Consortium International e-Miage
&
Université de Picardie Jules Verne d'Amiens

Master MIAGE

Devoir 2, Novembre 2019

Nombre de pages : 4

Code du Module : C217
Intelligence artificielle
Elève : Alessio Rea

I) EXERCICE 1

1.1) Les algorithmes parcourent tous les états possibles d'un problème soit exhaustivement, soit en utilisant une fonction heuristique qui leur permet d'optimiser leur recherche. Ils terminent dès lors qu'ils ont atteint un état solution qui satisfait les contraintes du problème.

1.2) Il s'agit d'un jeu à deux joueurs où chaque joueur ne sait pas comment va réagir l'autre. Il n'est donc pas possible de parcourir l'ensemble des états car ces derniers comprendraient aussi les actions de l'adversaire qui sont inconnues. Il s'agit plutôt de mettre en place une stratégie permettant de déterminer dans chaque configuration du damier l'action qui maximise les chances de gagner. Ceci peut se faire par le biais d'une fonction de gain à maximiser.

Etat initial : nom du joueur devant jouer en premier + la grille est totalement vide. Ceci se formalise de la manière suivante :

quels que soient $1 < i,j < 3$, $a_{i,j} = null$

Fonction de successeur : les cases que le joueur peut jouer sont celles restant libres sur le damier.

Opérateur : positionner le pion sur l'une des cases restantes.

Il est possible de ne pas explorer l'ensemble des noeuds en éliminant d'emblée ceux qui mènent à une victoire imminente de l'adversaire (en mois de deux coups par exemple).

Etat but : les pions alignés sur l'une des trois colonnes, l'une des trois lignes ou l'une des deux diagonales. Ceci se formalise de la manière suivante (en supposant que le programme joue croix) :

quel que soit $1 < i < 3$, $a_{i,1} = X$,
ou bien
quel que soit $1 < i < 3$, $a_{i,2} = X$,
ou bien
quel que soit $1 < i < 3$, $a_{i,3} = X$,
ou bien
quel que soit $1 < j < 3$, $a_{1,j} = X$,

ou bien
quel que soit $1 < j < 3$, $a_{2,j} = X$,
ou bien
quel que soit $1 < j < 3$, $a_{3,j} = X$,
ou bien
quel que soit $1 < i < 3$, $a_{i,i} = X$,
ou bien
quel que soit $1 < i < 3$, $a_{(4-i),i} = X$.

Fonction de gain : Soit M positif et très grand. Dans une configuration donnée, développer à partir de l'état en cours chaque état fils, puis pour chaque fils, développer l'arbre jusqu'à atteindre un état terminal (victoire de l'un des joueurs ou bien damier plein sans vainqueur). Chaque feuille correspondant à la victoire de l'adversaire est affectée du poids -M, chaque feuille correspondant à un match nul est affectée du poids 0 et chaque feuille correspondant à la victoire de l'algorithme est affectée du poids M. La fonction de gain de chaque fils est alors la somme de toutes les feuilles issues de lui-même. Le fils choisi est celui correspondant au plus grand gain. Par ailleurs, il convient de tenir compte de l'imminence d'une défaite ou d'une victoire. Pour cela, à chaque fois que l'on rentre en profondeur dans l'arbre, le coefficient M est divisé par deux. Ainsi, les défaites ou victoires lointaines auront une influence moindre sur la décision que celles qui sont imminentes.

Il est à noter que cette technique est peu efficace du point de vue de la complexité algorithmique et de ce fait infaisable si la taille de l'échiquier était de quatre ou plus au lieu de trois. D'ailleurs, Le temps de chargement est très long au début du jeu si c'est l'algorithme qui doit commencer à jouer en premier, car il doit explorer tous les coups possibles de chaque joueur. Il serait possible de réduire cette complexité en explorant une profondeur limitée puis en utilisant une heuristique pour terminer l'évaluation.

Cependant, pour un échiquier de taille trois, l'algorithme fonctionne très bien dans le sens où il ne fait jamais d'erreur « bête » (comme ne pas empêcher une victoire imminente de l'adversaire ou ne pas profiter d'une victoire facile) et il prend toujours des décisions stratégiquement intéressantes.

Enfin, le fichier *itération.txt* démontre les explorations de noeuds effectuées par l'algorithme implémenté dans le script python *tictac.py* en annexe.

II) EXERCICE 2

a) En largeur :
{S} -> {S-A,S-C} -> {S-C,S-A-E,S-A-B} -> {S-C-D, S-C-G2,S-A-E,S-A-B}.

Solution : S-C-G2

b) En profondeur :
{S} -> {S-A,S-C} -> {S-A-E, S-A-B,S-C} -> {S-A-E-G1, S-A-E-G2, S-A-B,S-C}.

Solution : S-A-E-G1

c) En coût uniforme :
On ne tiendra pas compte des arcs qui nous font retourner à l'origine S, ni ceux qui partent d'un noeud solution G1 ou G2.
{(S,0)} -> {(S-A,2), (S-C,3)} -> {(S-A-B,3), (S-C-D,4), (S-C-G2,8), (S-A-E,10)}

-> {(S-A-B-D,4), (S-C-D-G2,5), (S-A-B-G1,7), (S-C-G2,8), (S-C-D-G1,9),
(S-A-E-G2,17), (S-A-E-G1,19)}

-> {(S-A-B-D-G2,5), (S-C-D-G2,5), (S-A-B-G1,7), (S-C-G2,8), (S-A-B-D-G1,9),
(S-C-D-G1,9), (S-A-E-G2,17), (S-A-E-G1,19)}

Solution : S-A-B-D-G2 avec un poids total de 5.

d) En profondeur itérative :

L=0 : {S}
L=1 : {S} -> {S-A, S-C}
L=2 : {S} -> {S-A, S-C} -> {S-A-E, S-A-B, S-C} -> {S-A-E, S-A-B, S-C-D, S-C-G2}

L'algorithme s'arrête puisqu'on a trouvé un noeud solution G2.

Solution : S-C-G2

e) Best first search :

{(S,5)} -> {(S-A,2), (S-C,3)} -> {(S-A-B,1), (S-C,3), (S-A-E,6)}
-> {(S-A-B-G1,0), (S-A-B-D,1), (S-C,3), (S-A-E,6)}

L'algorithme s'arrête car nous avons atteint un noeud solution.

Solution : S-A-B-G1.

f) A* :

{(S,5)} -> {(S-A,2+2=5), (S-C,3+3=6)} -> {(S-A-B,3+1=4), (S-A-E,10+6=16), (S-C,3+3=6)}
-> {(S-A-B-C,4+3=4), (S-A-B-D,4+1=5), (S-A-B-G1,7+0=7), (S-C,3+3=6)
(S-A-E,10+6=16)}

L'algorithme s'arrête car nous avons atteint un noeud solution.

Solution : S-A-B-G1.

II) EXERCICE 2

a) L'algorithme élimine des noeuds enfants sur la simple condition qu'ils ont déjà été visités. Or, un noeud pourrait très bien à ce stade de l'algorithme avoir un coût chemin moins élevé et par conséquent une fonction d'évaluation f plus avantageuse car f ne dépend pas seulement du noeud en question mais également du chemin par lequel il a été atteint. Des chemins optimaux peuvent donc être éliminés.

b) Il convient de réécrire l'algorithme en tenant compte de cette remarque et donc ne pas éliminer inconditionnellement ces noeuds enfants, mais plutôt tenir compte du chemin par lequel ils ont été atteint (par le biais de la fonction *coûtChemin)* :

F = {D}
Tant que *F* n'est pas vide
    Prendre *F1*, le premier élément dans *F*
    Noeuds_enfants = développer(*F1*)
    Pour Chaque noeud de *Noeuds_enfants* ayant déjà été visité, faire
        conserver des deux noeuds uniquement celui ayant le *coûtChemin* le moins élevé
    Pour tous les noeuds restant dans *Noeuds_enfants*, faire
        Si l'enfant est un état but
            Retourner Succès et Sortir
    Fin pour tous
    Ajouter les enfants à F
    Trier *F* selon la fonction suivante : *f = coûtChemin(D à noeud) + h(noeud)*
Fin tant que