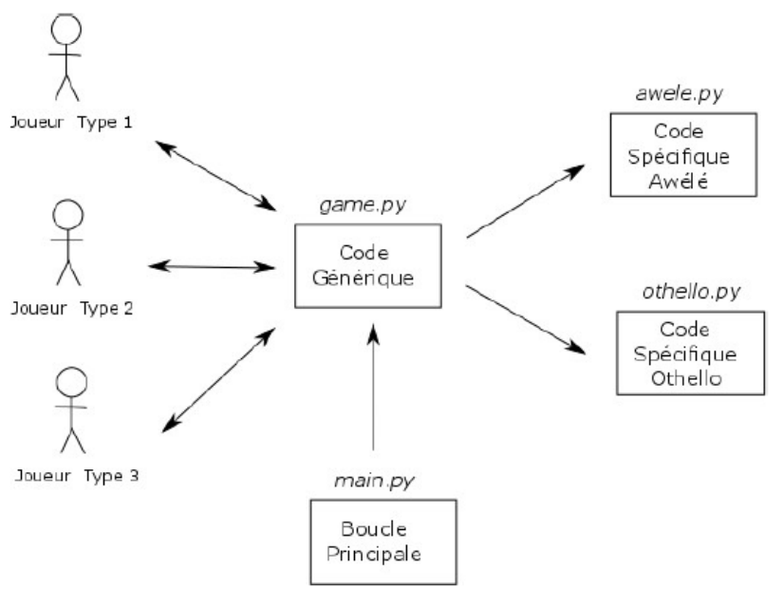
Rapport 2I013 – Projet jeu à deux joueurs

Introduction

Pour créer ce projet, on est passé par plusieurs étapes :

* Implémentation des jeux
* Implémentation des joueurs (humain ou IA)
* Implémentation d'une interface graphique pour Othello avec pygame (on peut y faire jouer deux humains avec un écran et une souris)
* Implémentation d'une fonction d'évaluation combinaison linéaire de plusieurs multiples fonctions d'évaluation élémentaires
* Implémentation de l'algorithme minimax et de sa version optimisée alpha_beta
* Implémentation de différentes techniques d'apprentissage, notamment algorithme génétique et exploration locale (hill-climbing)

Architecture générale



game.py

Les jeux de plateau ont certaines propriétés communes. Les fonctions génériques sont donc placées dans ce fichier. Par exemple, pour les jeux à 2 joueurs comme Othello et Awale, la fonction `getJoueur` ou `getPlateau` permettent d'obtenir le tour du joueur ou la position des pièces sur le plateau.
Mais il y a aussi des fonctions qui sont spécifiques à chaque jeu (par exemple, `getCoupsValides`). Ces fonctions spécifiques sont alors préfixées par le module game qui contient le module du jeu spécifique.
De plus, on respecte les conditions de typage de variable :

```
coup = (int, int)
jeu = List[...] :
      0: plateau       List[List[int]]
      1: joueur        int (1 ou 2)
      2: coups valides List[(int, int)]
      3: coups joues   List[(int, int)]
      4: scores        List[int, int]
```

Enfin, `game.joueur1, game.joueur2` contiennent le module des joueurs stockés dans le dossier Joueurs du jeu en question.

awele.py et othello.py

Ces fichiers contiennent l'implémentation des jeux Awale et Othello.

main.py et main_permutation.py

Ces fichiers permettent de lancer des parties avec le joueur1 qui commence en premier ou en deuxieme. Si l'on veut des parties non déterministes pour faire des statistiques sur le nombre de victoires/égalités/défaites, alors on a intérêt de randomiser les 4 premiers coups.

survival.py et supervise_faible.py

Ces fichiers ont la même structure que les mains mais en y ajoutant différents algorithmes d'apprentissage (algorithme génétique ou supervise faible).

Awele/Joueurs et Othello/Joueurs

Les dossiers Joueurs contiennent tous les joueurs pour un jeu donné : joueur_humain, joueur_random, joueur_premier_coup et les joueurs correspondant à différents algorithmes de décision (minimax, negamax, alpha_beta)

Fonction d'évaluation (heuristique) des coups

Il s'agit simplement d'un produit scalaire entre un vecteur qui représente les poids et un autre qui représente les évaluations élémentaires (features) d'une certaines position du plateau. La fonction `heuristic` correspond à cette fonction d'évaluation. Cette fonction nous sera utile dans les algorithmes de décision minimax, negamax, alpha_beta lorsque la profondeur maximale sera atteinte.

TABLE 4

The Weights of The Features in The Evaluation Function as Generated by The GA for Mini-Max Search of Depth 5

Feature	Weight
The number of pits that the opponent can use to capture 2 seeds. Range: 0 - 6.	0.80
The number of pits that the opponent can use to capture 3 seeds. Range: 0 – 6.	1.00
The number of pits that Ayo can use to capture 2 seeds. Range: 0 - 6.	0.06
The number of pits that Ayo can use to capture 3 seeds. Range: 0 - 6.	0.00
The number of pits on the opponent's side with enough seeds to reach Ayo's side. Range: 0 - 6.	0.87
The number of pits on Ayo's side with enough seeds to reach the opponent's side. Range: 0 - 6.	0.60
The number of pits that has more than 12 seeds on the opponent's side. Range: 0 – 6.	0.00
The number of pits with more than 12 seeds on Ayo's side. Range: 0 - 6.	0.20
The current score of the opponent. Range: 0 - 48.	0.73
The current score of Ayo. Range: 0 - 48.	0.93
The number of empty pits on the opponent's side. Range: 0 – 6.	0.00
The number of empty pits on Ayo's side. Range: 0 – 6.	0.80

En fait, il y a deux cas possibles :

* si la profondeur maximale est atteinte, alors on utilise la fonction heuristic pour évaluer la feuille de l'arbre de décision
* si la partie est terminée avant ou en même temps que la profondeur maximale est atteinte, alors on évalue différemment la feuille. Si le joueur correspondant à l'algorithme (donc le joueur qui veut gagner) est perdant, on affecte à cette feuille -10000+plies, si égalité 0 sinon 10000-plies où plies represente en fait la profondeur.

On pourra faire varier les poids pour améliorer (ou pas) de tels joueurs à la main ou via des techniques d'apprentissage.

Plus il y a de fonctions d'évaluations élémentaires, plus on a une meilleure représentation d'un état donné du jeu et donc on peut diminuer la profondeur maximale et les ressources necessaires consommées par l'ordinateur.

Le joueur Ayo décrit dans l'article « Ayo, the Awari player, or how better representation trumps deeper search » arrive à avoir de meilleurs résultats de victoire à profondeur 5 que le joueur de Davis et al à profondeur 7 en ajoutant 6 heuristiques de plus.

On a donc « cloné » le joueur Ayo et on a bien observé qu'il était meilleur qu'un joueur avec seulement 4 heuristiques. La fonction d'évaluation de Ayo est décrite dans l'article (voir Table 4) et les poids ont été obtenus via un algorithme génétique contre une IA qui possède 4 niveaux.

Pour Othello, on a utilisé une fonction d'évaluation simple avec les heurisitiques élémentaires suivantes :

* positional qui correspond à une « heatmap » des positions du plateau
* d_mobility
* d_score

On peut trouver sur internet :

Game Phases

- An Othello game can be split into three phases where strategies can differ:
  - Beginning
    - First 20 to 25 moves
  - Middle
  - End
    - Last 10 to 16 moves
- Usually heuristic players use Positional/Mobility for beginning and middle phases. Then switch to Absolute for the end phase

```
private int boardWeight[][] = {
    { 100, -10, 11,  6,  6, 11, -10, 100 },
    { -10, -20,  1,  2,  2,  1, -20, -10 },
    {  10,   1,  5,  4,  4,  5,   1,  10 },
    {   6,   2,  4,  2,  2,  4,   2,   6 },
    {   6,   2,  4,  2,  2,  4,   2,   6 },
    {  10,   1,  5,  4,  4,  5,   1,  10 },
    { -10, -20,  1,  2,  2,  1, -20, -10 },
    { 100, -10, 11,  6,  6, 11, -10, 100 }
};
```

Figure 7 – Board weights matrix

Comparaisons espace-temps des algorithmes de décision

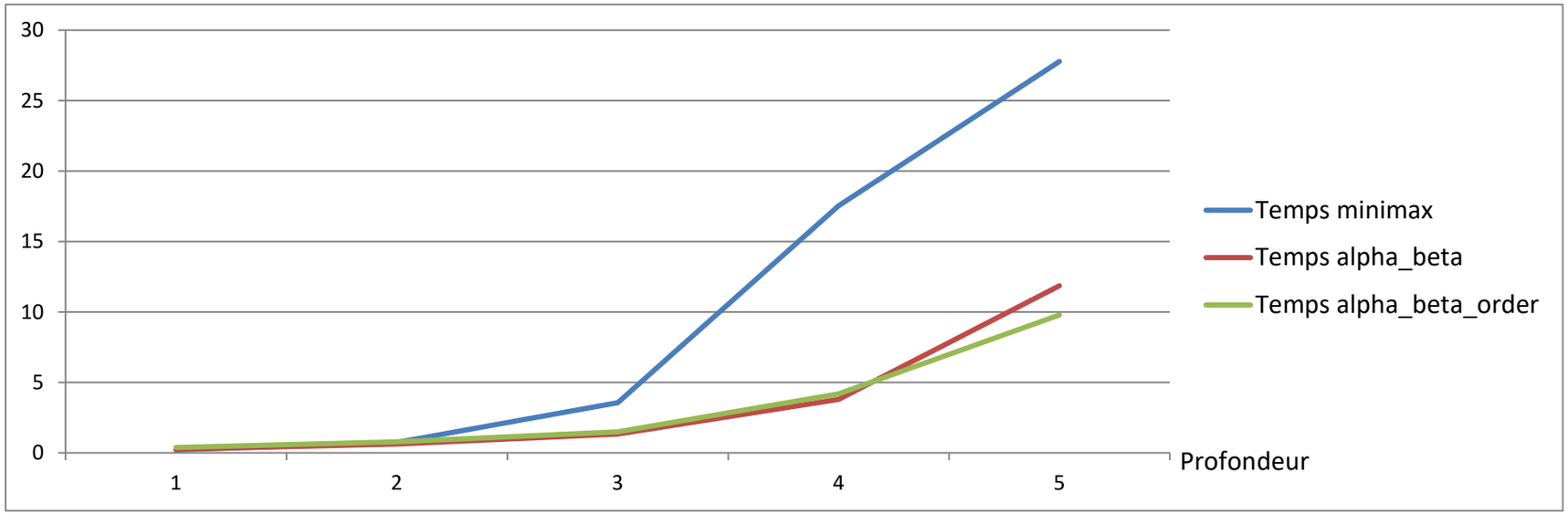
Pour effectuer ces comparaisons, on a utilisé des parties non deterministes d'Awale contre un joueur_premier_coup et on a pris la moyenne de temps et de nbNoeuds pour 100 parties (sauf pour minimax profondeur 5 où l'on a fait seulement 10 parties).

Profondeur	Temps minimax	Temps alpha_beta	Temps alpha_beta_order	nbNoeuds minimax	nbNoeuds alpha_beta	nbNoeuds alpha_beta_order
1	0.20571	0.25454	0.38073	122.9	135.8	115.6
2	0.73724	0.62136	0.78934	535.4	379	374.2
3	3.56173	1.32762	1.49205	2741.2	1115	1048
4	17.53777	3.80026	4.18633	12745	2688.65	2864.15
5	27.77323	11.85777	9.78384	43764	8558.5	7242.5

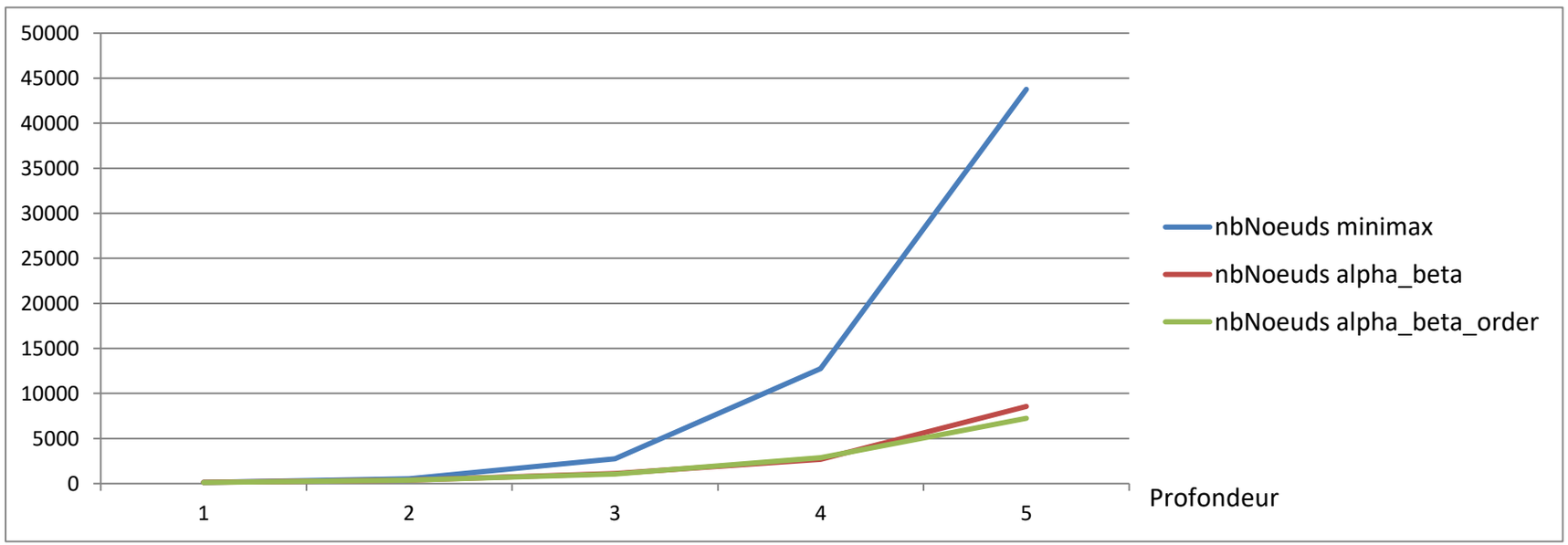
ATTENTION: pour alpha beta avec tri superficiel, il y a un facteur chance. Parfois une partie peut être très rapide. Cela dépend de l'ordonnancement des nœuds

Mais en moyenne il vaut mieux laisser les nœuds tels qu'ils sont, car trier (même pour 1 seul niveau) rajoute de la complexité temps

Temps moyen (en secondes) pour 100 parties des algorithmes de décision en fonction de la profondeur



nbNoeuds moyen pour 100 parties des algorithmes de décision en fonction de la profondeur



Apprentissage (métaheuristiques)

Méthode hill-climbing (exploration locale) :

On modifie (ou pas) de +/– epsilon chaque coordonnée du vecteur de poids à chaque fois que le joueur s'améliore.

Cet algorithme est peu efficace :

* Estimations coûteuses : nécessitent un grand nombre de simulations
* Estimations peu fiables : large part d'aléatoire, mauvaises décisions probables
* Interdépendance des paramètres, chaque décision sur un paramètre peut impliquer de reconsidérer tous les autres
* Nombreux optima locaux

Avec le joueur alpha beta à 4 heuristiques, on obtient :

```
Au debut: [1.0, 0.0, 0.0, 0.0]
THRESHOLD_score_max = 8.0

*** DEB *** score_max = 3
*** Sens + *** score_max = 4
Index 0: 1.1
*** Sens + *** score_max = 6
Index 1: 0.1
Index 2: Aucune modification
Index 3: Aucune modification

*** FIN *** [1.1, 0.1, 0.0, 0.0]
score_max = 6

Temps: 4.1506758570671085 minutes
```

Algorithme génétique (exploration locale) :

Conditions d'application (Wikipedia) :

- Le temps de calcul de la fonction d'évaluation (*fitness* en anglais) doit être raisonnablement court. En effet, celle-ci sera évaluée de nombreuses fois.
- Nombre de solutions important : les performances des algorithmes génétiques par rapport aux algorithmes classiques sont plus marquées lorsque les espaces de recherches sont importants. En effet, pour un espace dont la taille est faible, il peut être plus sûr de parcourir cet espace de manière exhaustive afin d'obtenir la solution optimale en un temps qui restera somme toute correct. Au contraire, utiliser un algorithme génétique engendrera le risque d'obtenir une solution non optimale (voir la section *limites*) en un temps qui restera sensiblement identique.
- Pas d'algorithme déterministe adapté et raisonnable.
- Lorsque l'on préfère avoir une solution relativement bonne rapidement plutôt qu'avoir la solution optimale en une durée indéfinie. C'est ainsi que les algorithmes génétiques sont utilisés pour la programmation de machines qui doivent être très réactives aux conditions environnantes.

Malheureusement, les 2 premières conditions ne sont pas respectées …

Limites (Wikipedia) :

- Le temps de calcul : par rapport à d'autres métaheuristiques, ils nécessitent de nombreux calculs, en particulier au niveau de la fonction d'évaluation.
- Ils sont le plus souvent difficiles à mettre en œuvre : des paramètres comme la taille de la population ou le taux de mutation sont parfois difficiles à déterminer. Or le succès de l'évolution en dépend et plusieurs essais sont donc nécessaires, ce qui limite encore l'efficacité de l'algorithme. En outre, choisir une bonne fonction d'évaluation est aussi critique. Celle-ci doit prendre en compte les bons paramètres du problème. Elle doit donc être choisie avec soin.
- Il faut aussi noter l'impossibilité d'être assuré, même après un nombre important de générations, que la solution trouvée soit la meilleure. On peut seulement être sûr que l'on s'est approché de la solution optimale (pour les paramètres et la fonction d'évaluation choisie), sans la certitude de l'avoir atteinte.
- Un autre problème important est celui des optima locaux. En effet, lorsqu'une population évolue, il se peut que certains individus qui à un instant occupent une place importante au sein de cette population deviennent majoritaires. À ce moment, il se peut que la population converge vers cet individu et s'écarte ainsi d'individus plus intéressants mais trop éloignés de l'individu vers lequel on converge. Pour vaincre ce problème, il existe différentes méthodes comme l'ajout de quelques individus générés aléatoirement à chaque génération, des méthodes de sélection différentes de la méthode classique…

La phase d'exploration de l'algorithme passe par la **création d'une population d'initiale** qui correspond à un ensemble de vecteurs de poids (gènes). Puis on fait jouer chaque individu de cette population avec le joueur MASTER. On passe ensuite par une phase de **selection** des meilleurs individus de la population qui sont evalués selon une **fitness function** (nombre total de parties gagnés). On passe enfin par une phase de **crossovers et de mutations** pour

apporter de la diversité et ainsi explorer de nouveaux vecteurs de poids qui peuvent (ou pas) rendre notre IA plus performante. Il est nécessaire de choisir un taux de mutation relativement faible, de manière à ne pas tomber dans une recherche aléatoire et à conserver le principe de sélection et d'évolution. Cette phase de crossovers et mutations permet aussi d'éviter une convergence prématurée vers un extremum local, problème que l'on rencontre dans la technique de hill-climbing.
Enfin, on répète le processus pour un nombre N de générations jusqu'à un atteindre une valeur de fitness function satisfaisante (80% de parties gagnantes par exemple). On s'est contenté de seulement 2 générations car notre fitness function prend beaucoup de temps …
Avec le joueur alpha beta Ayo, on obtient :

```
                                        DEPTH 4

    CONTRE MASTER:

            [0.5534748381291195, 0.4616523993252173, 0.493022495979617 67, 0.6862599526142863,
0.3444121846742263, 0.721958905863068, 0.6509160093379074, 0.44053105064314346, 0.6506520262665414,
0.5194329933897871, 0.355696633999106, 0.7590032048352058]



=== GEN 1 === 3 best solutions ===

(5, [0.4746814056431268, 0.08439711520556892, 0.371885043837853 04, 0.53612423380965 42,
0.20152165347756 38, 0.338699367303184 95, 0.130445018844691 84, 0.73662106510895 73, 0.333026267726128 76,
0.881107494133617 8, 0.172923414574154 72, 0.352707351729732 1])

(5, [0.3629896622951405, 0.5460233393759555, 0.2582546488803469, 0.014193906570616144,
0.6158182971869506, 0.06469030488498273, 0.31402729074745 93, 0.03668154951482461, 0.5024514150150688,
0.3983320325096431, 0.8003992606869791, 0.5519372799935668])

(4, [0.6181246226825173, 0.8840541044629479, 0.24739152341908366, 0.19462225033223268,
0.2808294023833151, 0.06252527973225486, 0.9962916467394782, 0.5690770842848133, 0.66011648869301,
0.09964392425999646, 0.7233205659079345, 0.95090807756047 48])
```

```
=== GEN 2 === 3 best solutions ===

(8, [0.6144990095777895, 0.08390208349891629, 0.3697037502298756, 0.19348069253727876,
0.2003396271985201, 0.06431086357406718, 0.1296798929260424, 0.5657391597649498, 0.6562445720855371,
0.8759393536645629, 0.17190913141022976, 0.9453305214158368])

(6, [0.47189715945438643, 0.08390208349891629, 0.2567398549294539, 0.5329795944011367,
0.6122062118460969, 0.062158537387337016, 0.1296798929260424, 0.5657391597649498, 0.49950428369572103,
0.3959956139556752, 0.7957045147699381, 0.3506385449719369])

(4, [0.6144990095777895, 0.08390208349891629, 0.2459404471855981, 0.5329795944011367,
0.6122062118460969, 0.062158537387337016, 0.9904478930399837, 0.7323004105210728, 0.6562445720855371,
0.8759393536645629, 0.17190913141022976, 0.3506385449719369])



Temps: 30.120272394021352 minutes
```