IA TP1
Dames anglaises avec MCTS

Vincent Drevelle

Option IA - M1 GL ISTIC - Année 2020-2021

Durée : 2 séances - En binôme
Rendre sur Moodle : code Java + Court rapport avec choix d'implémentation et résultats/comparatifs IA, avant la première séance du TP2.

Dans ce TP, vous implémenterez en Java le jeu de dames anglaises, puis un algorithme de recherche arborescente Monte-Carlo (MCTS) permettant de réaliser des parties contre l'ordinateur.

Des fichiers source Java complets nécessaires à la réalisation du TP ainsi que des squelettes de classes sont fournis dans l'archive *tp1_mcts_src.zip*.

1 Les dames anglaises

Les dames anglaises sont une variante du jeu de dames se jouant sur les cases vertes d'un damier de 64 cases (8x8). Le jeu oppose un joueur avec des pions rouges (aussi appelés noirs) et un joueur avec des pions blanc, possédant chacun au départ 12 pions. Les pions noirs sont placés sur les trois rangées en haut du damier, les blancs sur les trois rangées en bas du damier.

1.1 Règles du jeu

Les cases jouables sont numérotées de 1 à 32. Au début du jeu, les pions noirs sont placés sur les cases 1 à 12, et les pions blancs sur les cases 21 à 32.

Le joueur avec les pions blancs commence la partie.

1.1.1 Déplacements

Les pions se déplacent en diagonale d'une case vers l'avant (c'est à dire vers le camp opposé).

Un pion peut prendre une pièce adverse, en « sautant » par dessus en diagonale, à condition qu'il y ait une case vide derrière. La capture par un pion ne se fait qu'en diagonale vers l'avant.

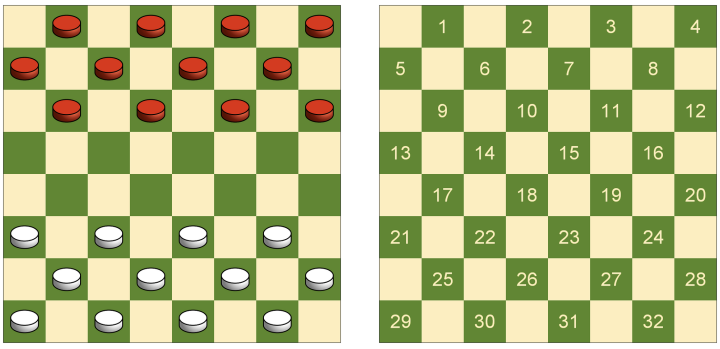

FIGURE 1 – Position de départ et numérotation des cases

Lorsqu'après une première capture, un pion se retrouve en situation de pouvoir capturer un autre pion, il doit continuer de capturer jusqu'à ce qu'il arrive à une position ne permettant plus de capturer. Il n'est pas possible de capturer deux fois le même pion au cours d'une rafle.

La capture est obligatoire (il n'est pas possible de jouer un mouvement de déplacement si on peut jouer un mouvement de capture). Par contre, il n'est pas obligatoire de jouer le mouvement entraînant la capture du maximum de pions.

1.1.2 Promotion en dame

Si un pion arrive, à la fin de son mouvement, à l'extrémité du plateau située dans le camp adverse, il est promu en *dame*. Il n'est pas possible de faire « dame en passant ».

La dame possède la capacité de se déplacer et de capturer aussi bien vers l'avant que vers l'arrière. Son mouvement reste limité à une case, ou à la prise d'une pièce immédiatement adjacente. Au cours d'une capture multiple, la dame peut alterner des déplacements vers l'avant et vers l'arrière.

1.1.3 Fin du jeu

La défaite d'un joueur est déclarée quand :
- soit tous ses pions ont été capturés par l'adversaire,
- soit, arrivé à son tour de jeu, il est bloqué et ne peut plus réaliser aucun déplacement ni capture.

Partie nulle Nous considérerons la règle suivante pour décider d'une partie nulle (c'est-à-dire égalité entre les deux joueurs) :

Une partie sera déclarée nulle si, durant 25 coups consécutifs, aucune prise n'a eu lieu, ni aucun déplacement de pion (autrement dit, pendant 25 coups, seuls des mouvements de dames ont eu lieu, sans capture).

1.2 Notation

On utilise généralement la notation Manouri pour représenter les coups joués. Les cases noires sont numérotées de 1 à 32, de gauche à droite et de haut en bas, en partant de la première case du camp des noirs en haut à gauche.

— Un tiret « - » représente un déplacement simple entre deux cases.
— Une croix « x » représente une prise.

Exemples

11-16 : le pion situé sur la case 11 se déplace vers la case 16.
11x18 : le pion situé sur la case 11 capture le pion en 15 et arrive en 18.
14x23x30 : le pion situé en 14 rafle deux pions et arrive sur la ligne adverse en case 30 (avec promotion en dame).

Cette numérotation des cases et notation des coups sera adoptée dans votre programme (cela est déjà implémenté dans le code fourni).

2 Première partie : Implémentation du jeu de dames anglaises

2.1 `MainGameLoop.java` : Boucle de jeu

La classe *MainGameLoop* fournit une interface console permettant la sélection du mode de jeu ainsi que du type de joueurs.

La fonction *main* implémente la boucle de jeu : demande du mouvement au joueur courant et mise à jour du plateau, jusqu'à la fin de la partie, et affichage du vainqueur.

2.2 `CheckerBoard.java` : Damier, pions, voisinages

La classe *CheckerBoard* fournit une représentation du damier avec les différentes pièces, ainsi que les voisinages des cases (en numérotation Manouri), et de nombreuses fonctions utilitaires pour accéder aux pièces présentes sur le plateau, les déplacer, et réaliser la promotion en dame.

La dénomination d'un pion simple est « checker » et celle d'une dame est « king ». L'état du plateau de jeu est représenté par un tableau d'entiers, dont chaque case contient une constante (EMPTY, BLACK_CHECKER, BLACK_KING, WHITE_CHECKER ou WHITE_KING) correspondant au type de pièce présente sur la case verte correspondante du plateau.

2.3 `EnglishDraughts.java` : Mécanique du jeu de dames et coups possibles

C'est dans la classe *EnglishDraughts* que vous allez implémenter les règles du jeu de dames anglaises. La classe doit permettre de :

— lister les coups possibles,
— jouer un coup,
— déterminer un éventuel vainqueur ou une égalité.

Ceci correspond aux fonctions *possibleMoves*, *play* et *winner* de l'interface *Game* implémentée par la classe *EnglishDraughts*.

2.3.1 Utilitaires

Implémenter les méthodes *isEmpty*, *isAdversary*, *isMine* et *myPawns* dans la classe *EnglishDraughts*, **en vous appuyant sur les méthodes de *CheckerBoard*** (le code fait donc juste quelques lignes). Ces méthodes seront utiles pour analyser le plateau en se plaçant dans le point de vue du joueur courant.

Vous pouvez également, sur le même principe, implémenter des méthodes indiquant les cases voisines vers l'avant et vers l'arrière (vu du joueur courant) pour simplifier le codage des déplacements possibles.

2.3.2 Coups possibles

Une étape importante est la recherche des coups possibles à partir d'une situation de jeu.

Un coup est représenté par un objet de la classe *DraughtsMove*, qui **hérite de *ArrayList<Integer>*** et implémente l'interface *Move*. Un coup en Java est ainsi représenté par la liste des numéros de case parcourus par un pion. Par exemple, le coup 11-16 est représenté par la liste (11,16), et le coup avec prise multiple 14x23x30 est représenté par la liste (14,23,30). La classe *DraughtsMove* implémente l'affichage de la liste en notation Manouri. Il n'y a pas de différence explicite de représentation entre un simple déplacement et un coup avec prise : la distance entre la case de départ et de la case de destination suffit à les distinguer.

Pour pouvoir implémenter la méthode *possibleMoves*, qui retourne la liste des *DraughtsMove* possibles pour la situation de jeu courante, il est recommandé de commencer par écrire :

— une fonction donnant la liste des déplacements simples (sans prise) possibles
— une fonction cherchant, pour une pièce donnée, les déplacements avec prise (et prise multiple) possibles. On pourra utiliser la récursivité ici afin d'explorer en profondeur tous les chemins de prise multiple possibles. Voir l'algorithme 1 pour une idée d'implémentation de cette recherche (il faudra éventuellement passer d'autres paramètres à la fonction afin de tenir compte du type de pion, d'éviter de prendre plusieurs fois la même pièce, etc).
— enfin, une fonction listant tous les déplacements avec prise possible (à partir de chaque pion).

Ainsi, l'implémentation de *possibleMoves* consistera simplement à retourner la liste des coups possibles avec prise ou, si cette dernière est vide, la liste des déplacements simples.

Algorithme 1 prisesPossiblesDepuisCase(*case* : int) → List<DraughtsMove>

```
moves = {}
destSauts = listeDestinationsSautsPossiblesDepuisCase(case)
pour chaque dest ∈ destSauts
    movesPrisesDest = prisesPossiblesDepuisCase(dest)
    si movesPrisesDest est vide
        moves = moves ∪ (case, dest)
    sinon
        pour chaque moveDest ∈ movesPrisesDest
            moves = moves ∪ concaténer( (case), moveDest )
retourne moves
```

2.3.3 Application d'un coup

Compléter la méthode *play(Move aMove)* qui joue le coup *aMove* et met à jour l'état du jeu. Cela consiste à

- déplacer le pion selon le mouvement *aMove* et réaliser les éventuelles captures
- promouvoir le pion en dame s'il termine son mouvement sur la dernière rangée du plateau
- changer le joueur courant (l'enum *PlayerId* possède une méthode *opponent()*)
- mettre à jour le nombre de tours de jeu
- mettre à jour le compteur de mouvements sans capture pour la condition d'égalité

2.3.4 Fin de partie

Écrire la méthode *winner()* qui retourne le cas échéant le vainqueur de la partie (*PlayerId.ONE* ou *PlayerId.TWO*), *PlayerId.NONE* si égalité, et *null* si la partie n'est pas encore terminée.

3 Seconde partie : Monte Carlo Tree Search (MCTS)

La deuxième partie de ce TP consiste à implémenter la méthode de recherche arborescente Monte-Carlo (MCTS) afin de déterminer le meilleur coup à jouer. On imposera une limite de temps de calcul à l'algorithme, l'augmentation de cette limite permettant d'augmenter le niveau de difficulté.

Une classe *PlayerMCTS* compatible avec l'interface utilisée est fournie. Elle s'appuie sur la classe *MonteCarloTreeSearch* dans laquelle vous implémenterez le MCTS.

3.1 `MonteCarloTreeSearch.java`

On rappelle ci dessous l'algorithme général de MCTS, avec une approche *anytime* (possibilité d'interrompre l'exploration à tout moment pour obtenir un résultat de décision).

```
while (il_reste_du_temps) {
    visités = new List<EvalNode>()
    node = racine
    visités.add(node)
    // Sélection
    while (node_n_est_pas_une_feuille) {
        // tree policy pour choisir un fils
        node = choixFils(node.children)
        visités.add(node)
    }

    // Expansion: créer un nouveau fils au hasard
    newFils = expand(node)
    visités.add(newFils)

    // Simulation à partir de newFils
    valeur = rollOut(newFils)

    // Rétropropagation
    for (node : visités) {
        node.updateStats(valeur)
    }
}
return fils_de_la_racine_avec_la_meilleure_valeur
```

Pour la phase de sélection, on utilisera la stratégie UCT (Upper Confidence Bound for Tree), afin de gérer le compromis exploration / exploitation.

Pour la phase de simulation (*rollout*), la stratégie peut être un simple jeu au hasard dans un premier temps (vous pourrez utiliser la classe *RandomPlayer* à cet effet).

Vous pourrez varier le nombre de simulations effectuées depuis le nœud courant.

En raison de la possibilité d'égalité, les scores à propager seront 1 pour une victoire, 0 pour une défaite, et 0,5 pour une partie nulle.

3.2 Analyse

Vous comparerez les résultats de l'IA implémentée lors de matches avec différents temps de calcul autorisés.