Rapport IN104

Bastien Hubert, Aurélien Manté

22 mai 2021

Table des matières

1 Introduction

Dans le cadre du cours IN104, nous avons réalisé une intelligence artificielle, pour le jeu de dames (*checkers*) ou de puissance 4 (*connect4*), à l'aide du package `aiarena`. Nous étions libres de rédiger et/ou d'améliorer en particulier les fonctions `minimax` et d'évaluation. Dans ce rapport, nous présentons les raisonnements que nous avons suivis dans l'élaboration de ces fonctions.

Dans toute la suite, ***le joueur*** *fera référence au "brain" en train de jouer au tour considéré. Pour faire référence au "brain" dans l'absolu,* ***l'IA*** *sera utilisé.*

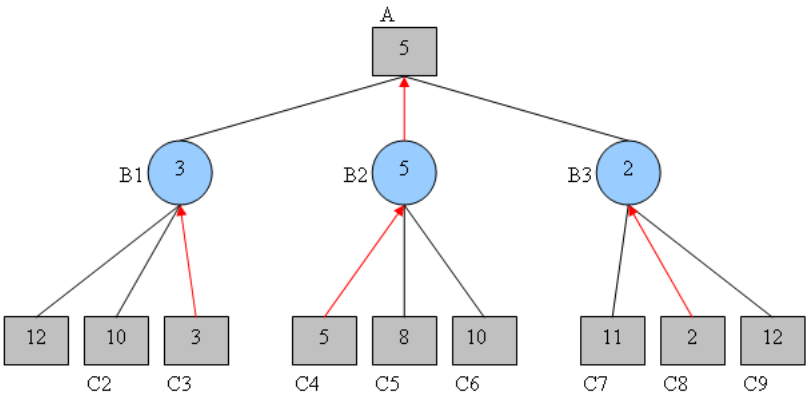
2 Fonction minimax

2.1 Première version de l'algorithme

L'algorithme minimax est un algorithme de théorie des jeux à somme nulle qui consite à maximiser le gain d'un jeu en considérant que l'adversaire tente de minimiser ce gain. Il s'agit donc de maximiser un minimum pour nous, et de minimiser un maximum pour notre adversaire, d'où le nom de cet algorithme.

Il s'agit d'un algorithme récursif de parcours d'arbre, durant lequel on distingue les coups du joueur (maximisation des gains) et ceux de l'adversaire (minimisation des gains). Cependant, les jeux pour lesquels nous allons appliquer cet algorithme présentent un facteur de branchement important (seulement 7 pour le puissance 4 mais d'autres difficultés comme nous le verrons par la suite, 35 en moyenne pour les dames) et un nombre moyen de coups par partie élevé, ce qui rend impossible en pratique le parcours complet de l'arbre jusqu'à ses feuilles (qui représentent une victoire, une défaite ou une partie nulle). Nous devons donc faire appel à une heuristique d'évaluation des états de jeu au bout d'un certain nombre de coups afin que l'algorithme se termine en un temps raisonnable.

Un exemple d'arbre parcouru par l'algorithme minimax est le suivant, où les noeuds en gris sont ceux du joueur, et ceux en bleu sont ceux de l'adversaire :



2.2 Optimisations mises en place

On remarque dans l'exemple ci-dessus que le parcours de l'arbre en entier est inutile, notamment l'étude de la feuille C9. En effet, nous savons que l'adversaire cherche à minimiser notre

gain et que nous cherchons à le maximiser. Sachant que B5 vaut 5 et que C8 vaut 2, nous en concluons que B3 vaudra au plus 2 et ne sera donc pas pris en compte dans l'étape suivante : c'est ce que l'on appelle l'élagage $\alpha\beta$.

De plus, il arrive régulièrement que l'algorithme retombe sur des états dont il a déjà calculé la valeur, notamment quand les joueurs changent l'ordre dans lequel ils jouent leurs pièces. Pour éviter de devoir recalculer les valeurs de ces états, nous enregistrons dans un dictionnaire (appelé table de transposition) les valeurs des états étudiés au moyen d'une clé de hachage, correspondant ici à une chaîne de caractère, et nous vérifions la présence de l'état courant dans la table avant de commencer de nouveaux calculs :

Algorithm 1 minimax_v2(node, maximize, α, β, max_depth, table)

```
if node hash code is in table then
    return table[node hash code]
else
    score ← minimax(node, maximize, α, β, max_depth)
    table[node hash code] ← score
    return score
end if
```

Enfin, en raison du temps de calcul variable de minimax (qui dépend du facteur de branchement de l'état courant, lequel varie en fonction du nombre de pièces disponibles et de l'état du plateau), nous avons implémenté un parcours en profondeur progressif, consistant à faire tourner minimax à une profondeur maximale de plus en plus grande et à retourner la valeur calculée par la dernière itération de minimax (la plus précise donc, car ayant évalué le plus de coups en avance l'état du plateau).

Algorithm 2 minimax_v3(node, maximize, α_0, β_0, table, timeLimit)

```
depth ← 0
while there is time left do
    α ← α0, β ← β0          # voir 5.1 sur l'initialisation de ces valeurs
    max_depth ← max_depth + 1
    table ← ∅
    score ← minimax_v2(node, maximize, α, β, max_depth, table) stopping when timeout
end while
return score
```

3 Fonction d'évaluation pour les dames

3.1 Fonction d'évaluation simpliste

Aux dames, une fonction d'évaluation très simple de l'état courant du plateau consiste à compter la différence entre le nombre de pion possédés par le joueur et celui de son adversaire. Cette fonction d'évaluation se calcule très rapidement et présente une heuristique relativement correcte d'un état du plateau.

Cependant, à l'instar du jeu d'échec, les différents types de pièces sont plus ou moins libres de se déplacer sur le plateau, donc ont un impact plus ou moins important sur le déroulement de la partie. Il convient alors d'accorder une plus grande valeur heuristique aux pièces capables de se déplacer plus librement (ici les `aiarena.checkers.cell.KING`) qu'aux pièces dont le déplacement est limité (ici les `aiarena.checkers.cell.MAN`).

Dans un but de lisibilité du code, nous faisons appel à une fonction auxiliaire `evaluate_color(gameState, color)` qui évalue l'état du plateau en ne considérant que les pièces appartenant au joueur `color`. La fonction d'évaluation finale devient alors :

Algorithm 3 evaluate(gameState)

```
playerScore ← evaluate_color(gameState, myColor)
ennemyScore ← evaluate_color(gameState, ennemyColor)
return playerScore - ennemyScore
```

La version simpliste de notre heuristique telle que décrite ci-dessus s'implémente donc sous la forme :

Algorithm 4 evaluate_color(gameState, color)

```
score = 0
for i a row do
  for j a column do
    if cell in position (i, j) is ours then
      if current cell is MAN then
        score ← score + 2
      end if
      if current cell is KING then
        score ← score + 3
      end if
    end if
  end for
end for
return score
```

Il est important de noter que les pondérations (2 pour un pion et 3 pour une dame) ont été l'objet de tests visant à optimiser l'heuristique : un poids trop faible sur les dames conduit l'IA à adopter une position défensive qui fait stagner le jeu, voire à dévaloriser l'impact des dames adverses ce qui lui fait perdre la partie, tandis qu'un poids trop important mène à des coups très aggressifs de l'IA, qui cherche à tout prix à atteindre l'autre bord du plateau, au risque de parfois sacrifier ses propres pions.

3.2 Version finale

Si le calcul de la version précédente est particulièrement rapide, cette fonction n'en reste pas moins trop simple pour être pleinement satisfaisante, car elle ne prend pas en compte la position des pièces sur le plateau de jeu.

En effet, il est souvent intéressant de ne pas déplacer les pions situés sur la dernière rangée, car ceux-ci empêchent passivement la promotion des pions adverses en dames, en plus de pouvoir protéger les arrières des pions que nous avons engagé sur le plateau en cas de besoin. De plus, le contrôle du centre par nos pièces est extrêment important car il assure la plus grande couverture potentielle du plateau (surtout pour les dames), ce qui nous permet de nous adapter à un plus grand nombre de situations.

Ici encore, les pondérations choisies ont fait l'objet de tests visant à optimiser l'évaluation de l'état courant du jeu. Ces tests ont été réalisés par des membres de l'université de Cornell dans le cadre d'un projet analogue au nôtre : https://github.com/kevingregor/Checkers/blob/master/Final%20Project%20Report.pdf.

Algorithm 5 evaluate_color(gameState, player)

```
score = 0
for i do
  for j do
    if cell in position (i, j) is ours then
      if current cell is MAN then
        score ← score + 5
      end if
      if current cell is KING then
        score ← score + 8
      end if
      if current cell is in back row then
        score ← score + 4
      end if
      if current cell is near center then
        score ← score + 3
      end if
    end if
  end for
end for
return score
```

Pour des questions de réduction du temps de calcul de la fonction d'évaluation (que le profiler Python `cProf` a calculé comme étant la fonction la plus coûteuse en temps), nous avons décidé de ne pas implémenter la recherche de pièces vulnérables ou protégées sur le plateau. En effet, si une pièces est réellement vulnérable, il y a fort à parier que celle-ci sera capturée dans les prochains coups : il est donc préférable de réduire le temps de calcul de l'heuristique afin de pouvoir étudier plus de coups, car l'évaluation du plateau après ces coups montrera vraisemblablement l'existence de pièces vulnérables, car capturées par l'adversaire.

4 Fonction d'évaluation pour le puissance 4

4.1 Structure de la fonction

Nous étions totalement libres dans la rédaction de la fonction d'évaluation pour le jeu de puissance 4. Évaluer un `gameState` de puissance 4 nous a semblé plus complexe que pour

les dames : la méthode consistant à "simplement" faire la différence entre le nombre de pièces blanches et noires ne fonctionne bien sûr absolument pas. Il faut prendre en compte la disposition des pièces.

Afin de simplifier l'évalutation, notre fonction `evaluate(gameState)` fait appel à une sous-fonction `color_score(gameState, color)` qui retourne le score (qui reste à définir) de la couleur passée en argument. D'après le Wiki, cette couleur peut être `aiarena.connect4.cell.WHITE` ou `aiarena.connect4.cell.BLACK`, et la couleur `WHITE` est toujours celle du joueur en train de jouer.

```
def evaluate(gameState):

    # White scores positive, black scores negative
    score_w = color_score(gameState, aiarena.connect4.cell.WHITE)
    score_b = color_score(gameState, aiarena.connect4.cell.BLACK)

    return(score_w - score_b)
```

Listing 1 – Structure de la fonction d'évaluation du puissance 4

4.2 Première version

Notre première intuition a alors été de considérer le plus long alignement de pièces que le joueur a réalisé. Pour cela, la solution sûrement peu optimale mais efficace que nous avons suivie est de parcourir le plateau, et pour chaque case de la couleur `color`, observer ses voisins sur une distance de 3 cases, en s'arrêtant à la première case pas de la couleur `color`. La fonction `color_score` renvoyait ensuite le meilleur alignement. Si un alignement de 4 était trouvé, la fonction arrêtait ses recherches pour renvoyer `score = 4`, car un alignement de 4 signifie une victoire, il n'y a pas de meilleur alignement.

Cette méthode fonctionne en particulier en début de partie uniquement : l'IA a beaucoup de mal à s'adapter à une situation où l'adversaire a bloqué un alignement qu'elle a créé. Par exemple (figure 1), si elle a aligné 3 pions mais que cette ligne est bloquée, elle "ne verra pas l'intérêt" de commencer une nouvelle ligne, donc elle jouera au hasard (tous les coups ayant le même score). Il nous fallait une meilleure méthode.

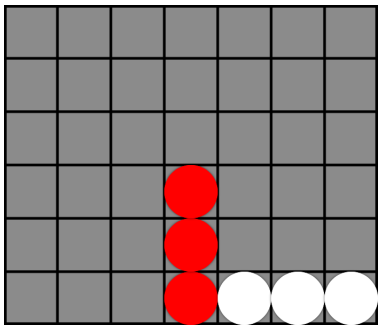


FIGURE 1 – Situation limite de l'algorithme (IA joueur en blanc)

4.3 Version finale

Il faut faire en sorte de prendre en compte les alignements même quand ces derniers ne sont pas les plus longs réalisés : si le joueur a un alignement de 3 pions bloqué, il est préférable de réaliser un nouvel alignement de 2 plutôt que de placer un pion isolé, idem pour un nouvel alignement

de 3. Pour implémenter cela, nous avons modifié la fonction `color_score(gameState, color)` afin de compter les alignements trouvés. Ces alignements sont stockés dans un dictionnaire (on aurait pu utiliser une liste, mais le dictionnaire rend le code plus lisible).

Pour calculer le score, nous attribuons un coefficient de pondération à chaque longueur d'alignement. Ce choix est crucial, il faut par exemple que réaliser un alignement de 3 et 1 pion isolé rapporte plus de points que réaliser deux alignements de 2. Après tests, nous avons fait le choix de ne pas considérer les pions isolés, car cela n'aide pas l'IA mais ajoute en complexité dans la fonction d'évaluation (qui, sans la modifier, les compte plusieurs fois).

En listing 2 est présentée la structure de la fonction de calcul du score par couleur, le détail du compte des alignements est dans le fichier `connect4.py`.

```python
def color_score(gameState, color):

    cells = gameState.cells
    width = gameState.width
    height = gameState.height

    lines = {"2": 0, "3": 0, "4": 0}

    for i in range(height):
        for j in range(width):

            # Detection des alignements horizontaux

            # Detection des alignements diagonaux droits

            # Detection des alignements diagonaux gauches

    for j in range(width):
        for i in range(height):

            # Detection des alignements verticaux

    score_color = lines["2"] + 10 * lines["3"] + 100 * lines["4"]

    return(score_color)
```

Listing 2 – Structure de la fonction de calcul du score

5 Pistes d'amélioration

5.1 Dames

Pour des raisons de temps de calcul, nous n'avons pas implémenté de détection des pièces vulnérables dans la fonction dévaluation des dames. De plus, le temps économisé permet de prévoir plus de tours, ce qui met en évidence les pièces vulnérables (qui seraient alors capturées peu après). Cependant, il serait utile d'essayer une telle amélioration de la fonction d'évaluation, pour évaluer quelle méthode est la plus efficace afin de conserver nos pions.

De plus, on aurait pu encore améliorer minimax en utilisant un $\alpha\beta$ fenêtré, c'est-à-dire en limitant les valeurs initiales de α et β, ici fixées à $\mp\infty$ pour couper plus de branches et trouver un score adéquat en procédant par dichotomie sur les valeurs initiales de α et β.

5.2 Puissance 4

La fonction d'évaluation du puissance 4 pourrait encore être bien améliorée. La principale piste d'amélioration que nous avons repérée est la prise en compte du jeu adverse. Actuellement, l'IA arrive à jouer de façon à maximiser son score, mais elle prévoit mal le score que fera l'adversaire au tour suivant. Nous avons remarqué cela lors de certaines parties IA contre IA où chacune se contentait de réaliser une colonne (figure 2). La première à jouer (IA1) réagissait bien, mais la seconde (IA2) devrait prendre en compte le fait que l'IA1 finira sa ligne de 4 avant elle. Pour cela, il faudrait jouer sur les coefficients de calcul du score pour attribuer aux lignes de 3 adverses un plus gros poids que les lignes de 3 du joueur, afin que ce dernier bloque son opposant.

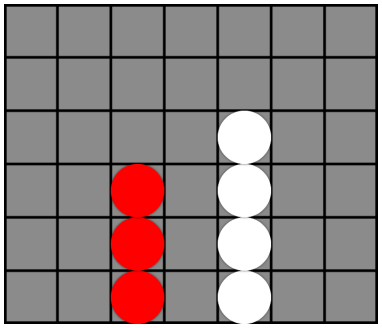


FIGURE 2 – Situation où chaque IA forme une colonne (IA1 en blanc)