# Puissance 4

18 février, 4, 11, 18 mars

---

— Le code implémentant les deux premiers exercices (permettant de jouer à deux sur une interface graphique) est demandé pour la troisième séance.
— Un rapport décrivant ce que vous avez implémenté ainsi que le code correspondant est demandé pour le vendredi suivant le dernier TP. Dans celui-ci, vous décrirez (rapidement) les différentes stratégies implémentées. Une évaluation des performances de chaque algorithme sera appréciée.

---

L'objectif des derniers TPs est de réaliser un jeu de puissance 4. On s'intéressera d'abord à réaliser une interface graphique sur laquelle on puisse jouer à deux joueurs puis on réalisera un programme contre lequel on puisse jouer.

## Exercice 1. Le Jeu

Le puissance 4 est un jeu à deux joueurs qui se joue sur une grille comportant 6 rangées et 7 colonnes. À chaque tour, un joueur glisse un pion de sa couleur dans une colonne (non pleine), ce pion glisse jusqu'à atteindre la position la plus basse dans la colonne. Le premier joueur qui a réussi à aligner 4 pions de la même couleurs (à l'horizontal, la verticale ou en diagonale) gagne. Si personne n'a réussi à en aligner 4, il y a match nul.

**a.** Programmer une classe `Plateau` contenant l'état d'un jeu ainsi que les méthodes :
— `void imprime()` – imprime la configuration sur la sortie standard.
— `int joue(int i)` – modifie le plateau pour que le joueur courant ait joué dans la colonne $i$.
— `int gagne()` – rend $i$ si le joueur $i$ a gagné, -1 sinon.

**b.** Écrire une classe de test qui vous permette de jouer à 2 joueurs en mode texte.

## Exercice 2. Construction d'une interface graphique

On se propose maintenant d'ajouter une interface graphique à notre jeu en utilisant la classe `Plateau` ci-dessus.

**a.** Dans un premier temps, afficher des ronds de trois couleurs différentes selon si une case est vide ou remplie par un pion des deux joueurs.

**b.** Récupérer le clic de souris pour pouvoir jouer à deux joueurs sur votre interface et que le jeu vous indique qui a gagné à la fin.

**c.** Ajouter des boutons "Nouvelle Partie", "Quitter".

**d.** Dans le jeu réel, un pion est inséré dans une colonne où il tombe jusqu'à sa position. Pour simuler ça, créer un thread d'affichage et dessiner la chute du pion.

**e.** (Facultatif) Améliorer l'interface graphique : remplacer les ronds par des images, ajouter des menus...

**f.** (Facultatif) Implémenter le choix d'une grille supérieure à $6\times7$.

## Exercice 3. Heuristiques

Dans la suite, on se propose d'implémenter diverses heuristiques afin de pouvoir jouer contre l'ordinateur.

**a.** (Pour se chauffer) : programmer un ordinateur qui joue au hasard.

Le principe de toutes les stratégies que nous allons implémenter est le suivant :

1. À chaque configuration du plateau peut être associé un score indiquant si cette configuration est plutôt bonne ou mauvaise. Une configuration où l'on a aligné quatre points de sa couleur doit avoir le score maximal, celle où quatre pions de la couleur adverse sont alignés doit avoir un score minimal. Il s'agit donc de créer une fonction objectif qui à une configuration donnée associe un entier.
2. À un instant donné, le coup à jouer sera déterminé en fonction de valeurs de cette fonction objectif sur des coups futurs de façon à maximiser les chances de gagner.

Ainsi, meilleure est la fonction objectif, meilleur sera votre algorithme.

**b.** Proposer une fonction objectif et l'ajouter à la classe `Plateau` : `int valeur()`. Dans un premier temps on pourra se contenter d'une fonction simple que l'on améliorera plus tard.

Les algorithmes suivants permettent d'optimiser cette fonction à un tour à l'avance ou plus.

**c.** Implémenter un algorithme glouton.

**d.** Implémenter un algorithme min-max. Quelle profondeur atteignez vous en un temps raisonnable ?

**e.** Si vous n'aviez pas mis votre jeu dans un thread, mettez le maintenant de façon à ce que quand l'ordinateur réfléchit, l'interface graphique ne reste pas bloquée.

**f.** On se donne une limite de temps de 5 secondes par coup. Afin de respecter cette limite, on propose de calculer la stratégie optimale pour une profondeur $1, 2, \ldots$ jusqu'à ce que les 5 secondes soient écoulées. Fait-on beaucoup de calculs pour rien en calculant les première profondeurs ?

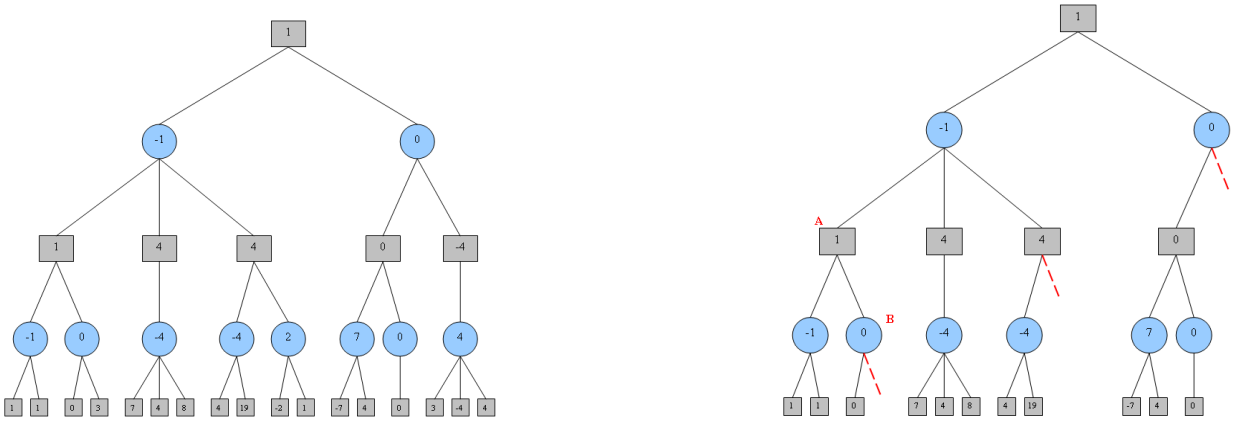


FIGURE 1 – Minmax vs alpha-beta

**g.** L'élagage alpha-beta permet d'accélérer grandement les calculs en évitant de parcourir inutilement certains noeuds tout en donnant le même résultat. On peut donc atteindre une meilleure profondeur. Implémenter l'alpha-beta.

**h.** Choisir une profondeur pour laquelle votre min-max calcule une durée raisonnable. Combien de noeuds l'alpha-beta permet il d'élaguer ? Quelle profondeur pouvez vous atteindre maintenant ?

**i.** Améliorer votre alpha-beta :
— Proposer une nouvelle fonction d'évaluation.
— L'élagage alpha-beta est optimal quand il commence par visiter la meilleure branche d'abord. Pour améliorer son rendement, on peut commencer par trier les coups par "potentiel".

**j.** MTD(f) est une amélioration d'alpha-beta (voir `https://askeplaat.wordpress.com/534-2/mtdf-algorithm/` pour une description de cette technique). Implémenter cet algorithme.