INTELLIGENCE ARTIFICIELLE

Monte Carlo Tree Search

Algorithmes de recherche heuristique

DEGUT Julian
GUYOT DE LA POMMERAYE Benjamin

Polytech 4ème année

[Lien du Github]

@jDegut

@LiakisGuyot

Table des matières

Introduction

Dans le cadre de notre 4ème année d'étude en ingénierie informatique, nous avons pour projet d'étudier le comportement de certains algorithmes de recherche heuristique. Notre objectif est d'implémenter les algorithmes Minimax, Alpha-beta et Monte Carlo Tree Search, avec le langage Java.

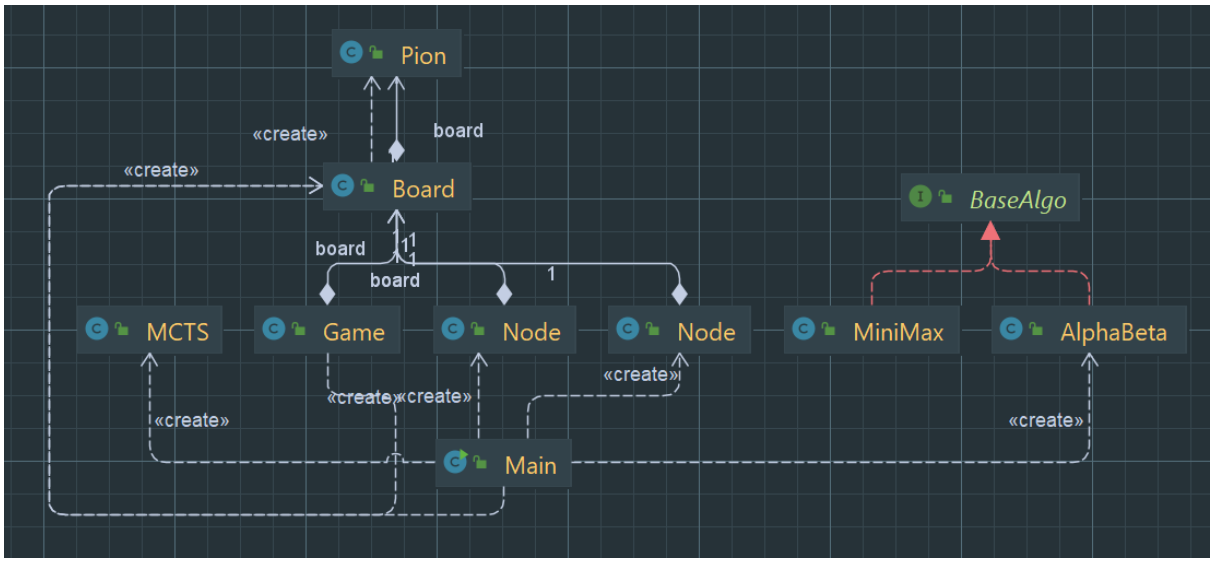
Pour implémenter chacun des algorithmes étudiés, nous allons définir comme contexte un jeu à deux joueurs où chaque joueur dispose d'une vision sur l'entièreté du jeu. Nous avons choisi de nous baser sur le jeu du Puissance 4. Étant simple à programmer et peu complexe, ce jeu nous permettra de nous concentrer sur les algorithmes sans avoir à se focaliser sur les règles du jeu.

De plus, l'aléatoire n'intervient d'aucune façon dans le jeu du Puissance 4. En effet l'ensemble des états du jeu résulte d'actions des joueurs et ne sont pas l'œuvre d'un facteur extérieur (comme dans le jeu de 2048 où l'aléatoire est responsable de la création des nouvelles cases).

Pour des raisons de simplicité d'écriture, nous utiliserons l'acronyme MCTS pour parler de l'algorithme Monte Carlo Tree Search.

Structure du code

Afin de mettre en place notre projet, nous avons développé un microprogramme comprenant le jeu et son affichage, les algorithmes, ainsi que les fonctions principales qui serviront pour nos expériences. La structure UML de notre programme se présente ainsi :



Nous avons choisi de répartir les classes en 3 packages différents :

- *Base* comprenant l'ensemble des classes pour l'implémentation des algorithmes qui serviront de « base » à nos expérimentations, à savoir *Minimax* et *AlphaBeta*. Chacun de ces algorithmes utilisent une structure de données *Node(base)* permettant de les stocker et les manipuler.

- *Mcts* composé de l'implémentation du Monte Carlo Tree Search et de ses composantes. Nous avons redéfini une structure *Node(mcts)* similaire à la précédente mais avec d'autres informations propres à cet algorithme.

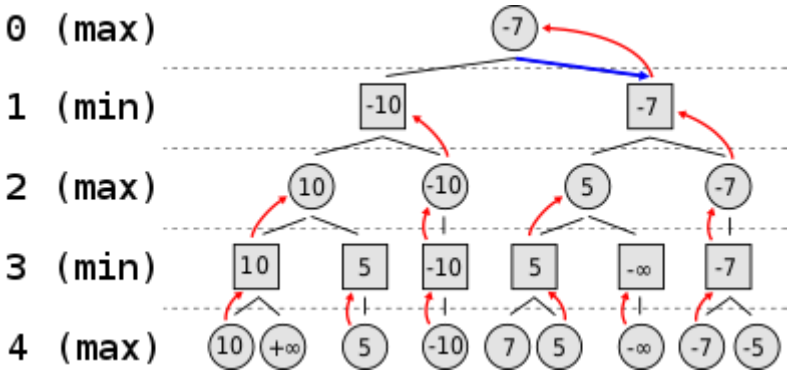
- *Game*, le dernier package dédié au fonctionnement du jeu, des pions et de son affichage (console). Cette partie n'étant pas notre objectif principal, nous n'avons pas voulu développer un affichage graphique du plateau en utilisant *JavaFX* ou *JPanels.*

Les algorithmes

L'objectif est de comparer les résultats d'un algorithme sur le Puissance 4 en fonction des paramètres mis en entrée. Nous pourrons ensuite poursuivre nos recherches sur un éventuel affrontement entre ces derniers.

Minimax

L'objectif de l'algorithme Minimax est de parcourir l'arbre de possibilité d'un jeu avec une profondeur définie, de placer une valeur, caractéristique à l'état du jeu, sur chacun des nœuds de ces branches. Chaque coup est ensuite choisi de donner le maximum de bénéfice au joueur 1, et le minimum au joueur 2.



Fonctionnement de Minimax (arbre), Wikipédia

L'implémentation de cet algorithme se base sur deux fonctions : la première maximisant le score du joueur à maximiser, et la deuxième minimisant l'adversaire.

```
function MINIMAX(racine, maxProfondeur)
    eval, action = JOUEURMAX(racine, maxProfondeur)
    return action
end function
```

```
function JOUEURMAX(n, p)
    if n est une feuille ou p = 0 then
        return EVAL(n), _null_
    end if
    u = −∞ et a = _null_
    for all f fils de n (obtenu par une action a_f) do
        eval, . = JOUEURMIN(f, p − 1)
        if eval > u then
            u = eval et a = a_f
        end if
    end for
    return u, a
end function
```

```
function JOUEURMIN(n, p)
    if n est une feuille ou p = 0 then
        return EVAL(n), _null_
    end if
    u = +∞ et a = _null_
    for all f fils de n (obtenu par une action a_f) do
        eval, . = JOUEURMAX(f, p − 1)
        if eval < u then
            u = eval et a = a_f
        end if
    end for
    return u, a
end function
```

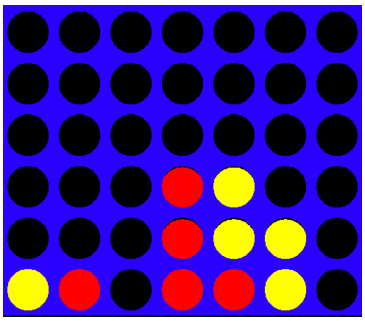
Pseudo-code de Minimax, Cours de M. Bonnevay

Enfin, l'évaluation d'un état du jeu passe par la disposition des pions présents sur le plateau. Nous définirons une fonction d'évaluation telle quelle :

- 4 pions du même joueur alignés (en ligne, colonne, ou diagonale) apportent 1000 points.
- 3 pions du même joueur et 1 case vide alignés apportent 50 points.
- 2 pions du même joueur et 2 cases vides alignés apportent 5 points.
- 1 pion du joueur et 3 cases vides alignés apportent 1 point.

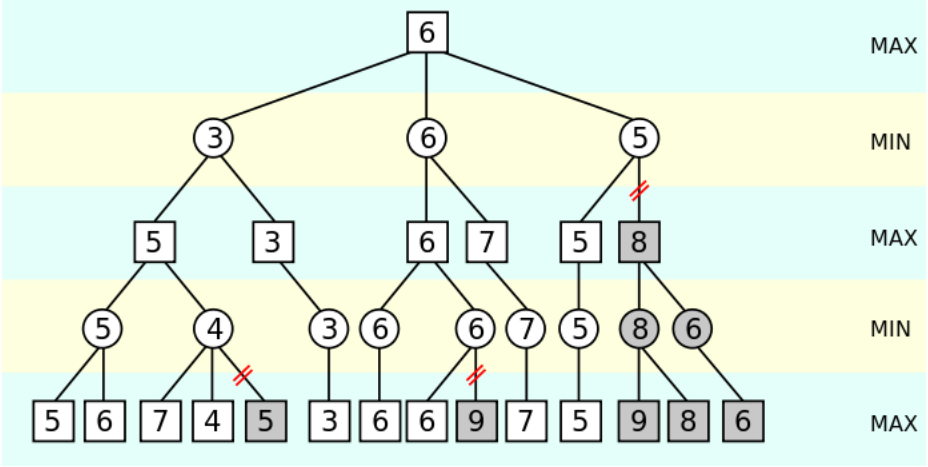
La somme des récompenses représentera le score d'un état du jeu.

Par exemple, un plateau présenté comme ceci (image) aura un score de 121 (2*50 + 3*5 + 6*1) pour le joueur rouge, et un score de 20 (3*5 + 5*1) pour le joueur jaune (le rouge domine).



AlphaBeta

L'algorithme AlphaBeta est le même que Minimax avec en plus une fonctionnalité d'élagage des branches lors de la recherche, le rendant donc plus rapide et performant. Cet élagage est rendu possible grâce au stockage d'une valeur maximale (α) et d'une valeur minimale (β). Ces valeurs permettent d'élaguer des branches, normalement visités par Minimax, inutiles dans la recherche de la meilleure possibilité.



Fonctionnement d'AlphaBeta (arbre), Wikipédia ; branches inexplorées en gris

L'implémentation de cet algorithme est semblable à celle de Minimax avec l'insertion de deux nouvelles variables *alpha* et *beta*.

```
function ALPHA-BETA(racine, maxProfondeur)
    eval, action = JOUEURMAX(racine, maxProfondeur, −∞, +∞)
    return action
end function
```

```
function JOUEURMAX(n, p, α, β)
    if n est une feuille ou p = 0 then
        return EVAL(n), _null_
    end if
    u = −∞ et a = _null_
    for all f fils de n (obtenu par une action a_f) do
        eval, . = JOUEURMIN(f, p − 1, α, β)
        if eval > u then
            u = eval et a = a_f
        end if
        if u ≥ β then
            return u, a
        end if
        α = max(α, u)
    end for
    return u, a
end function
```

```
function JOUEURMIN(n, p, α, β)
    if n est une feuille ou p = 0 then
        return EVAL(n), _null_
    end if
    u = +∞ et a = _null_
    for all f fils de n (obtenu par une action a_f) do
        eval, . = JOUEURMAX(f, p − 1, α, β)
        if eval < u then
            u = eval et a = a_f
        end if
        if u ≤ α then
            return u, a
        end if
        β = min(β, u)
    end for
    return u, a
end function
```

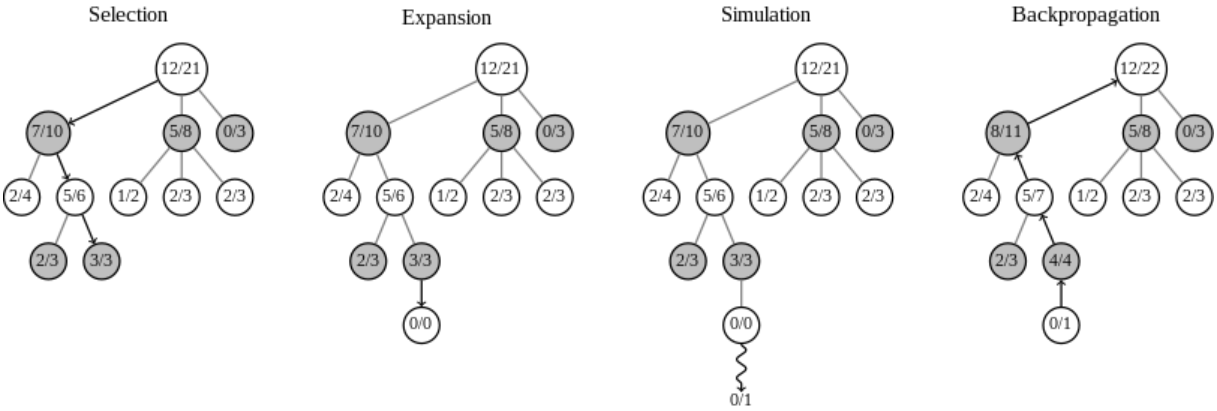
Pseudo-code d'AlphaBeta, Cours de M. Bonnevay

La fonction dévaluation de l'état de jeu est la même que celle utilisée pour Minimax.

Monte-Carlo Tree Search

Le MCTS est très différent des algorithmes précédemment étudiés puisque ce dernier ajoute une dimension probabiliste à la recherche du meilleur coup. L'objectif du MCTS est de sélectionner un nœud grâce à une fonction mathématique (Upper Confidence Bound), de l'étendre s'il n'est pas encore 100% étendu, puis de simuler une partie à partir de l'état représenté par le nœud.

Il n'est donc pas question de profondeur d'arbre ici, mais plutôt de « budget calculatoire ». Les machines d'aujourd'hui étant limité par leur capacité, nous définissons un nombre d'itérations maximal de processus de recherche (sélection, expansion, simulation, rétropropagation).



Fonctionnement du MCTS (arbre), Wikipédia

L'implémentation de cet algorithme s'effectue via de nombreuses fonctions :

- *uctSearch* permettant à partir d'un nœud de départ, et d'un nombre *maxIter* de lancer les processus de recherche.

- *treePolicy* qui se charge de vérifier si un nœud est étendu au maximum, et l'étend si ce n'est pas le cas

- *expand* dont le rôle est d'étendre le nœud mis en paramètre avec une action qui n'a pas encore été effectuée

- *bestChild* récupérant le meilleur nœud fils en appliquant la formule de sélection

- *defaultPolicy* lançant *n* simulations de parties à partir d'un état donné (dans notre cas : *n=10*)

- *backUp* qui remonte les nœuds afin de leur attribuer le score final et donc de mettre à jour les branches explorées.

3.3 Upper Confidence Bounds for Trees (UCT)

This section describes the most popular algorithm in the MCTS family, the *Upper Confidence Bound for Trees* (UCT)

function UCTSEARCH(s_0)
  create root node v_0 with state s_0
  while within computational budget **do**
    $v_l \leftarrow$ TREEPOLICY(v_0)
    $\Delta \leftarrow$ DEFAULTPOLICY($s(v_l)$)
    BACKUP(v_l, Δ)
  return a(BESTCHILD($v_0, 0$))

function TREEPOLICY(v)
  while v is nonterminal **do**
    if v not fully expanded **then**
      return EXPAND(v)
    else
      $v \leftarrow$ BESTCHILD(v, Cp)
  return v

function EXPAND(v)
  choose $a \in$ untried actions from $A(s(v))$
  add a new child v' to v
    with $s(v') = f(s(v), a)$
    and $a(v') = a$
  return v'

function BESTCHILD(v, c)
  return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$

function DEFAULTPOLICY(s)
  while s is non-terminal **do**
    choose $a \in A(s)$ uniformly at random
    $s \leftarrow f(s, a)$
  return reward for state s

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

Pseudo-code de MCTS, Cours de M. Bonnevay (& Thèse de M. Browne sur MCTS)

La fonction mathématique utilisée pour la sélection du nœud à explorer est la formule Upper Confidence Bound (UCB) qui s'exprime ainsi :

$$v^* = \underset{v \in children\ of\ v'}{\arg\max} \frac{Q(v')}{N(v')} + c * \sqrt{\frac{2\ln\ N(v)}{N(v')}}$$

Avec *Q(v)* représentant le score du nœud *v* et *N(v)* le nombre de visite du nœud *v*. La constance *c* sera ici définie sur $\sqrt{2}$.

Pour finir, nous redéfinissons l'évaluation d'une partie (les états évalués ne sont que des états finaux) :

- Le gain de la partie apporte 1 point.
- La perte de la partie enlève 1 point.

Expériences sur les algorithmes

Avant de comparer l'efficacité de chacun des algorithmes l'un contre l'autre, nous devons comparer celle résultant de leur paramètre respectif. Pour cela, nous avons implémenté des méthodes de test dans la classe *Main* permettant de lancer un nombre *n* de parties dont le joueur 1 est soit l'algorithme soit un joueur qui joue au hasard (nous étudierons si le choix du joueur 1 ou 2 possède un réel impact).

Efficacité de Minimax

Le seul paramètre de l'algorithme Minimax est la profondeur maximale de l'arbre exploré. En toute logique, plus cette profondeur est élevée, plus l'algorithme va pouvoir chercher de meilleure solution et donc retourner un coup plus « fiable » sur le long terme. Toutefois, nous posons l'hypothèse que le jeu du Puissance 4, n'étant pas assez complexe puisque à chaque tour le joueur possède 7 coups possibles, n'est peut-être pas le jeu qui illustrera le mieux l'impact de la profondeur.

Voici les résultats après expérience :

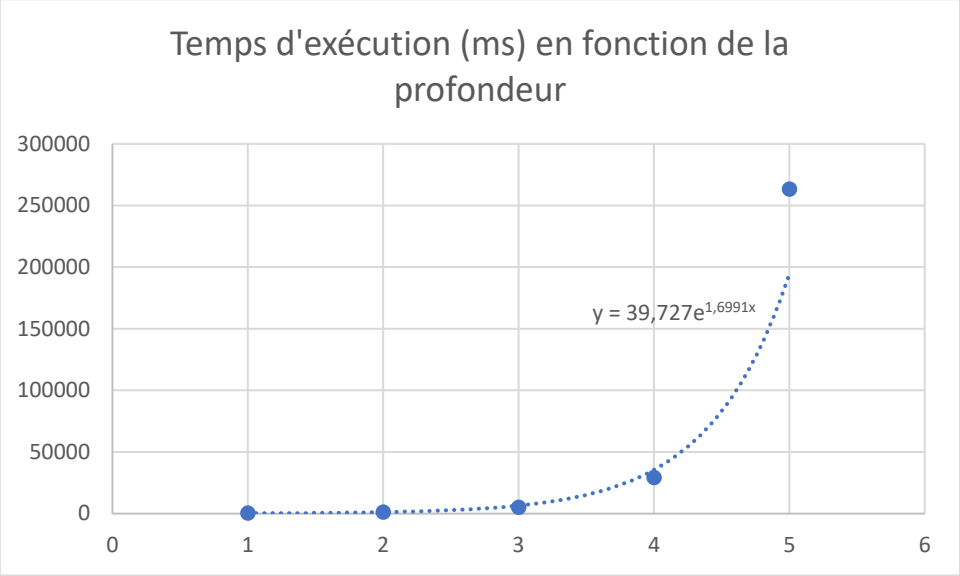
Profondeur maximale	Nombre de parties jouées	Temps d'exécution total	Nombre de nœuds visités (moyenne)	% parties gagnées
1	100	279 ms	30	100
2		1 087 ms	365	100
3		4 964 ms	2301	100
4		29 265 ms	13960	100
5		263 086 ms	101356	100

Résultats Minimax (joueur 1) vs Random

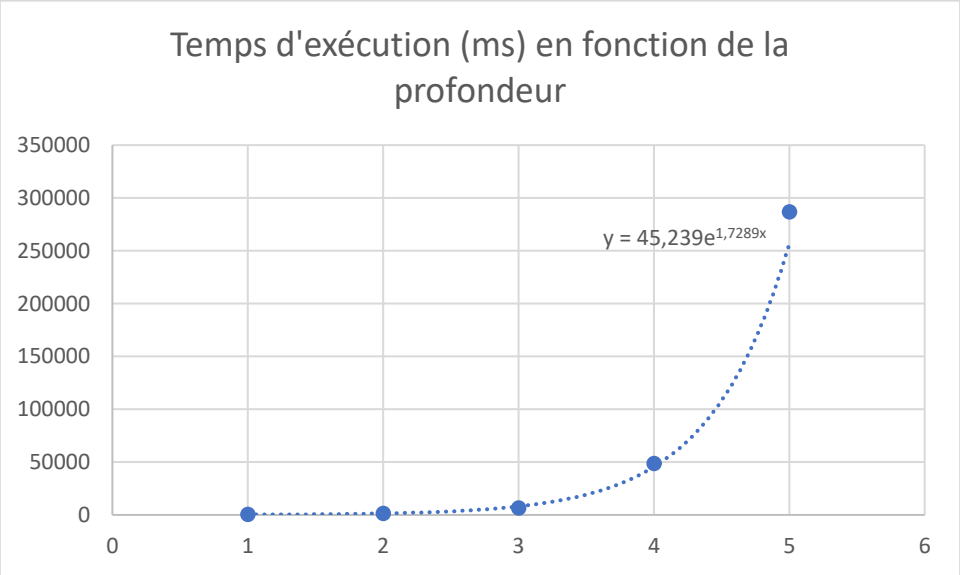
Profondeur maximale	Nombre de parties jouées	Temps d'exécution total	Nombre de nœuds visités (moyenne)	% parties gagnées
1	100	320 ms	36	98~100
2		1 208 ms	384	99~100
3		6 454 ms	2517	100
4		48 513 ms	18276	100
5		286 780 ms	115377	100

Résultats Random vs Minimax (joueur 2)

On remarque que pour la plupart des exécutions de l'algorithme, nous obtenons principalement 100% de parties gagnées, notre hypothèse est donc valide. Toutefois, la caractéristique intéressante ici est le temps d'exécution total dans chaque cas :



Graphique du temps d'exécution total en fonction de la profondeur choisie avec Minimax (joueur 1)



Graphique du temps d'exécution total en fonction de la profondeur choisie avec Minimax (joueur 2)

Nous n'étudions pas cet algorithme avec une profondeur supérieure ou égale à 6 à cause du budget calculatoire de ce dernier (malgré l'utilisation de Java).

Efficacité d'AlphaBeta

Tout comme Minimax, l'unique paramètre d'AlphaBeta est la profondeur d'exploration maximale. Cet algorithme est une copie conforme de Minimax avec l'insertion de nouvelles variables afin d'élaguer l'arbre de recherche. Nous posons donc l'hypothèse que nous aurons des résultats aussi bons que Minimax mais en un temps bien plus court.

Voici les résultats après expérience :

Profondeur maximale	Nombre de parties jouées	Temps d'exécution total	Nombre de nœuds visités (moyenne)	% parties gagnées
1	100	323 ms	31	100
2		757 ms	365	100
3		2 312 ms	1549	100
4		7 754 ms	7272	100
5		45 340 ms	30095	100
6		150 550 ms	139517	100

Résultats AlphaBeta (joueur 1) vs Random

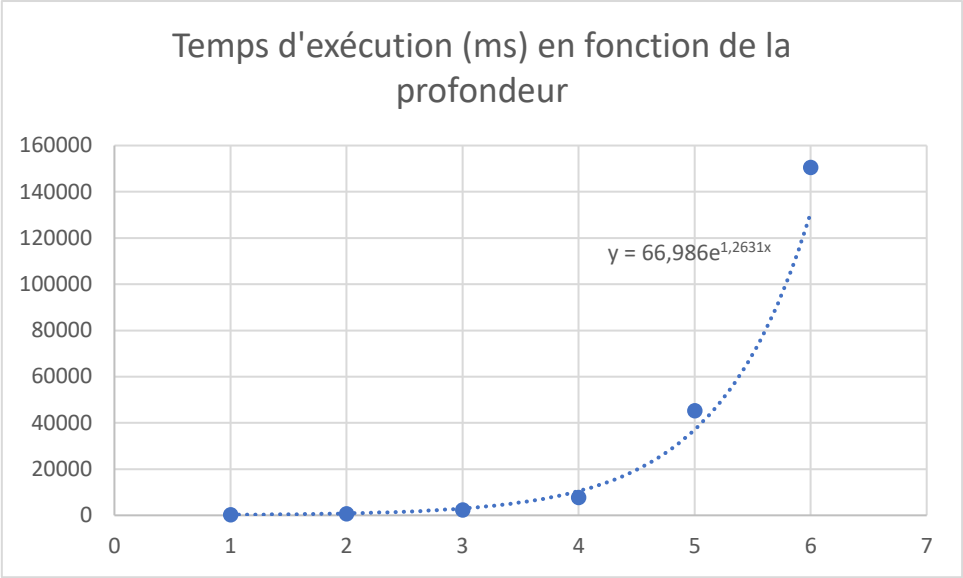
Profondeur maximale	Nombre de parties jouées	Temps d'exécution total	Nombre de nœuds visités (moyenne)	% parties gagnées
1	100	320 ms	38	98~100
2		889 ms	384	99~100
3		3 100 ms	1743	100
4		12 567 ms	9004	100
5		55 906 ms	37153	100
6		205 741 ms	164243	100

Résultats Random vs AlphaBeta (joueur 2)

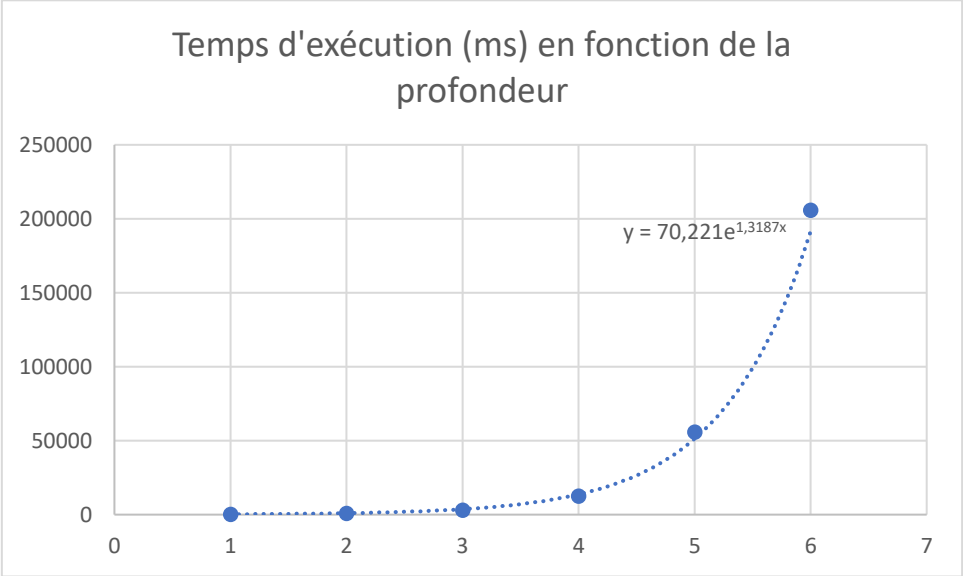
Le temps est amplement réduit en comparaison avec Minimax. Le nombre de parties gagnées lui n'a pas bougé, comme précédemment hypothétisé.

De plus on voit clairement une diminution du nombre moyen de nœuds visités (baisse de 40% allant jusqu'à 80%).

Les graphiques suivants montrent l'évolution du temps d'exécution total (ms) en fonction de la profondeur maximale :



Graphique du temps d'exécution total en fonction de la profondeur choisie avec AlphaBeta (joueur 1)

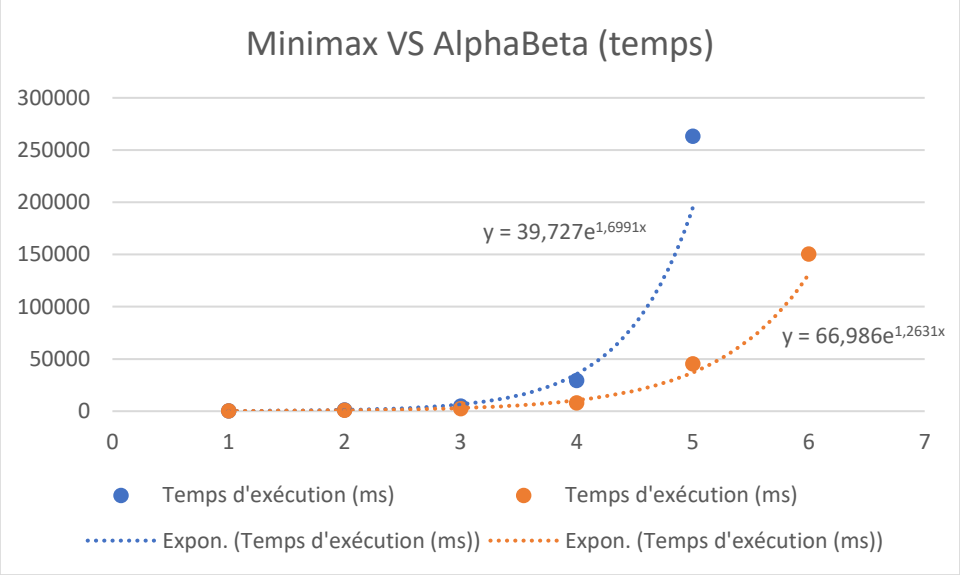


Graphique du temps d'exécution total en fonction de la profondeur choisie avec AlphaBeta (joueur 2)

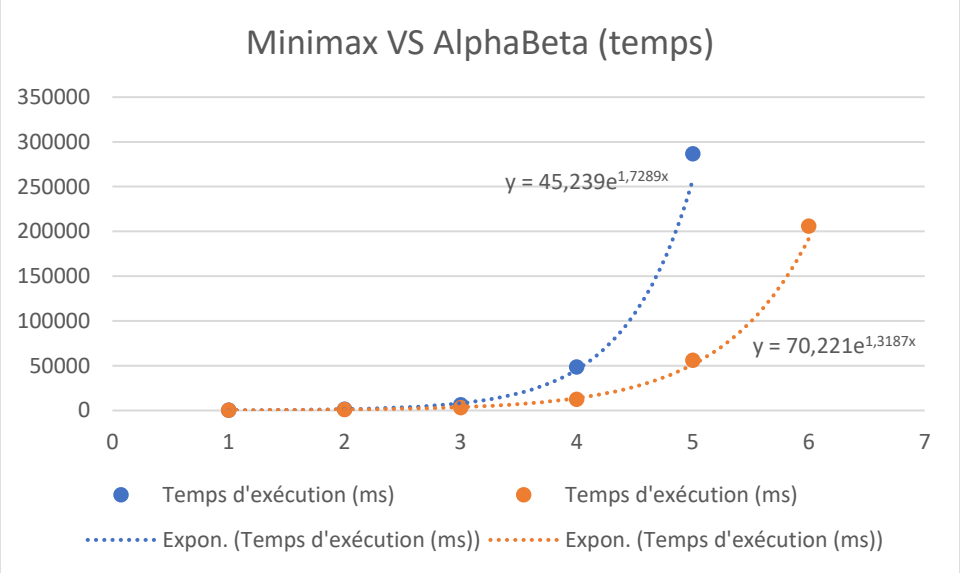
Nous n'étudions pas cet algorithme avec une profondeur supérieure ou égale à 7 à cause du budget calculatoire de ce dernier (malgré l'utilisation de Java).

Minimax contre AlphaBeta : le plus rapide

Nous pouvons maintenant comparer les vitesses d'exécutions des deux algorithmes sur les précédentes expériences. Pour cela, superposons les graphiques :



Comparaison du temps d'exécution total en fonction de la profondeur choisie entre Minimax et AlphaBeta (joueur 1)



Comparaison du temps d'exécution total en fonction de la profondeur choisie entre Minimax et AlphaBeta (joueur 2)

On voit clairement l'amélioration de la complexité temporelle de l'algorithme AlphaBeta sur Minimax. L'efficacité étant la mesure visant à obtenir le meilleur score le plus rapidement possible (avec le moins de coût calculatoire), nous en déduisons que l'algorithme AlphaBeta est plus efficace que Minimax.

Efficacité de MCTS

L'algorithme MCTS se présente différemment des deux précédents algorithmes. Pour fonctionner il a besoin de connaître le nombre maximal de recherche à effectuer dans l'arbre. Nous allons donc tester cet algorithme en fonction de son paramètre contre un joueur dont la stratégie est de jouer au hasard.

Voici les résultats après expérience :

Nombre d'itérations	Nombre de parties jouées	Temps d'exécution total	Nombre de niveaux (moyen)	Profondeur maximale	% parties gagnées
10	100	475 ms	3	7	84
100		2 743 ms	6	10	93
1000		20 860 ms	13	16	98
10000		180 896 ms	19	24	96

Résultats MCTS (joueur 1) vs Random

Nombre d'itérations	Nombre de parties jouées	Temps d'exécution total	Nombre de niveaux (moyen)	Profondeur maximale	% parties gagnées
10	100	497 ms	3	6	86
100		2 843 ms	7	12	95
1000		20 896 ms	15	20	96
10000		174 244 ms	20	26	93

Résultats Random vs MCTS (joueur 2)

Nous pouvons observer un pourcentage de partie gagnées qui varie beaucoup en fonction du budget calculatoire. Dans un premier temps, il a tendance à augmenter si le budget augmente ; puis redescend pour un nombre d'itérations égal à 10 000. Cela peut être dû au fait que le puissance 4 n'a pas besoin d'avoir une recherche aussi poussée et profonde par le peu de possibilités de coups offert aux joueurs à chaque tour.

Entre l'algorithme AlphaBeta et MCTS, la différence est la construction et le parcours de l'arbre des états du jeu. Le MCTS vient parcourir un large panel de solutions en exploitant les différentes options à chaque état. Toutefois, il y a une part de hasard, notamment dans la simulation des parties pour l'attribution du score d'un nouveau nœud.

Cette différence de stratégie vient impacter le pourcentage de gains puisque contrairement au MCTS, AlphaBeta lui va explorer de manière plus complète et moins hasardeuse les solutions envisageables à partir de l'état actuel.

MCTS contre AlphaBeta

AlphaBeta est similaire à Minimax : il permet d'en résulter les mêmes solutions optimales dans un temps plus courts puisqu'il n'explore simplement pas toutes les solutions envisageables. Il n'est donc pas nécessaire de comparer MCTS avec Minimax.

Comparer deux algorithmes très différents est souvent difficile. Tout d'abord, il faut savoir que le jeu du puissance 4 n'est peut-être pas le contexte le plus recherché pour avoir une comparaison fiable entre ces algorithmes. Chacun des algorithmes ont leurs avantages et leurs inconvénients, toutefois, dans notre étude, nous voulions savoir lequel des deux en sortirait vainqueur.

Voici les résultats après expérience :

Profondeur maximale (AB)	**Nombre d'itérations**	**Nombre de parties jouées**	**Temps d'exécution total**	**AB/MCTS (gains)**
1	10	100	630 ms	96/4
2			1 612 ms	94/6
3			4 276 ms	99/1
4			14 487 ms	98/2
5			75 857 ms	98/2
6			217 136 ms	100/0
1	100		2 953 ms	87/13
2			4 654 ms	83/17
3			9 074 ms	87/13
4			22 670 ms	89/11
5			88 478ms	92/8
6			282 294 ms	98/2
1	1000		18 917 ms	87/13
2			21 439 ms	89/11
3			27 834 ms	79/21
4			40 022 ms	96/4
5			112 350 ms	91/9
6			296 147 ms	97/3
1	10000		137 291 ms	87/13
2			160 978 ms	94/6
3			179 279 ms	81/19
4			187 340 ms	96/4
5			243 861 ms	90/10
6			421 434 ms	97/3

Résultats AlphaBeta (joueur 1) vs MCTS (joueur 2)

Profondeur maximale (AB)	Nombre d'itérations	Nombre de parties jouées	Temps d'exécution total	MCTS/AB (gains)
1	10	100	718 ms	7/93
2			1 732 ms	7/93
3			4 428 ms	1/99
4			17 785 ms	1/99
5			73 685 ms	2/98
6			324 816 ms	3/97
1	100		3 400 ms	18/82
2			5 009 ms	14/86
3			8 141 ms	7/93
4			28 234 ms	13/87
5			88 852 ms	1/99
6			397 178 ms	11/89
1	1000		23 995 ms	26/74
2			25 126 ms	13/87
3			26 151 ms	4/96
4			49 784 ms	19/81
5			100 134 ms	1/99
6			407 142 ms	11/89
1	10000		182 446 ms	24/76
2			184 192 ms	10/90
3			171 298 ms	3/97
4			203 165 ms	12/88
5			208 724 ms	3/97
6			556 582 ms	6/94

Résultats MCTS (joueur 1) vs AlphaBeta (joueur 2)

Dans l'intégralité des cas, on peut observer une meilleure performance de la part d'AlphaBeta que de MCTS sur les parties de Puissance 4. On en déduit qu'un algorithme comme AlphaBeta convient plus sur un contexte de jeu comme celui du Puissance 4 que MCTS : l'algorithme MCTS est un algorithme probabiliste qui donnera un résultat différent à chaque recherche. A l'inverse AlphaBeta lui est déterministe et donnera toujours la même recherche résultant d'un état.

C'est pour cette raison qu'il n'est d'ailleurs pas utile de faire jouer AlphaBeta contre lui-même ou bien contre Minimax puisqu'ils auront exactement la même stratégie de jeu.

Jouer en premier : l'impact significatif

Dans notre étude, nous avons pris soin d'effectuer nos expériences en échangeant l'ordre du joueur 1 et du joueur adversaire (2). Nous voulons maintenant savoir si jouer en premier possède un impact significatif sur le jeu.

Pour cela, nous allons simplement faire le taux du cumul des parties gagnées par le MCTS contre AlphaBeta lorsqu'il commençait, sur celui des parties gagnées lorsqu'il était joueur 2 :

$$T = \frac{\sum parties\ gagnées(j1)}{\sum parties\ gagnées(j2)} = \frac{217}{195} \approx 1{,}113$$

On obtient donc un taux supérieur à 1 signifiant que MCTS a plus gagné en tant que joueur 1 que joueur 2 (11,3% de victoires en plus).

> *« En 1988, James D. Allen et Victor Allis ont publié leurs résultats à la suite d'une expérience scientifique à propos du Puissance 4 : le jeu est résolu. En effet, le joueur qui commence peut assurer sa victoire à 100% s'il joue les coups adéquats », Wikipédia*

Les travaux de M. D. Allen et M. Allis appuie nos résultats : si le MCTS commence la partie, il pourra en toute logique mieux s'en sortir que s'il ne commençait pas. Nous en déduisons donc que, non seulement jouer en premier possède un impact significatif sur le déroulement d'une partie, mais également sur l'intégralité des parties si les coups sont joués intelligemment.

Conclusion

Pour conclure, les algorithmes Minimax et AlphaBeta étant des algorithmes déterministes sont plus à même de trouver les coups optimaux dans un contexte de Puissance 4. A l'inverse, l'algorithme MCTS, aujourd'hui très reconnu pour son efficacité dans de nombreux domaine, est un algorithme probabiliste qui ne pourra rarement donner de meilleurs résultats. Toutefois, chacun de ces algorithmes possèdent leurs avantages et inconvénients. Ils proposeront une meilleure efficacité dans un domaine plutôt qu'un autre.

Dans le contexte de notre étude, les algorithmes déterministes se montrent meilleurs que ceux probabilistes. Cela peut être dû au faible choix proposé au joueur à chaque tour, et donc à une dimension d'arbre de jeu relativement faible (en comparaison avec le jeu des échecs par exemple).

Sources

- Puissance 4, Wikipédia
- Algorithme Minimax, Wikipédia
- Elagage AlphaBeta, Wikipédia
- Monte Carlo Tree Search, Wikipédia
- A survey of Monte Carlo Tree Search methods, Cameron Browne
- Pseudocode Minimax, AlphaBeta et MCTS, Cours de M. Bonnevay