Travaux pratiques - Algorithmes d'exploration et de mouvement

Algorithme MinMax

Dans ce TP, nous étudions une implémentation possible de l'algorithme *minimax*.

1 Jeu de Morpion (Tic-tac-toe)

Le morpion est un jeu de réflexion qui se joue à deux et chacun place à son tour une marque de sa couleur sur une grille de 3×3 cases (respectivement $m \times m$ cases). Le but des jeux de morpion est d'aligner en premier trois (respectivement m) marques de sa couleur sur la même ligne ou la même colonne ou le même diamètre diagonal. Les marques peuvent être des pions ou des symboles. Le premier qui parvient à aligner 3 (respectivement m) de ses symboles gagne un point. Le match est nul, si après le remplissage de la totalité des cases, aucun joueur n'a pu réaliser cet alignement. Une grille de 3×3 cases est présentée en Figure 1. Dans ce jeu de morpion, nous avons deux joueurs le premier joue avec des ronds (fond gris) et le deuxième joue avec des croix (fond orange).

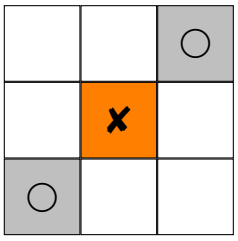

FIGURE 1 – Une grille de 3×3 cases.

L'ensemble des coups possibles est classiquement visualisé sous forme d'un arbre où chaque nœud est un coup. Les feuilles de l'arbre représente une position finale (gain d'un des deux joueurs ou nul). Considérons par exemple la grille de morpion de 3×3 cases. L'arbre visualisant les coups possibles est présenté en Figure 2.

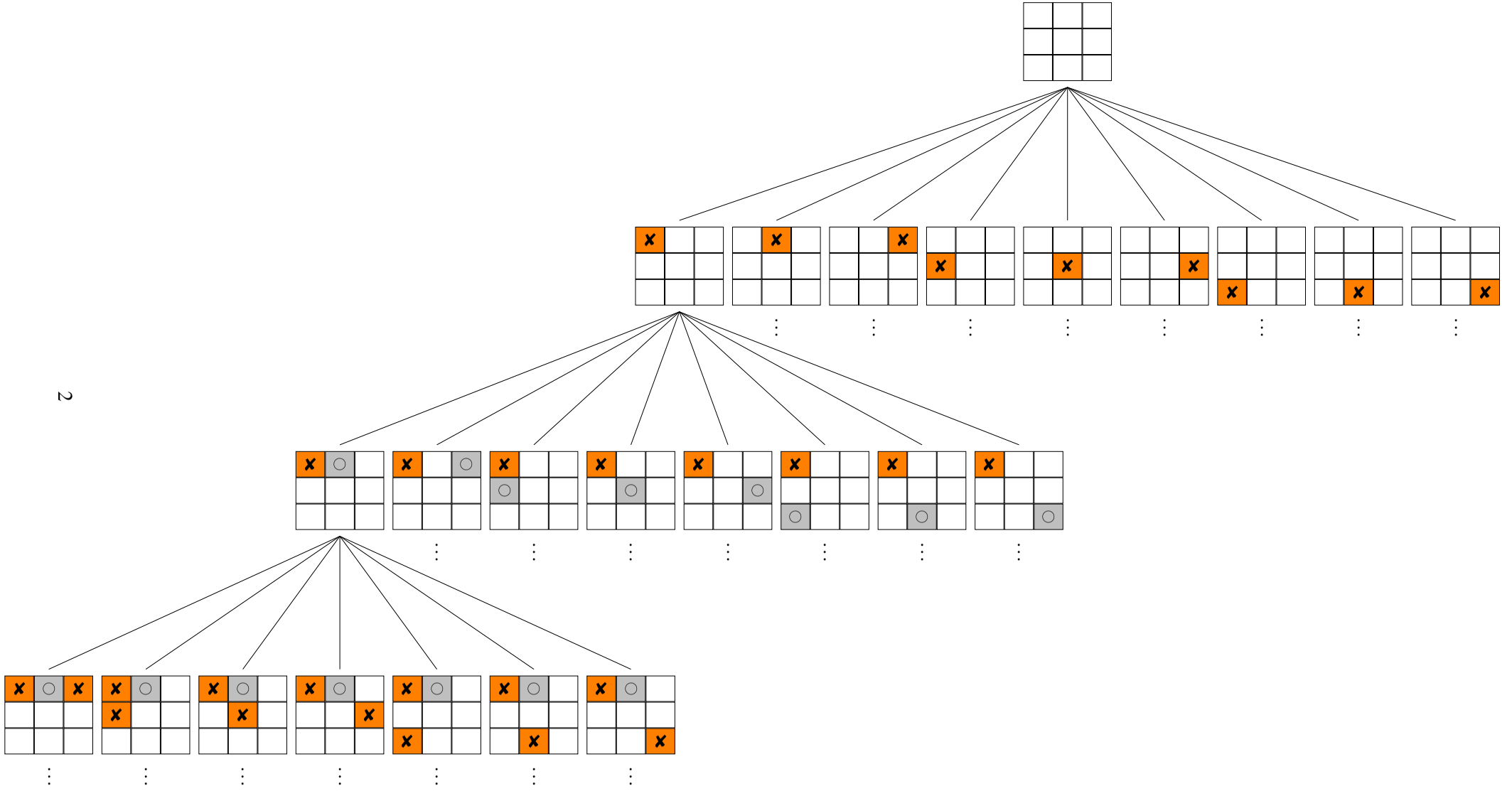

FIGURE 2 – L'arbre visualisant les coups possibles du morpion sur une grilles de 3×3.

2 Minimax

Le principe de l'algorithme `minimax` se base sur la minimisation de la perte maximum possible car on suppose que notre adversaire, dans le jeu, joue parfaitement. On suppose donc que l'adversaire toujours jouera ce qui sera optimal pour lui et on ne suppose jamais qu'il fera une erreur. Dans l'algorithme `minimax` on cherche à explorer toutes les successions de coups et on choisit celle qui minimise les gains de l'adversaire (mini) tout en maximisant les gains du joueur (max). Le pseudo-code de `minimax` est présenté dans l'Algorithme 1.

Algorithme 1: Algorithme *minimax*.

```
procedure minimax(n: node)
1  if ( isLeaf(n) ) then return f(n);
2  else
3      if ( isMinNode(n) ) then
4          β ← +∞;
5          foreach ( s ∈ children(n) ) do
6              β_i ← minimax(s);
7              if ( β_i < β ) then
8                  β ← β_i;
9          return β;
10     else                                        /* n is a max node */
11         α ← −∞;
12         foreach ( s ∈ children(n) ) do
13             α_i ← minimax(s);
14             if ( α_i > α ) then
15                 α ← α_i;
16         return α;
```

$f(n)$ est la fonction du gain. Nous considérons une grille de 3×3 dont la fonction de gain *exact* pour chaque nœud (n) peut être calculer facilement. Si n est une position terminale (n est une feuille de l'arbre), nous pouvons assigner des gain $f(n)$ à n comme suite :

$$f(n) = \begin{cases} 1 & \text{si } n \text{ est gagnant.} \\ -1 & \text{si } n \text{ est perdant.} \\ 0 & \text{si } n \text{ est null.} \end{cases} \tag{1}$$

En partant du bas de l'arbre, nous pouvons donc calculer la fonction du gain $f(n)$ pour chaque nœud (n) dans notre arbre en utilisant l'algorithme `minimax` présenté en Algorithme 1. Dans ce cas on dit que le jeu est simple, i.e. si on peut se permettre d'explorer l'arbre en entier et trouver ainsi la stratégie optimale.

Question. 1 Le but de ce TP est d'implémenter Algorithme 1 (de préférence en Java) pour un morpion défini sur une grille de 3×3 cases. Vous devriez pouvoir jouer contre votre programme à la fin du TP. Donc une petite interface graphique est requise.

2.1 Heuristique

Dans le cas étudié dans la question 1, nous avons considéré une grille de 3×3 cases. Ceci nous a permis d'explorer la totalité de l'arbre de recherche avec l'algorithme `minimax`. Cependant, dans la plupart des cas, on ne peut pas se permettre d'explorer l'arbre en entier ($m \geq 4$). Puisqu'on ne peut pas trouver la fonction du gain "exacte" sans analyser l'arbre jusqu'à une position terminale, on utilise donc dans ce cas ($m \geq 4$) une heuristique pour tenter de trouver une bonne estimation à la fonction du gain.

Question. 2 Proposez une bonne heuristique.

Question. 3 Implémenter l'algorithme `minimax` pour un morpion défini sur une grille de $m \times m$ cases (m est typiquement supérieur à 4 pour rendre le jeu un peu plus difficile).

3 NegaMax

Si l'évaluation de la position pour le joueur J est égale à l'opposée de l'évaluation de la même position pour le joueur opposant O (l'ensemble des valeurs prises par $f(n)$ est symétrique par rapport à 0) alors nous pouvons définir une fonction $g(n)$ telle que :

$$g(n) = \begin{cases} f(n) & \text{si } n \text{ est un nœud de } J. \\ -f(n) & \text{sinon.} \end{cases} \tag{2}$$

Nous pouvons simplifié l'algorithme `minimax` en `negamax` présenté en Algorithme 2.

Ainsi on définit `negamax` à partir de $g(n)$ comme suite :

$$\texttt{negamax}(n) = \begin{cases} g(n) & \text{si } n \text{ est une feuille (position terminale).} \\ \max\{-\texttt{negamax}(s)\} & \text{sinon.} \end{cases} \tag{3}$$

Algorithme 2: Algorithme *negamax*.

```
procedure negamax()
  Input  : n : node ; d ∈ ℕ : depth ;
  Output : α ∈ ℝ ; c : node
1 if ( isLeaf(n) ) then return f(n) ;
2 else
3     α ← −∞;
4     foreach ( s ∈ children(n) ) do
5         α_i ← −negamax(s) ;
6         if ( α_i > α ) then
7             α ← α_i;
8     return α;
```

Question. 4 Implémenter l'algorithme `negamax` pour un morpion défini sur une grille de $m \times m$ cases (m est typiquement supérieur à 4 pour rendre le jeu un peu plus difficile).