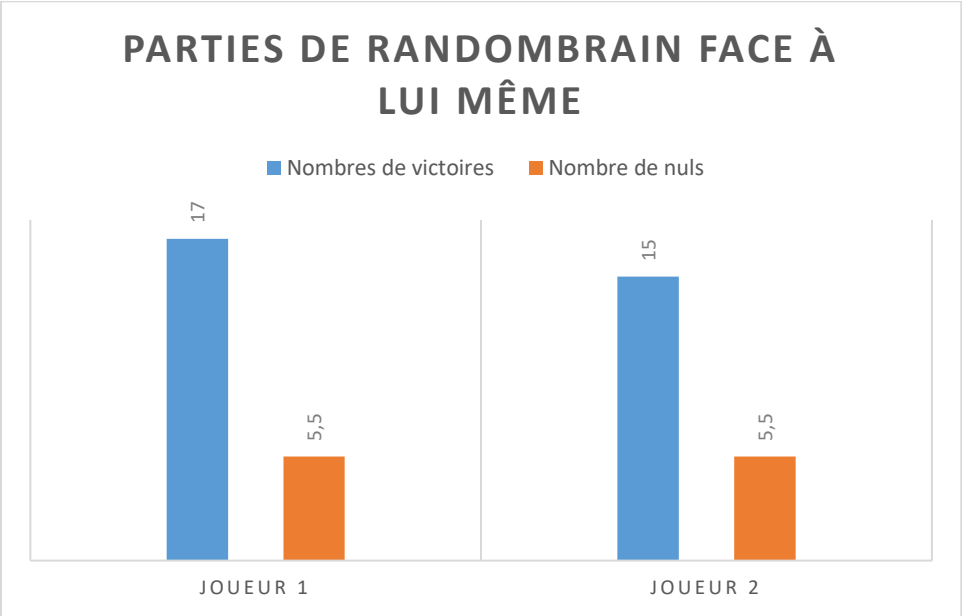
In104 - Checkers

Vincent Navarro et Philippe Dufour

Pour prendre en main le fonctionnement du simulateur de parties, nous avons tout d'abord codé un cerveau qui jouait des coups aléatoires parmi ceux proposés par le gamestate.

Celui-ci ne semblait pas spécialement performant. Quand on le fait se rencontrer face à lui-même, on obtient le graphe suivant pour un total de 43 parties.



Ainsi le développement de notre IA ne débute qu'avec le codage du Minimax. Cet algorithme peut être utilisé pour jouer aux dames car il n'y a que deux joueurs, que le jeu est déterministe (ie le hasard n'intervient jamais) et que l'information est parfaite (la position des pièces de l'adversaire à chaque instant est connue par exemple).

Le principe du minimax est de choisir, pour une profondeur fixe de recherche de coup, le coup qui maximise le score du joueur et minimise celui de l'adversaire. Ainsi la partie s'assimilant à un arbre, un algorithme récursif est bien adapté. La profondeur de recherche maximale diminuant d'un cran à chaque itération, le critère d'arrêt est atteint lorsque cette dernière est nulle.

Toutefois cet algorithme a quelques défauts. D'une part, il faut fixer à l'avance une profondeur maximale d'exploration, ce qui peut entraîner un dépassement du temps alloué pour jouer, et d'autre part le fait qu'il suppose que l'adversaire joue de manière optimale.

Pour cela, nous avons pensé à plusieurs fonctions d'évaluations possibles :

- Une évaluation basée sur la différence entre le nombre de pions restants pour chacun des joueurs.
- En accordant de l'importance à la présence de certaines pièces. Par exemple, plus le nombre de dames est élevé, plus la position est favorable.

- En accordant de l'importance à certaines positions. Par exemple, un pion proche de la dernière ligne adverse peut devenir une dame, ou encore certaines formation de pions sont plus avantageuses que d'autres.

On voulait combiner toutes ses évaluations, et les pondérer automatiquement en faisant jouer l'algorithme contre lui-même, pour obtenir le meilleur équilibre.

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1526597	5.136	3.364e-06	5.136	3.364e-06	evaluation.py:2(evaluate)

On voit ci-dessus que la fonction d'évaluation est appelée 1.52 millions de fois. Ainsi elle doit être la plus rapide possible puisque c'est la principale responsable du temps de calcul du programme. C'est pourquoi nous avons décidé de choisir la fonction d'évaluation basée sur la différence de nombre de pions.

En faisant se rencontrer MinimaxBrain face à RandomBrain, l'algorithme minimax gagne systématiquement pour une profondeur maximale de recherche de 5 coups.

	Partie n°	1	2	3	4	5	6	7	8	9	10
Temps utilisé pour jouer (en s)	MinimaxBrain	4,1	11,8	3,4	1,6	4,3	5,1	3	8,6	3,9	11,7
	RandomBrain	0,0025	0,004	0,0023	0,0025	0,0031	0,0028	0,0019	0,0025	0,0036	0,0032

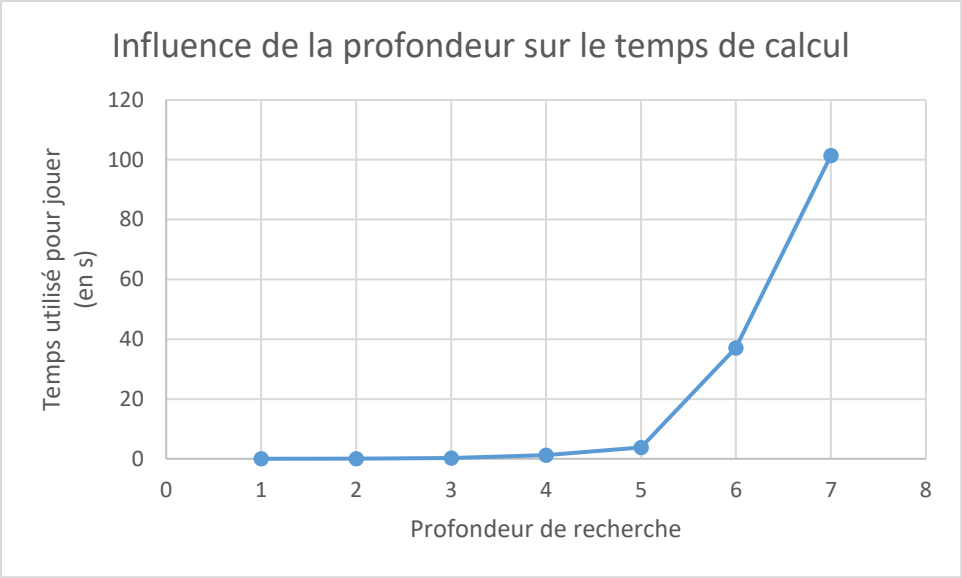
Par contre, le minimax met beaucoup plus de temps à jouer : en moyenne 5.75s contre 0.0028s pour le cerveau aléatoire.

Nous nous sommes rendus compte que certaines parties du code étaient redondantes et donc inefficaces. Par exemple ci-dessous, le minimax est calculé deux fois alors qu'une seule fois est nécessaire.

```
    goodState=gameState.findNextStates()[0]
    for state in gameState.findNextStates():
        if s<minimax(state,self.depth,True):
            s=minimax(state,self.depth,True)
            goodState=state
    return goodState
```

```
    for state in gameState.findNextStates():
        a=minimax(state,self.depth,True)
        if s<a:
            s=a
            goodState=state
    return goodState
```

On peut ensuite analyser l'influence de la profondeur maximale de recherche sur le temps que mets le programme à jouer.



A partir d'une profondeur égale à 8, le minimax dépasse la limite des 10s pour un coup. On comprend alors que la profondeur de recherche a une grande importance sur la qualité du minimax. Le graphe ci-dessus justifie notre choix d'une profondeur de recherche maximale de 5. En effet, pour celle-ci nous avons le meilleur compromis temps de calcul/profondeur.

Ensuite pour améliorer encore sa rapidité, nous avons ajouté des tables de transpositions au minimax. L'idée consiste à garder en mémoire des positions qui ont déjà été analysées, ainsi que leur score. Pour cela, on utilise un dictionnaire dans lequel on stocke les états et leur score. A chaque itération, on vérifie si la position n'est pas déjà dans le dictionnaire. Si c'est le cas, on récupère son score sinon on l'ajoute au dictionnaire.

On fait alors s'affronter cette nouvelle version du minimax avec transposition face au cerveau aléatoire.

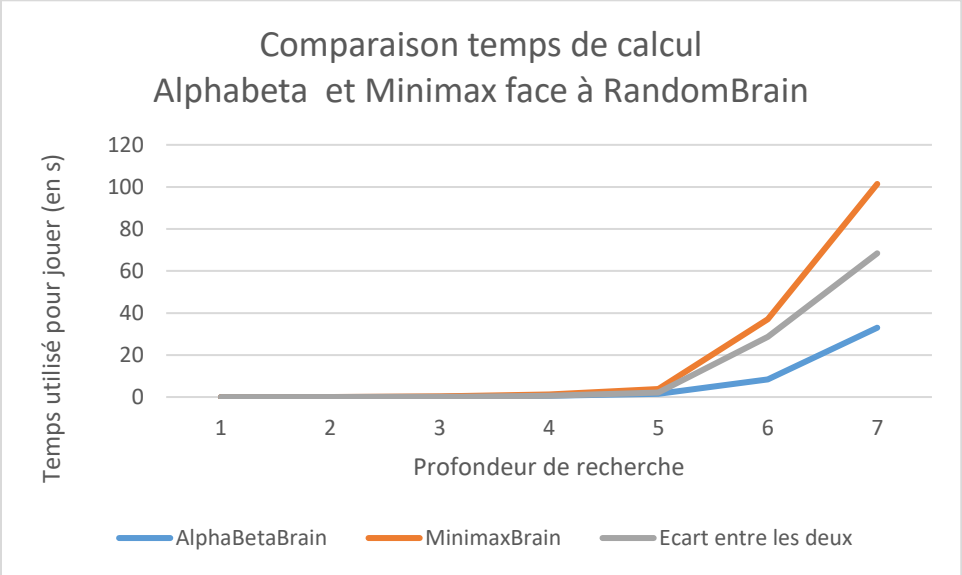
	Partie n°	1	2	3	4	5	6	7	8	9	10	En moyenne
Temps utilisé pour jouer (en s)	MinimaxBrain	4,1	11,8	3,4	1,6	4,3	5,1	3,0	8,6	3,9	11,7	4.93
	MinimaxTranspositionBrain	4.5	2.2	4.26	5.6	6.9	6.4	4.4	2.3	2.0	6.3	4.48

Désormais le minimax (avec transposition) toujours avec une profondeur de 5 prend en moyenne 4.48 s pour jouer, soit une amélioration de 9.1% du temps de calcul.

Enfin nous avons codé l'élagage alpha-beta pour continuer à augmenter la compétitivité de notre IA.

L'élagage alpha-beta consiste à ne calculer que les sous-arbres qui sont nécessaires. En effet, si la valeur courante d'un fils d'un nœud min est supérieure aux autres valeurs de ce nœud fils, alors l'algorithme ne l'étudiera pas.

On mesure le temps de calcul pour AlphabetaBrain lorsqu'il rencontre RandomBrain.



On voit que celui est bien meilleur que le simple minimax, surtout lorsque la profondeur augmente.

Par exemple, pour une profondeur de recherche de 6, AlphaBetaBrain met 4,4 fois moins de temps que MinimaxBrain.

On remarque qu'avec l'élagage alpha-beta la fonction d'évaluation est trois fois moins appelée. Ce qui explique le gain de temps par rapport au minimax simple.

ncalls	tottime	percall	cumtime	percall	filename:lineno(functio
416920	1.639	3.931e-06	1.639	3.931e-06	evaluation.py:2(evaluate)

Quand on fait une partie entre MinimaxTranspositionBrain et MinimaxAlphabetaBrain pour la même profondeur de recherche, on obtient des matchs nuls à chaque fois. Mais l'algorithme Alphabeta devient légèrement plus rapide quand la profondeur augmente.

Profondeur	MinimaxAlphaBetaBrain	MinimaxTranspositionBrain
1	0,0079	0,0077
2	0,029	0,027
3	0,12	0,13
4	0,54	1
5	3,6	3,4
6	5,3	5,6
7	53,7	57,3

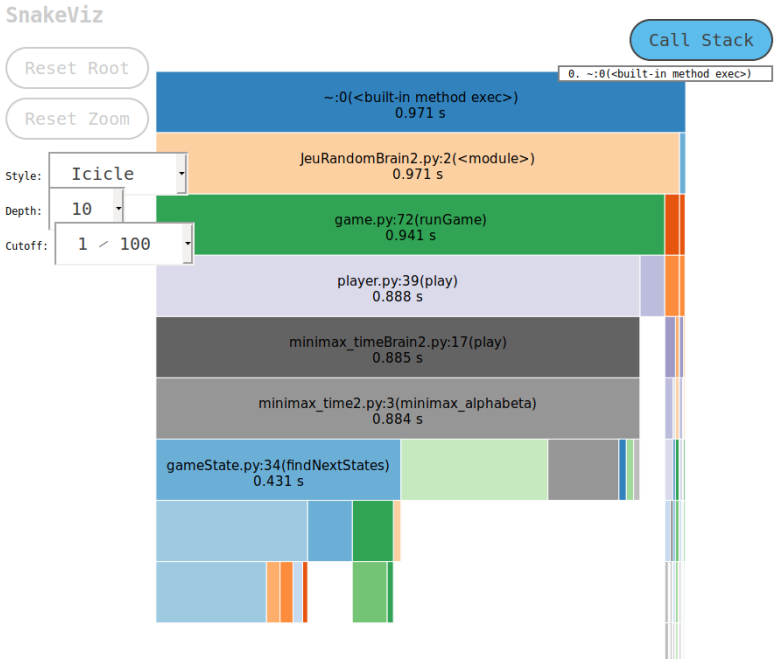
On se propose alors de combiner les deux (élagage alpha-beta et transposition) pour essayer d'obtenir le meilleur de chaque programme. On écrit alors un programme AlphabetaTranspositionBrain.

Profondeur	AlphaBetaTranspositionBrain
1	0,03
2	0,079
3	0,21
4	0,8
5	4,1
6	36,4
7	38,3

Celui-ci se révèle moins efficace qu' AlphaBeta seul. Nous gardons donc seulement l'élagage pour la suite.

Nous avons codé un minimax avec élagage alpha-beta qui prend en compte le temps d'exécution pour optimiser la profondeur de recherche et s'assurer que le programme ne dépasse pas le temps imparti. Ceci a l'avantage de ne plus spécifier la profondeur de recherche, ce qui permet de calculer le maximum de coups dans les limites temporelles imparties. Celui-ci utilise en moyenne la moitié du temps disponible pour jouer ses coups (par exemple, il utilise 0.05s pour un temps limite de 0.1s par coup).

Le profiler nous a notamment permis de déterminer le temps nécessaire à calculer les états fils du nœud examiné et donc d'optimiser la condition de temps intervenant dans le minimax_time.



Finalement nous avons modifié notre fonction d'évaluation pour prendre en compte aussi la création et la présence de dames. Ainsi lors du calcul du score d'un nœud, on compte la différence du nombre de pions et le nombre de dames : un pion valant un et une dame trois. Après l'avoir testé, celle –ci ne se révèle pas particulièrement plus efficace. Pour ce test, nous avons fait une série de parties entre le minimax_time doté de la fonction d'évaluation simple et celui doté de la nouvelle fonction d'évaluation. Même en augmentant le nombre de coups avant une partie nulle, nous obtenons à chaque fois un match nul. On suppose toutefois que la fonction d'évaluation avec pris en compte des dames est plus performante, c'est donc elle que nous utilisons dans notre IA finale.

ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
43663	0.258	5.91e-06	0.258	5.91e-06	evaluation2.py:2(evaluate2)

Sur la capture d'écran ci-dessus, on remarque que le nombre d'appels à la fonction d'évaluation a encore été divisé par 10 par rapport à l'algorithme alphabeta. En effet, le fait d'utiliser une profondeur variable permet d'optimiser l'utilisation de la fonction d'évaluation. Il faut toutefois que la fonction d'évaluation reste appelée un nombre suffisant de fois pour être sûr que le maximum de coups possibles est recherché.

En conclusion, au fur et à mesure du projet, nous avons considérablement amélioré la rapidité et l'efficacité de notre IA par rapport à la première version du minimax. Toutefois nous avons aussi rencontré des difficultés avec la gestion du git et l'installation du profiler.

Notre IA est désormais prête pour la compétition !