Projet MPRI 2 : Puissance 4 et MCTS

**LANOIX Alexis**

**LIPSKI Guillaume**

# Fonctionnement du programme

## Organisation du code source

Tout d'abord, nous avons choisi d'utiliser le code source fourni et donc par conséquent de réaliser le projet en C.

Par soucis de lisibilité, nous avons séparé notre code en plusieurs fichiers de la façon suivante :

* **main.c** qui contient uniquement la fonction main
* **puissance4.h** et **puissance4.c** qui contiennent les fonctions et tout ce qui est propre au jeu
* **mcts.h** et **mcts.c** qui contiennent les fonctions liées à l'algorithme MCTS
* **utils.h** et **utils.c** qui contiennent quelques fonctions utilitaires

## Compilation du programme

On peut (par exemple) compiler le code avec gcc ainsi :

gcc main.c puissance4.c mcts.c -lm utils.c -o Puissance4

## Lancement du programme

Notre implémentation de l'algorithme MCTS utilise les récompenses {0 ; 0,5 ; 1} respectivement pour une défaite, un match nul et une victoire. La constante C utilisée pour le calcul de la B-valeur est égale à .

Nous avons utilisé la fonction **getopt** (**getopt\_long** plus précisément) afin de proposer diverses options au lancement de notre programme.

Il est possible d'afficher la liste détaillée des options ainsi que leur utilité en utilisant -h ou --help (avec notre exemple de compilation précédent : ./Puissance4 -h).

Voici la liste des options (affichage produit par -h) :

**utilisation :** Puissance4 **[options] [methode]**

(L'ordre n'a aucune importance)

**options :**

**-t** arg (ou **--temps** ou **--time**) avec arg étant un nombre décimal positif non nul (par exemple 1.5).

Permet de définir la limite de temps imposée à l'ordinateur pour l'exécution de l'algorithme MCTS.

Si un nombre d'itérations maximal est également donné, le facteur le plus limitant sera appliqué.

**-i** arg (ou **--iteration** ou **--iterations**) avec arg étant un nombre entier positif non nul.

Permet de définir le nombre d'itérations maximal imposé à l'ordinateur pour l'exécution de l'algorithme MCTS.

Prend le pas sur la limite de temps (de 5 secondes) par défaut. Si une limite de temps est également donnée, le facteur le plus limitant sera appliqué.

**-o** arg (ou **--optimisation** ou **--optimization**) avec arg étant un nombre entier positif non nul.

Permet de définir le niveau d'optimisation/amélioration de l'algorithme MCTS, c'est-à-dire :

**0 :** fonctionnement basique de l'algorithme MCTS avec UCB (UCT) (les simulations sont réalisées au hasard).

**(par défaut) 1 :** (Question 3) amélioration des simulations consistant à toujours choisir un coup gagnant lorsque cela est possible.

**2 :** lorsqu'un coup gagnant est possible, l'algorithme n'est pas utilisé et le coup est joué directement.

**-v** arg (ou **--verbose**) avec arg étant un nombre entier positif non nul.

Permet de définir le niveau de verbosité du programme, c'est-à-dire :

**0 :** aucun affichage autre que la demande de coup et le plateau de jeu.

**(par défaut) 1 :** (Question 1) affichage (à chaque coup de l’ordinateur) du nombre total de simulations réalisées (= nombre d'itérations) et d'une estimation de la probabilité de victoire pour l’ordinateur en jouant ce coup.

**2 :** affichage (à chaque coup de l’ordinateur) du temps passé dans la boucle principale de l'algorithme MCTS et du nombre d'itérations réalisées.

**3 :** affichage (à chaque coup de l’ordinateur) du nombre de simulations réalisées pour chaque coup.

**4 :** affichage (à chaque coup de l’ordinateur) de la moyenne des récompenses pour chaque coup.

**methode :** {**-r** (ou **--robuste** ou **--robust**) | **-m** (ou **--max**) } :

Permet de définir la méthode pour choisir le coup à jouer à la fin de l'algorithme MCTS :

**(par défaut) -r** pour robuste (coup avec le plus grand nombre de simulations).

**-m** pour max (coup avec la plus grande moyenne des récompenses).

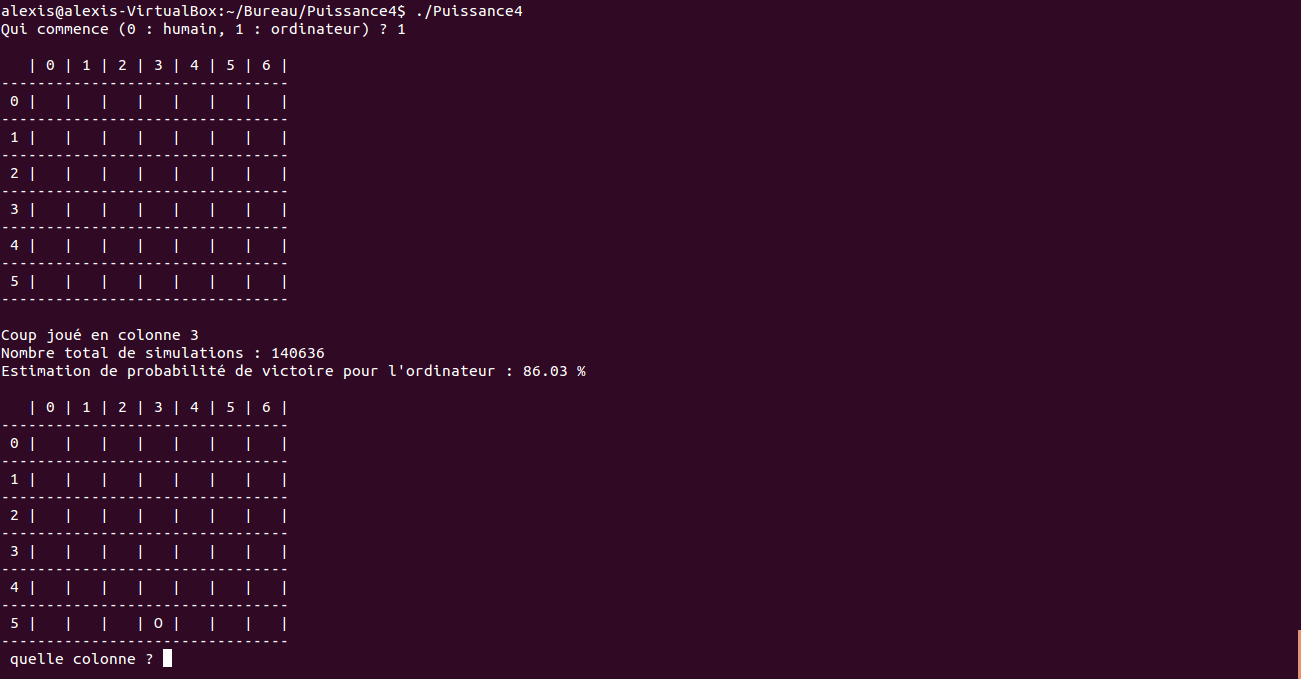
**Par défaut**, l'algorithme utilise donc le critère **robuste** avec **l’amélioration des simulations** consistant à toujours choisir un coup gagnant lorsque cela est possible (question 3) et une **limite de temps de 5 secondes**. Il **affiche** également le **nombre total de simulations réalisées** et **une estimation de la probabilité de victoire** pour l’ordinateur en jouant ce coup (question 1).

Parmi toutes les options disponibles, les plus intéressantes sont évidemment celles permettant de choisir la méthode de choix du meilleur coup, la limite de temps et du la limite du nombre d'itérations qui nous sont très utiles pour tester le comportement du programme en fonction des paramètres de l'algorithme.

# 2) Questions

## 2.1) Question 1

Voici une capture d'écran de l'affichage produit :



## 2.2) Question 2

Au Puissance 4, mathématiquement parlant, si le joueur qui commence fait une "partie parfaite", il gagnera forcément. On peut donc supposer qu'à partir d'un certain temps, notre programme nous battra obligatoirement à tous les coups si c'est lui qui commence (ce qui n'est pas forcément vrai avec l'algorithme MCTS).

On peut donc couper la question en deux en fonction du joueur qui commence.

Bien qu'il nous soit en théorie toujours possible de gagner en commençant et en jouant une "partie parfaite", en pratique, nous ne jouerons pas "parfaitement" et les tests seront donc subjectifs. A noter donc qu'en commençant, en théorie, il est impossible que l'ordinateur nous batte à tous les coups.

La limite de temps étant un paramètre donnant des résultats variables d'une machine à une autre (et même pour une même machine sur plusieurs exécutions), nous utiliserons en premier lieu l'option -i afin de tester différentes limites pour le nombre d'itérations de l'algorithme.

Toutes les parties test sont jouées sans l'amélioration des simulations (-o 0) et avec le critère robuste.

Résultats des parties test (limite itérations) :

* Le joueur commence :
  + Avec 1000 itérations : victoire possible
  + Avec 5000 itérations : victoire possible
  + Avec 10000 itérations : non réussi
* L'ordinateur commence :
  + Avec 500 itérations : victoire possible
  + Avec 1000 itérations : non réussi

Avec l'option -v2, nous avons pu observer le temps mis par la machine pour ces différentes limites d'itérations, on a environ :

* 500 itérations : 0.01s et moins
* 1000 itérations : de 0.02s à 0.01s
* 5000 itérations : de 0.11s à 0.08s (à la fin)

Grâce à ces observations, nous pouvons maintenant avoir une idée des limites de temps à partir desquelles l'ordinateur nous bats à tous les coups et donc effectuer quelques tests sur des limites de temps (avec -t). Il est important de noter qu'en prenant les limites de temps supérieures ci-dessus, le nombre d'itérations en milieu et fin d'algorithme sera sensiblement supérieur aux limites d'itérations utilisées précédemment, ainsi, nous avons plutôt intérêt à prendre les limites de temps inférieurs pour réaliser nos tests.

Résultats des parties test (limite temps) :

* Le joueur commence :
  + Avec 0.08s : victoire possible
  + Avec 0.09s : victoire possible
  + Avec 0.1s : non réussi
* L'ordinateur commence :
  + Avec 0.01s : victoire possible
  + Avec 0.05s : victoire possible
  + Avec 0.07s : non réussi

En conclusion, il nous a été possible de battre l'ordinateur jusqu'à 5000 itérations lorsque le joueur commence et jusqu'à 500 itérations lorsque l'ordinateur commence.

Quant aux limites de temps sur la machine utilisée pour les tests, nous avons pu battre l'ordinateur jusqu'à 0.09s lorsque le joueur commence et jusqu'à 0.05s lorsque l'ordinateur commence.

Il est important de préciser à nouveau que ces tests sont entièrement subjectifs car nous ne jouons évidemment pas "parfaitement", si cela était le cas, nous pourrions gagner à tous les coups en commençant en premier. Il n'existe donc pas de réelle limite de temps à partir de laquelle l'ordinateur pourrait battre le joueur à tous les coups si c'est ce dernier qui commence.

## 2.3) Question 3

On peut tout d'abord noter que du fait de cette amélioration, l'estimation des probabilités de victoire de l'ordinateur (question 1) est biaisée car l'ordinateur choisit toujours les simulations où il peut gagner lorsque c'est possible et non plus au hasard.

En exécutant le programme avec l'amélioration (-o1) et sans (-o0) sur la limite par défaut de 5 secondes, on constate sur le premier coup que le nombre de simulations/itérations réalisées est plus élevé sans l'amélioration (sur notre test : 140 531 avec l'amélioration contre 234 583 sans).

En effet, le fait de chercher les coups gagnants durant les simulations prend plus de temps que de jouer un coup au hasard et on réalise par conséquent moins de simulations mais celles-ci sont plus précises et plus efficaces.

La qualité de jeu de la version avec l'amélioration des simulations est donc supérieur à la version sans du fait que les simulations apportent plus de précisions sur les possibilités de gagner dans un état particulier que les simulations entièrement aléatoires.

## 2.4) Question 4

L'option -O3 de gcc permet d'optimiser l'exécutable généré à la compilation afin d'améliorer sa vitesse d'exécution. Il existe différents niveaux d'optimisations et -O3 est le niveau le plus élevé hormis -Ofast qui va jusqu'à passer outre certaines conventions pour optimiser encore et d'autres options qui ont d'autres buts comme -Os qui réduit la taille du code.

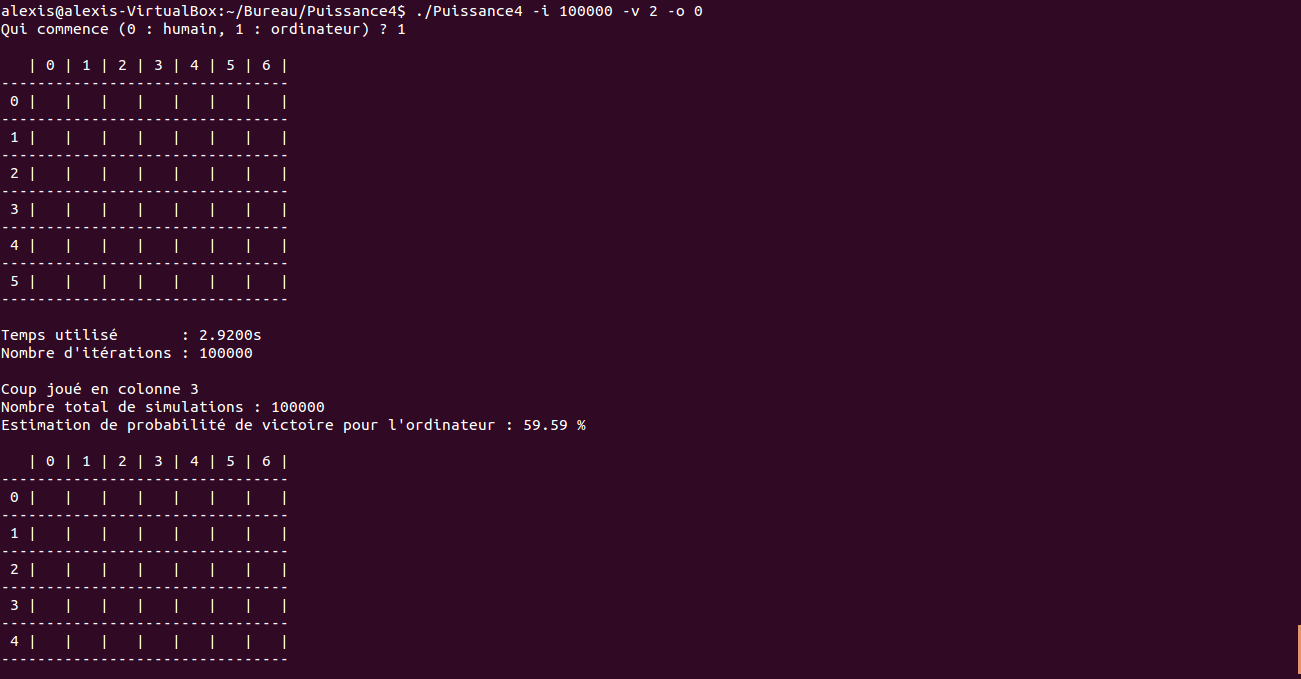
Parmi les optimisations alors mises en œuvre, il y a par exemple le fait que certaines variables soient placées dans des registres (à partir du premier niveau : -O ou -O1).

Afin de comparer ces différentes options, nous allons utiliser l'option -i qui va nous permettre de fixer un nombre d'itérations précis à l'algorithme et -v2 (verbosité au niveau 2) afin d'afficher le temps passé dans la boucle principale de l'algorithme, nous pourrons ainsi comparer les temps d'exécution et l'efficacité des options de compilation sur notre programme.

Pour les tests suivants, nous comparerons les temps d'exécution de l'algorithme sur 100 000 itérations, sur le premier coup (l'ordinateur commence). Nous utiliserons évidemment toujours les mêmes paramètres :

./Puissance4 -i 100000 -v 2 -o 0

On aura donc un affichage de la sorte (coup de l'ordinateur tronqué) :



Après plusieurs exécutions (une dizaine), on peut donner une moyenne grossière pour quelques options :

* Sans option : 2,795s
* -O1 : 1.834s
* -O2 : 1.541s
* -O3 : 1.521s
* -Ofast : 1.379s
* -Os : 1.548s

On constate donc que -Ofast donne effectivement les meilleurs performances pour notre programme. Si on ne le prend pas en compte du fait de son non-respect des standards, -O3 est bien l'option d'optimisation qui donne les meilleures résultats.

En conclusion pour cette question, le fait de rendre l'exécution de notre programme plus rapide permet évidemment d'augmenter le nombre d'itérations possibles en un temps donné et ainsi d'améliorer les performances de notre IA.

## 2.5) Question 5

Les critères max et robuste donnent assez rarement des coups différents du fait que le coup qui a été le plus visité et bien souvent aussi celui qui a la moyenne des récompenses la plus élevée.

Cependant, avec un faible nombre d'itérations, il y a plus de chance qu'une des moyennes des récompenses soit surévaluée ou sous-évaluée et que le nombre de simulations soient plus fiables rendant ainsi le critère robuste plus sûr dans ce genre de cas.

Au final, les deux critères se valent plus ou moins même si le critère robuste semble plus sûr dans certaines situations alors que le critère max semble quelques fois plus efficace (impression général tiré à force de jouer).

## 2.6) Question 6

Le Puissance 4 (classique) impliquant un facteur de branchement de 7 et une profondeur maximale de 6\*7 = 42, on aura donc un arbre complet composé de 742≈ 3\*1035 nœuds ce qui est vraiment conséquent.

Ainsi un algorithme Min-Max exhaustif (sans limitation de profondeur) n'est pas envisageable dans le cas du Puissance 4.