Rapport Checkers

I. Introduction

Tout d'abord, nous avons pris connaissance du contenu du package « IN104_simulateur » permettant de simuler une partie de jeu de dames en gérant son déroulement ainsi que la validité des coups entrepris par les joueurs. Nous avons de ce fait lu la présentation détaillée de chacune des classes du package afin de bien maîtriser les différents éléments que nous allions manier au cours du projet. Nous n'avons pas pu lors de la première séance tester le fonctionnement du simulateur en se plaçant dans le cas d'une partie de dames opposants deux joueurs. En effet, les deux ordinateurs de la salle informatique ne fonctionnaient pas, c'est pourquoi nous avons travaillé sur nos ordinateurs personnels qui, étant sous Windows et non pas Linux, ne nous ont pas permis de pouvoir utiliser le simulateur. Ceci nous a fait perdre un temps précieux et nous avons au fur et à mesure, à chaque séance, rattrapé petit à petit ce retard initial. N'ayant plus eu de problèmes avec les ordinateurs de la salle informatique par la suite, nous avons pu enfin tester le simulateur et effectuer une partie entre deux « RandomBrain », que nous avions implémentés auparavant. Nous avons constaté que les deux IA jouaient très rapidement (avec une moyenne de 0.0006378 secondes sur 10 parties) mais les coups étant choisis au hasard, ces derniers n'étaient pas les meilleurs. Nous obtenions logiquement des résultats de match aléatoires entre ces deux IA.

II. Minimax basique

Nous avons ensuite étudié le principe de l'algorithme « minimax ». Dans un premier temps, nous avons tous deux tenté d'implémenter cet algorithme dans le cas d'une partie de dames en réfléchissant chacun de notre côté. Nous avons ensuite mis en commun nos idées et nos ébauches de code pour réaliser notre algorithme de minimax. Ce dernier procède de la façon suivante : il prend en argument l'état dont on cherche le score, la profondeur à laquelle on veut sonder et un argument « maximize », booléen qui indique si l'on maximise où minimise le score. Deux cas sont alors envisagés : soit la profondeur de l'état dans lequel on se trouve est nulle et il faut alors l'évaluer via la fonction d'évaluation que l'on a implémentée (on ne veut pas explorer plus profond), ou bien la profondeur est non nulle et il s'agit dans ce cas de descendre tout en bas de la branche en cours d'exploration de l'arbre (on parcourt en profondeur l'arbre des différents états accessibles à chaque tour) et d'évaluer l'état associé à la feuille située au bout de la branche. En effet, l'algorithme minimax est récursif: on calcule via le minimax les scores des nœuds enfants en changeant le booléen « maximize » (car c'est à l'autre joueur de jouer à l'étage des enfants) et en décrémentant la profondeur de 1 (on se situe un cran en dessous donc on cherche à plonger un cran moins profond). Si l'état n'a pas d'enfant, c'est une feuille et on récupère sa valeur grâce à la fonction d'évaluation. Une fois les scores des enfants calculés, on en prend le maximum ou le minimum suivant la valeur de « maximize », pour obtenir la valeur du nœud parent.

III. Minimax avec table de transposition

L'un de nous a implémenté une fonction d'évaluation basique, qui renvoie pour un état donné la différence entre le nombre de pièces du joueur blanc et du joueur noir, et le « MinimaxBrain », en s'appuyant sur l'exemple donné dans l'énoncé de la séance 2. Pendant ce temps, l'autre a implémenté une table de transposition au sein de notre algorithme de minimax. Nous avons testé l'efficacité de la table de transposition en comparant les temps avec et sans table de transposition. Nous avons alors joué les matchs suivants :

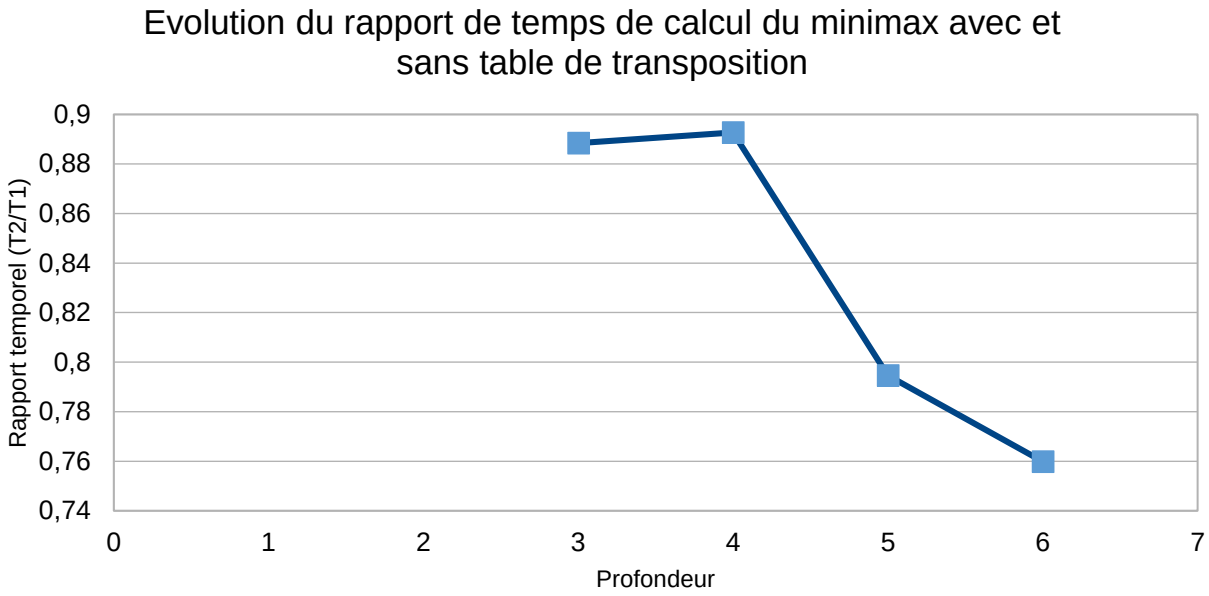
Minimax1 (en blanc) vs Minimax2 (en noir)

Minimax1 (en blanc) vs Minimax avec Transposition (en noir)

Ainsi Minimax2 et Minimax avec transposition sont dans les mêmes conditions de jeu et on peut comparer de manière pertinente leurs temps de calculs.

On peut par exemple tracer le rapport des temps de calculs en fonction de la profondeur utilisée (la même pour Minimax et Minimax avec Transposition) :

Profondeur d'exploration	Temps de Minimax2 = T1 (en secondes)	Temps de Minimax avec Transposition = T2 (en secondes)
3	0.7881	0.7002
4	2.1821	1.9480
5	53.9110	42.8395
6	34.3908	26.1325



On s'aperçoit alors que la table de transposition est d'autant plus efficace que la profondeur d'exploration est grande (cela s'explique simplement par le fait que plus la profondeur est grande, plus le nombre d'état évalué est grand, et donc la table de transposition est beaucoup plus remplie, ce qui permet en l'interrogeant de gagner du temps). Tous les matchs dont les résultats sont présentés ci-dessus se sont soldés par une égalité.

IV. Minimax avec élagage alphabeta

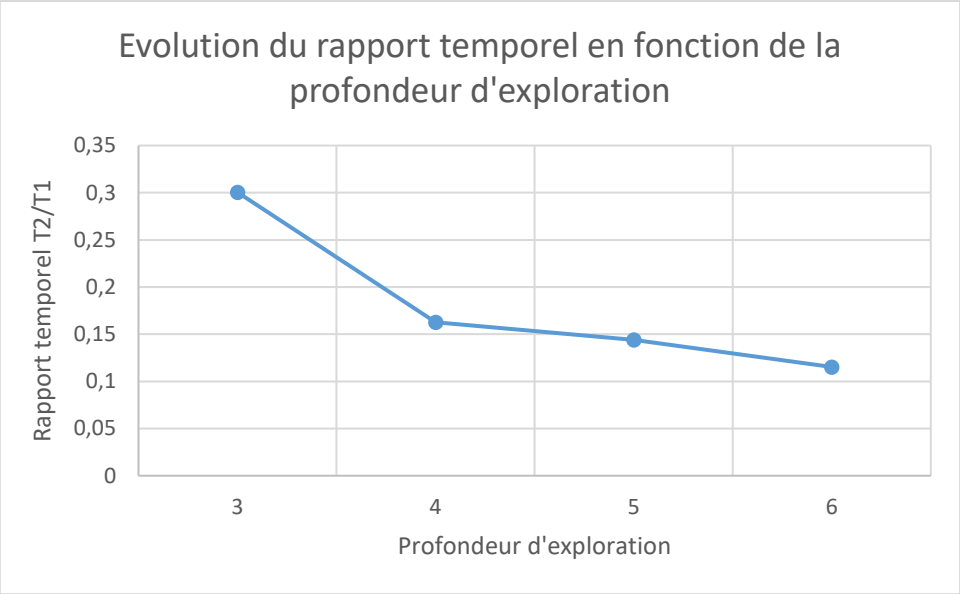
Nous avons ensuite pu tester une partie entre notre « MinimaxBrain » et un « RandomBrain ». Ce dernier joue de manière quasiment instantanée, tandis que notre « MinimaxBrain » a toujours besoin d'un temps limite pour jouer son tour. Au-delà de 7 secondes

pour jouer son tour, pour une profondeur de 5, il ne perd aucune partie. Cependant, le « MinimaxBrain » ne perd jamais, pour une raison autre que le temps limite pour jouer qui est écoulé, contre le « RandomBrain », ce qui est bien normal !

Temps pour jouer un tour (en secondes) pour une profondeur de 5	5	3	2	1.2	1.1	1
Nombre de parties perdues sur 10 parties jouées	3	5	7	8	9	10

Au vu des résultats précédents, nous voyons dès lors la nécessité d'améliorer notre minimax. D'une part, il ne respecte pas du tout la durée limite pour jouer, c'est-à-dire qu'il va toujours explorer jusqu'à la profondeur spécifiée l'arbre, sans se soucier de retourner une valeur avant que son temps pour jouer soit quasiment écoulé, ce qui est normal pour l'instant. D'autre part, il est possible d'optimiser notre code pour qu'il explore l'arbre plus rapidement, ce qui limitera dans un premier temps le nombre de défaites pour cause de temps limite pour jouer écoulé. C'est de ce point dont nous allons nous occuper pour l'instant. C'est pourquoi, nous avons implémenté un élagage alphabeta au sein de notre minimax. Nous avons pour ce faire, procédé de la même manière que pour implémenter notre minimax initial. Nous avons dans un premier temps réfléchi chacun de notre côté, d'abord sur papier puis sur ordinateur, puis nous avons mis en commun nos idées pour réaliser notre minimax avec élagage alphabeta. Ce dernier consiste à cesser d'explorer une branche quand on peut connaître son résultat à l'avance : dans les faits il s'agit de constamment comparer les scores des enfants à celui de leurs oncles. Pour cela, il suffit de passer en argument les scores maximum et minimum de la génération précédente et d'insérer après chaque calcul de score d'enfant, une comparaison avec le minimum ou le maximum des scores des oncles. Cela permet de gagner du temps en explorant l'arbre. PNous avons ensuite comparé la rapidité du minimax avec élagage alphabeta avec celle du minimax initial, à profondeur constante, en réalisant des parties entre ces deux types d'IA. Nous avons donc fait s'affronter une IA avec minimax contre une IA avec minimax avec élagage alphabeta avec un temps limite pour jouer de 7 secondes (pour être sûr que l'IA avec minimax ne perde pas au temps) et avec une profondeur d'exploration variable (même méthode expérimentale que pour la table de transposition).

Profondeur d'exploration	Temps de Minimax2 = T1 (en secondes)	Temps de MinimaxAlphaBeta = T2 (en secondes)
3	0.3415	0.1026
4	2.6505	0.4313
5	5.3730	0.7744
6	39.3337	4.5344



Nous constatons donc que plus la profondeur d'exploration est importante plus l'élagage alphabeta nous fait gagner du temps, toutes choses égales par ailleurs. Ceci est cohérent car plus la profondeur est importante plus le minimax a un arbre grand à explorer et donc plus il y a de branches qui vont être élaguées. Les résultats de ce type de partie étaient toujours des matchs nuls.

V. Minimax avec élagage alphabeta et gestion du temps

Il a ensuite fallu implémenter une IA capable de jouer en un temps imparti, que ce dernier soit infime ou important. En s'appuyant sur le principe décrit dans l'énoncé de la séance, pour réaliser un minimax ayant une gestion du temps, nous avons travaillé à deux directement cette fois pour implémenter cette IA. Pour déterminer le temps mis par la méthode « findNextStates », nous avons instancié un « gameState » dans la méthode « __init__ » de notre IA puis utilisé le module « time » avant et après l'application de la méthode « findNextStates » au « gameState ». Ce délai est passé en paramètres de notre « minimax_time » afin que note minimax gère bien le temps dont il dispose pour évaluer une des positions accessibles depuis l'état de jeu actuel. Le principe du « minimax_time » est : soit le minimax n'a pas le temps d'explorer les fils du nœud étudié et dans ce cas il retourne directement la valeur du nœud soit il a le temps d'explorer ses fils. Dans ce cas on répartit le temps pour jouer sur les différents fils (pour chaque fils on attribue le temps total restant divisé par le nombre de fils à traiter de manière à ce que tout le temps soit utilisé) et le principe récursif du minimax s'applique. Là encore, si le temps restant est inférieur au temps nécessaire à l'exécution de la méthode « findNextStates » alors certains fils peuvent ne pas être étudiés et on détermine alors la valeur du noeud parent à partir des fils étudiés. La première erreur que nous avons rencontrée est que notre IA était incapable de jouer dans le temps imparti, et ce peu importe ce temps. C'est-à-dire que malgré le fait que nous ayons repris notre minimax avec élagage alphabeta et que nous l'ayons fait évoluer afin d'avoir une gestion du temps, notre IA ne jouait plus, elle perdait par conséquent la partie dès le premier tour. Nous avons donc étudié notre « MinimaxBrainAlphaBetaTime » et nous avions oublié de commencer la gestion du temps dès la méthode « play » de notre IA, c'est-à-dire que les différentes positions accessibles depuis l'état de jeu étaient explorés avec chacune le temps limite au lieu que ce dernier soit répartis entre elle. Une fois ce problème corrigé, notre IA fonctionnait bien. Nous avons ensuite installé le profiler Snakeviz sur notre session ENSTA, afin de déterminer le temps passé par notre IA dans les différentes parties du code et d'ainsi pouvoir l'optimiser. Nous avons dans un premier temps profilé notre IA avec

toujours la fonction d'évaluation basique, lors d'une partie entre notre IA et un « RandomBrain ». Nous nous sommes rendu compte que nous pouvions grandement améliorer notre « minimax_time ». Nous avons réussi à le rendre deux fois plus compact et à gagner un peu de temps là-dessus, tout en le maintenant opérationnel notamment en retirant quelques boucles itératives et en faisant attention de réduire le nombre de fois où des fonctions coûteuses étaient appelées. Lors de match entre une IA avec élagage alphabeta et gestion du temps contre une IA avec seulement élagage alphabeta, il y a toujours match nul (quand on laisse un temps raisonnable à l'IA ne gèrant pas le temps), ce qui est cohérent car les deux suivent le même procédé algorithmique. Cependant, l'IA gèrant le temps peut s'adapter au temps limite imposé, que ce dernier soit de 0.1s ou plus. Il joue toujours dans le temps imparti et essaie d'explorer au mieux l'arbre selon le temps limite imposé. Nous avons remarqué lors de plusieurs tests qu'il utilise en moyenne 68 % du temps limite pour jouer. Il serait donc encore possible d'améliorer notre IA pour qu'elle utilise la plus grande part possible du temps limite et donc qu'elle puisse explorer plus profondément l'arbre.

```python
import time
infinite = 10**10

def minimax(state, T, maximize, get_children, evaluate, delai, max_A = -infinite, min_A= infinite):
    start = time.time()
    if(maximize == True):
        maximize0 = False
    else:
        maximize0 = True

    children_list = get_children(state)
    length = len (children_list)
    if(length == 0):
        return(evaluate(state))
    G = []
    i = 0
    end = time.time()
    delay = end-start
    if(T-delay > delai):
        for child in children_list:
            end = time.time()
            delay = end-start
            Time_for_child = (T-delay)/length
            length= length-1

            if(len(G) != 0):
                G.append(minimax(child, Time_for_child, maximize0, get_children, evaluate, delai ,max(G), min(G)))
            else:
                G.append(minimax(child, Time_for_child, maximize0, get_children, evaluate, delai, -infinite, infinite))
            if(maximize == False):
                if(G[i] < max_A):
                    break
            else:
                if(G[i] > min_A):
                    break
            i += 1
            end = time.time()
            delay = end-start
            if(T-delay < delai):
                if(maximize == True):
                    res = max(G)
                    return(res)
                else:
                    res = min(G)
                    return(res)
    else:
        return(evaluate(state))
    if(maximize == True):
        return(max(G))
    else:
        return(min(G))
```

VI. Fonction d'évaluation

Pour ce qui est de la fonction d'évaluation, nous avons songé à plusieurs axes d'amélioration. Comme les dames ont une amplitude de déplacement très supérieure à celle des pions, qui est de une case, il semblerait cohérent de bonifier un état de jeu dans lequel notre IA possède une dame. C'est pourquoi, nous avons décidé d'incrémenter de 2 points (dame = 3 pions, pour ne pas trop influer non plus sur le score d'un état) le score d'un état pour chaque dame présente. Nous n'avons pas modifié davantage notre fonction d'évaluation (qui d'après le profiler est la 2ème partie du code, après le minimax, dans laquelle l'IA passe le plus de temps) puisque nous avons préféré avoir une fonction d'évaluation rapide, ce qui nous permet de passer plus de temps à explorer l'arbre et donc d'aller plus profond. Cependant, on pourrait également imaginer bonifier un état de jeu qui amènerait une situation de rafle car cette prise multiple permettrait de donner rapidement un avantage important à notre IA. Le bonus de point varierait selon si c'est un pion ou une dame qui est en situation de rafle.

VII. Conclusion

Au niveau du travail d'équipe, nous nous sommes plutôt bien organisé et répartit le travail. Chacun avait toujours du travail à réaliser et ce en fonction de ses envies et de ses compétences. Nous avons pu nous voir en dehors des séances de TP imposées pour avancer ensemble ce projet d'IN104. Nous avons beaucoup communiqué et échangé nos codes, c'est pourquoi GitHub a été très utile dans l'accomplissement de notre travail. En effet, il nous a permis de centraliser nos travaux et de pouvoir travailler chacun sur des parties bien spécifiques d'un même programme en parallèle. Nous avons réussi à réaliser les différents algorithmes demandés, à utiliser les outils proposés, notamment le profiler même si nous avons eu du mal à le faire fonctionner à cause de problème d'installation de l'afficheur graphique Snakeviz, et à réfléchir aux différentes façons d'optimiser nos codes et donc de rendre plus performant notre IA.