Algorithmes et Complexité

TP n° 2 : Programmation dynamique

Université de Lille / FIL

2018

Description du problème

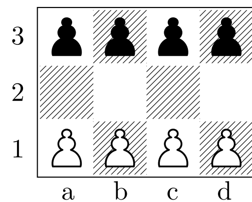
Dans ce TP, on s'intéresse à une variante du jeu *hexapawn* (cf. `https://en.wikipedia.org/wiki/Hexapawn`). Il s'agit d'un jeu d'échec simplifié sur un plateau de taille $n \times m$. Les seules pièces sont les pions. Chaque joueur possède m pions sur sa première rangée. Un joueur gagne si un de ses pions atteint la dernière rangée, ou bien si c'est le tour de l'adversaire et qu'il ne peut pas jouer (absence de coups légaux).

Chaque joueur, lorsque c'est son tour, peut avancer un de ses pions (s'il n'y a pas d'autre pion sur la case juste devant), ou bien il peut capturer un pion adverse en avançant « en diagonale » (cf. `https://fr.wikipedia.org/wiki/Echecs`). Contrairement au jeu d'échec classique, les pions ne peuvent avancer que d'une seule case lorsqu'ils sont sur leur rangée de départ.

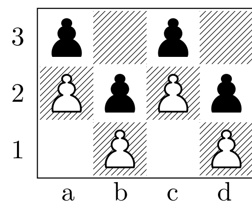
Les coups décrits ci-dessous sont donnés en *Portable Game Notation* [1].

Exemples

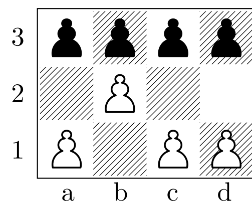
Voici la situation de départ en taille 3×4 :



Dans la situation suivante, celui dont c'est le tour a perdu, car aucun des deux joueurs ne peut déplacer de pion.

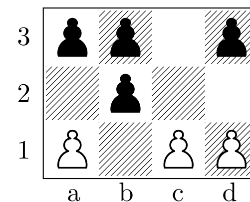


La situation suivante se produit lorsque le joueur blanc joue son pion b (1. b2) à partir de la situation initiale.



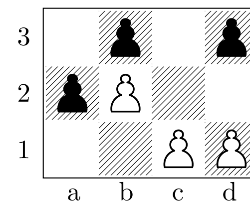
C'est maintenant le tour du joueur noir, et il a cinq possibilités : avancer le pion a (1... a2), prendre en b2 avec le pion a (1... axb2), prendre en b2 avec le pion c (1... cxb2), avancer le pion c (1... c2) et enfin avancer le pion d (1... d2).

Voyons un petit exemple de partie.



Le joueur blanc a avancé son pion b au premier coup, et le joueur noir l'a capturé avec son pion c. Les coups joués ont été 1. b2 cxb2.

La partie pourrait se poursuivre de la façon suivante : le joueur blanc reprend en b2 avec son pion a, puis le joueur noir joue le coup décisif : il avance son pion a (2. axb2 a2).



Et là, le joueur blanc peut abandonner : il n'a aucun moyen d'empêcher le pion en a2 d'atteindre a1, et donc le joueur noir de gagner. Il aurait mieux fallu jouer 2. cxb2 (reprendre en b2 du pion c).

1. `https://fr.wikipedia.org/wiki/Portable_Game_Notation`

Objectif du TP

Le but du jeu est de mettre au point un programme qui joue selon une stratégie optimale, c'est-à-dire qui ne commet *jamais* d'erreur. S'il peut gagner, alors il joue le coup qu'il faut pour gagner. S'il ne peut pas gagner, alors il joue la meilleure défense possible (il essaie de retarder sa défaite).

En effet, dans un jeu comme celui-ci (où il n'y a pas de match nul) on est forcément dans l'une de ces deux situations :

1. Soit le joueur qui commence peut, en jouant correctement, gagner quoi que fasse l'autre.
2. Soit le joueur qui commence, quoi qu'il fasse, va perdre si l'autre joue correctement.

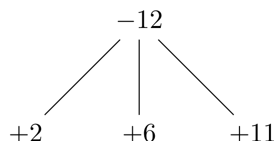
Plus précisément, l'état du jeu (la *configuration*) est complètement décrit par la position de tous les pions, ainsi que par la couleur du joueur dont c'est le tour.

À chaque configuration on associe une *valeur*, qui exprime à quel point elle est favorable au joueur dont c'est le tour. La valeur d'une configuration est l'entier $+k$ si le joueur dont c'est le tour peut gagner en maximum k coups, quoi que fasse l'autre (il peut gagner plus vite si l'autre joue mal) ; la valeur est l'entier $-k$ si le joueur dont c'est le tour ne peut pas gagner mais qu'il peut survivre au moins k coups face à un adversaire optimal (mais il peut éventuellement gagner ou survivre plus longtemps si son adversaire commet des erreurs). La valeur est 0 si le joueur dont c'est le tour a perdu (par exemple parce qu'il ne peut pas jouer, ou bien parce qu'un pion adverse a atteint la dernière rangée).

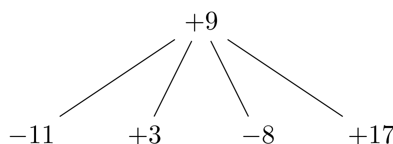
La première étape du TP est de programmer une fonction qui calcule la valeur d'une configuration.

Méthode

Pour programmer cela, on va utiliser une technique de programmation dynamique. En effet, la valeur d'une configuration peut se calculer à partir des valeurs de configurations « plus petites » qui sont accessibles à partir de la première. Disons donc que si on peut passer de la configuration A à la configuration B en un coup, alors B est un *successeur* de A. Considérons une configuration qui a 3 successeurs, dont les valeurs sont respectivement $+2, +6$ et $+11$.



Cela signifie qu'il y a 3 coups possibles, mais que toutes les coups offrent une configuration gagnante à l'adversaire. La configuration de départ est donc perdante, et sa valeur est -12. En effet, en choisissant le 3e successeur, l'adversaire ne pourra pas gagner en moins de 11 coups, et donc comme ça on pourra « tenir » 12 coups en tout avant de rendre l'âme.



Maintenant considérons une configuration qui a 4 successeurs de valeurs $-11, +3, -8$ et $+17$. Cette configuration est gagnante, parce qu'en choisissant (par exemple) le premier coup, on place l'adversaire dans une situation perdante. Mais pour gagner le plus vite possible, il vaut mieux choisir le 3e successeur, car alors on est sûr de gagner en 9 étapes maximum. La valeur de la configuration de départ est donc $+9$.

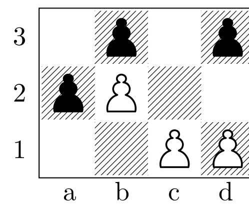
Question 1 : Donnez une formule mathématique permettant de calculer la valeur d'une configuration à partir des valeurs de ses successeurs. Il peut être utile de distinguer le cas où un des successeurs au moins a une valeur négative du cas où ils sont tous positifs.

Programmation de l'évaluation d'une configuration

Entrées / Sorties

Vous devez écrire un programme qui lit n et m ainsi qu'une configuration sur l'entrée standard. C'est toujours au joueur blanc de commencer. La configuration est donnée ligne-par-ligne. Un caractère « P »

signifie un pion blanc, tandis que « p » signifie un pion noir. Par exemple, pour évaluer la configuration suivante :



votre programme recevra sur l'entrée standard :

```
3
4
␣p␣p
pP␣␣
␣␣PP
```

Il doit afficher l'évaluation de la configuration sur la sortie standard, c'est-à-dire dans ce cas précis afficher -2 (en effet, le joueur blanc perd en deux coups : il joue n'importe quoi, puis le joueur noir joue pousse le pion a et gagne).

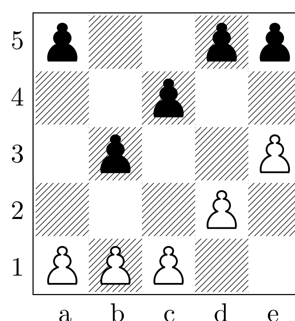
Version naïve

Le plus simple, a priori, consiste à écrire une fonction récursive qui renvoie la valeur de la configuration décrite par ses arguments en s'appelant récursivement pour évaluer les successeurs. La difficulté principale (outre l'implémentation correcte de la formule de la question 1) consiste à trouver une bonne structure de donnée pour représenter l'état du jeu.

Question 2 : Codez une version naïve de cette fonction, qui ne mémorise pas ses résultats. Soumettez-la sur la plate-forme de test. Elle sera testée sur des instances « faciles » à résoudre.

Version « dynamique »

La version naïve est peu performante car elle évalue plusieurs fois la même configuration récursivement. Le phénomène est bien connu des joueurs d'échecs : ce sont les *transpositions*. Pour éviter de répéter les calculs, il faut mémoriser le résultats des évaluations. Par exemple, l'évaluation naïve de la configuration suivante nécessite 14 811 506 999 appels récursifs. En éliminant la réévaluation de configurations déjà évaluées, le nombre d'appels récursifs tombe à 813 830 et le temps d'exécution est divisé par 200.



Question 3 : Codez une version « avec mémoïzation » de la fonction d'évaluation. Soumettez-la sur la plateforme de test.

Les interfaces un peu geek gagnent des points supplémentaires (par exemple en affichant en temps réel le nombre de configuration explorées).