utilise uniquement les valeurs relatives à la position des pièces, plutôt que de considérer l'état de la partie.

Pour mettre en œuvre cette approche, l'étude de Sacha Droste et Johannes Fürnkranz publiée en 2008 a été référencée. Cette étude définit les valeurs des pièces d'échecs en fonction de leur position sur l'échiquier comme le montre la figure suivante:

```
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -3.0, -4.0, -4.0, -5.0, -5.0, -4.0, -4.0, -3.0],
[ -2.0, -3.0, -3.0, -4.0, -4.0, -3.0, -3.0, -2.0],
[ -1.0, -2.0, -2.0, -2.0, -2.0, -2.0, -2.0, -1.0],
[  2.0,  2.0,  0.0,  0.0,  0.0,  0.0,  2.0,  2.0 ],
[  2.0,  3.0,  1.0,  0.0,  0.0,  1.0,  3.0,  2.0 ]
```

```
[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0],
[ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
[ -0.5,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
[  0.0,  0.0,  0.5,  0.5,  0.5,  0.5,  0.0, -0.5],
[ -1.0,  0.5,  0.5,  0.5,  0.5,  0.5,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -2.0, -1.0, -1.0, -0.5, -0.5, -1.0, -1.0, -2.0]
```

```
[  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
[  0.5,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[ -0.5,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -0.5],
[  0.0,   0.0, 0.0,  0.5,  0.5,  0.0,  0.0,  0.0]
```

```
[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0],
[ -1.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0, -1.0],
[ -1.0,  0.0,  0.5,  1.0,  1.0,  0.5,  0.0, -1.0],
[ -1.0,  0.5,  0.5,  1.0,  1.0,  0.5,  0.5, -1.0],
[ -1.0,  0.0,  1.0,  1.0,  1.0,  1.0,  0.0, -1.0],
[ -1.0,  1.0,  1.0,  1.0,  1.0,  1.0,  1.0, -1.0],
[ -1.0,  0.5,  0.0,  0.0,  0.0,  0.0,  0.5, -1.0],
[ -2.0, -1.0, -1.0, -1.0, -1.0, -1.0, -1.0, -2.0]
```

```
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0],
[-4.0, -2.0,  0.0,  0.0,  0.0,  0.0, -2.0, -4.0],
[-3.0,  0.0,  1.0,  1.5,  1.5,  1.0,  0.0, -3.0],
[-3.0,  0.5,  1.5,  2.0,  2.0,  1.5,  0.5, -3.0],
[-3.0,  0.0,  1.5,  2.0,  2.0,  1.5,  0.0, -3.0],
[-3.0,  0.5,  1.0,  1.5,  1.5,  1.0,  0.5, -3.0],
[-4.0, -2.0,  0.0,  0.5,  0.5,  0.0, -2.0, -4.0],
[-5.0, -4.0, -3.0, -3.0, -3.0, -3.0, -4.0, -5.0]
```

```
[0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0],
[5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0,  5.0],
[1.0,  1.0,  2.0,  3.0,  3.0,  2.0,  1.0,  1.0],
[0.5,  0.5,  1.0,  2.5,  2.5,  1.0,  0.5,  0.5],
[0.0,  0.0,  0.0,  2.0,  2.0,  0.0,  0.0,  0.0],
[0.5, -0.5, -1.0,  0.0,  0.0, -1.0, -0.5,  0.5],
[0.5,  1.0, 1.0,  -2.0, -2.0,  1.0,  1.0,  0.5],
[0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0,  0.0]
```

- **Avantages :**
  - Une bonne stratégie en début de partie.
- **Inconvénients :**
  - Il est difficile d'estimer de bonnes valeurs pour les pièces, en particulier en fin de partie.
  - Absence de stratégies fortes pour les fins de partie.

Troisième proposition : L'attaque est la meilleure défense

Pour cette fonction, nous avons essayé d'utiliser deux stratégies en même temps, la première s'occupera de la position des pièces en utilisant **Estim8** et la deuxième se concentre sur l'attaque avec **Estim4**. Au début du jeu, nous utiliserons la méthode de la table pour positionner les pièces à des endroits stratégiques

sur l'échiquier, en adoptant la première stratégie **tant que le nombre de pièces restantes du côté du joueur** qui utilise cette fonction est **supérieur à 10**.

Ensuite, nous favorisons **l'attaque** plutôt que la défense pour obliger l'adversaire à jouer en mode défensif, ce qui permettra au bot de prendre le contrôle sur le jeu. Si le nombre de pièces restantes diminue en dessous de 10, nous adopterons la deuxième stratégie pour exercer une pression plus importante. Bien sûr, les évaluations fournies par ces deux stratégies seront ajoutées aux évaluations déjà calculées concernant le nombre de pièces et leurs poids.

- **Avantages :**
  - Bonne stratégie au début et à la fin du jeu.
- **Inconvénients :**
  - Compliqué.

Tests et comparaisons

I) Iterative deepening

L'IDS est une heuristique utilisée dans la résolution de problèmes de recherche de chemin dans un arbre ou un graphe. Elle consiste à effectuer une recherche en profondeur, limitée à une profondeur donnée, puis à répéter cette recherche avec des profondeurs de plus en plus grandes jusqu'à ce que le nœud de destination soit trouvé. Cette approche combine les avantages de la recherche en profondeur et de la recherche en largeur, en explorant les nœuds les plus prometteurs en premier lieu tout en garantissant que tous les nœuds de l'arbre sont finalement visités.

Il est évident que l'utilisation de l'IDS permet généralement d'obtenir des temps de réponse plus rapides que les algorithmes de recherche en profondeur traditionnels, Dans notre cas, cette affirmation est vérifiée car nous avons constaté que le jeu se termine plus rapidement depuis l'utilisation de l'IDS., Mais le plus important est de savoir si l'IDS maintient ou diminue les performances par rapport à la première solution sans IDS.

Pour évaluer cela, nous avons mis en place un plan de test consistant à sélectionner 10 combinaisons de tests qui ont donné la victoire au joueur maximisant dans le programme sans IDS. Nous allons ensuite répéter ces mêmes combinaisons avec l'IDS en utilisant une profondeur maximale de 20 à 30 et un temps maximal de 5 secondes. En comparant les résultats obtenus avec et sans l'IDS, nous pourrons déterminer si l'utilisation de l'IDS maintient ou diminue les performances dans le jeu d'échecs.

Combinaison des fonction d'estimation	estim6 vs estim5	estim6 vs estim3	estim6 vs estim6	estim3 vs estim3	estim4 vs estim4	estim5 vs estim5	estim2 vs estim1	estim5 vs estim1	estim2 vs estim2	estim6 vs estim1
Victoire du maximisant	oui	oui	oui	oui	oui	non	non	oui	non	oui

Nous pouvons constater que, parmi les 10 victoires obtenues dans le jeu sans IDS, le jeu avec IDS en réalise 7, ce qui signifie qu'il conserve 70% des performances du programme initial sans IDS. Cela confirme la puissance de cette heuristique, qui peut conserver la plupart des victoires même avec une faible profondeur (20 à 30), sans oublier le temps important gagné dans le jeu.

II) Nouvelles estimations

Ensuite, pour évaluer et tester les fonctions d'estimation nouvellement mises en œuvre, nous avons organisé une compétition entre elles ainsi que les autres fonctions existantes. La compétition comprenait cinq matchs pour chaque fonction, en utilisant seulement IDS.
Nous avons ensuite calculé le nombre de victoires et de défaites pour chaque fonction et présenté les résultats dans le tableau suivant pour une meilleure visualisation et compréhension :

MAX\MIN	estim1	estim2	estim3	estim4	estim5	estim6	estim7	estim8	estim9
estim7	2/5	1/5	0/5	0/5	0/5	0/5	5/5	0/5	0/5
estim8	2/5	1/5	0/5	1/5	0/5	0/5	3/5	5/5	1/5
estim9	2/5	2/5	1/5	2/5	1/5	0/5	4/5	3/5	5/5

Lors des tests, il a été constaté que la fonction **Estim7** était légèrement plus lente que les autres et qu'elle ne produisait pas de bons résultats. Toutefois, l'introduction de la fonction **Estim8**, qui utilise des tables de position statiques pour placer les pièces dans les meilleurs endroits possibles en choisissant des paramètres avec de petites valeurs, a entraîné une légère amélioration. Cependant, une fois les pièces bien positionnées, cette fonction ne tient pas compte des autres facteurs tels que l'attaque, la défense ou la protection du roi, ce qui réduit considérablement son efficacité en milieu et fin de partie.

En revanche, la fonction **Estim9**, qui se base également sur les tables de position statique en début de partie pour bien se positionner, mais qui se concentre ensuite sur l'attaque pour le reste de la partie, est beaucoup plus efficace, surtout lorsqu'on utilise des paramètres plus élevés tels que des profondeurs et un nombre d'alternatives plus importants. Pour de petites valeurs de paramètres, les résultats de cette fonction sont souvent similaires à ceux des autres fonctions.

Conclusion

Ce travail démontre l'utilité de l'utilisation de l'algorithme IDS comme heuristique pour améliorer la consommation du temps dans le jeu d'échecs, tout en conservant les performances.

De plus, notre travail fournit des informations solides sur la possibilité d'utiliser l'apprentissage automatique dans le jeu d'échecs. Nous avons proposé une fonction d'estimation basée sur un réseau de neurones de régression, qui n'a pas donné de bons résultats. Cependant, nous croyons que la performance des réseaux de neurones peut être améliorée en ajustant les paramètres du modèle et en utilisant des données plus nombreuses et de meilleure qualité.

En outre, nos deux autres propositions basées sur des tables de valeur des pièces et sur la stratégie "l'attaque est la meilleure défense" ont montré des résultats remarquables pour le jeu d'échecs.

Nous croyons qu'en combinant des modèles d'apprentissage automatique avec des algorithmes de recherche basés sur les règles du jeu d'échecs, tels que celui utilisant les tables de valeur des pièces, nous pourrions créer des programmes d'échecs très performants capables de battre même les plus forts joueurs humains.

Références

- https://www.researchgate.net/publication/322539902_Learning_to_Evaluate_Chess_Positions_with_Deep_Neural_Networks_and_Limited_Lookahead
- L'étude de Sacha Droste et Johannes Fürnkranz
- FANN library
- Dataset ChessDB
- https://github.com/niklasf/python-chess

ANNEXE

Méthodologie employée pour la construction de l'estimation Estim 7

Nous avons suivi la méthodologie suivante :

- **Collecte des données :** Pour cette tâche, nous avons décidé d'utiliser une collection de parties d'échecs publiées sur le site web Kaggle sous le titre "3.5 Millions Chess Games" publié le 06/08/2018 par Miles. Cet ensemble de données contient des données pour plus de 3,5 millions de parties d'échecs depuis 1783, écrites dans un format algébrique.

- **Nettoyage des données :** Pour pouvoir utiliser les données dans l'ensemble de données, nous devons nettoyer les données. Cela signifie :
    - Supprimer les parties inachevées.
    - Supprimer les champs inutilisés.
    - Transformer les parties dans le format utilisé par le programme.
    - Classer les parties en victoires noires et victoires blanches.

  Pour accomplir cette tâche, nous avons écrit un programme Python basé sur un moteur d'échecs open source appelé "python-chess", qui peut être trouvé sur https://github.com/niklasf/python-chess.

- **Construction du modèle**

- **Utilisation du modèle** : Après la construction du modèle, nous exportons les résultats (les valeurs des nœuds du réseau neuronal) dans un fichier externe. Ensuite, dans l'implémentation de la fonction, nous importons ce fichier et reconstruisons le réseau neuronal pour l'utiliser pour essayer de prédire les résultats de l'évaluation.