

## TP 11 : Sherlock's Gambit

### Consignes de rendu

À la fin de ce TP, vous devrez rendre un dépôt Git respectant l'architecture suivante :

```
csharp-tp11-sherlock.holmes/  
|-- README  
|-- .gitignore  
|-- SherlockGambit/  
    |-- SafetyNet/  
        |-- Tout sauf bin/ et obj/  
    |-- SherlockGambit/  
        |-- Tout sauf bin/, obj/ et DotHelper/  
    |-- SherlockGambit.sln  
    |-- SherlockGambit.sln.DotSettings
```

N'oubliez pas de vérifier les points suivants avant de rendre :

- Remplacez `sherlock.holmes` par votre propre login.
- Le fichier `README` est obligatoire.
- Pas de dossiers `bin` ou `obj` dans le projet.
- Respectez scrupuleusement les prototypes demandés.
- Retirez tous les tests de votre code.
- **Le code doit compiler !**

### README

Vous devez écrire dans ce fichier tout commentaire sur le TP, votre travail, ou plus généralement vos forces / faiblesses, vous devez lister et expliquer tous les boni que vous aurez implémentés. Un `README` vide sera considéré comme une archive invalide (malus).

#### .gitignore

Vous avez peut-être, et si c'est le cas félicitations, remarqué que le `.gitignore` qui vous a été donné ne respecte pas l'architecture du TP. En effet, il n'est pas situé à la racine du projet mais plutôt à la racine du dossier de la solution `SherlocksGambit/`. Pour une architecture plus propre, veuillez déplacer le `.gitignore` à son emplacement correct. En étant à la racine de votre projet (dans `csharp-tp11-sherlock.holmes/`) exécutez `move SherlockGambit/.gitignore .`

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Objectifs . . . . .	3
<b>2</b>	<b>Cours</b>	<b>3</b>
2.1	Diagrammes UML . . . . .	3
2.2	Le jeu de Dames . . . . .	5
2.2.1	Notation de Forsyth-Edwards (FEN) . . . . .	5
2.2.2	Notation des coups . . . . .	7
2.3	L'algorithme MiniMax . . . . .	8
2.4	Elagage alpha-bêta (Alpha-Beta pruning) . . . . .	10
2.5	Fonction d'heuristique . . . . .	11
2.6	La boîte à outils de Sherlock . . . . .	12
2.6.1	Liens utiles . . . . .	12
2.6.2	Notations . . . . .	13
2.6.3	Des tests à votre disposition . . . . .	15
2.6.4	Visualisation du format Dot . . . . .	16
<b>3</b>	<b>Exercices</b>	<b>17</b>
3.1	Etape 1 : découvrez le plateau de jeu . . . . .	17
3.1.1	Encodage FEN . . . . .	18
3.2	Etape 2 : L'IA avec son plateau de jeu . . . . .	19
3.2.1	Est-ce déjà la fin ? . . . . .	19
3.2.2	La guerre des clones . . . . .	19
3.2.3	Transmettre un coup à un autre plateau . . . . .	20
3.3	Etape 3 : le coeur de l'IA . . . . .	21
3.3.1	Générer une liste de coups possibles . . . . .	21
3.3.2	Heuristique : fonctions par défaut . . . . .	22
3.3.3	Trouver un coup . . . . .	22
3.4	Etape 4 : votre IA contre le monde . . . . .	24
3.4.1	Encoder un coup . . . . .	24
3.4.2	Décoder un coup . . . . .	24
3.4.3	Fonction pilote de l'IA . . . . .	25
3.4.4	Instantier et tester l'IA . . . . .	26
3.5	La FAQ . . . . .	27
<b>4</b>	<b>Bonus</b>	<b>28</b>
4.1	Cours . . . . .	29
4.1.1	Les tables de transposition . . . . .	29
4.1.2	Quiescence search (Recherche de sérénité) . . . . .	30
4.2	Bonus 1 : La sérénité s'offre à vous . . . . .	32
4.3	Bonus 2 : NegaStonks . . . . .	32
4.4	Bonus 3 : Table de transposition . . . . .	32
4.4.1	Fonction de hachage Zobrist . . . . .	32
4.4.2	La classe Entry . . . . .	33
4.4.3	Table de transposition . . . . .	33
4.4.4	Intégrer la table de transposition à votre IA . . . . .	34
<b>5</b>	<b>Pour aller vers l'infini et au-delà</b>	<b>35</b>

# 1 Introduction

## 1.1 Objectifs

Ce TP a pour but de vous introduire aux notions d'intelligences artificielles (IA) en passant par l'un des algorithmes en la matière les plus connus : l'algorithme Minimax. En plus de ces notions, nous vous présenterons des notions d'optimisation poussées pour rendre votre IA la plus efficace possible. Ce TP s'inscrit dans la philosophie d'un des grands projets du S6 à EPITA : *Chess*®.

## 2 Cours

Cette section regroupe l'essentiel des notions et règles à suivre pour ce TP. Lire attentivement le cours vous aidera à débiter le TP avec de bonnes bases.

### 2.1 Diagrammes UML

Au travers de ce TP, vous allez découvrir un diagramme qui va vous guider dans la compréhension des différentes parties de votre implémentation.

Ce diagramme suit une norme standardisée et reconnue : [Unified Modeling Language](#).

Elle vise à simplifier la description de structures orientées-objet. Et donc pour faire ce TP, vous allez devoir vous familiariser avec ces normes.

Tout d'abord, une classe est représentée dans un rectangle contenant son nom, ses attributs et méthodes internes comme ceci :

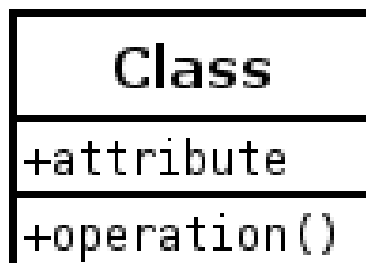


FIGURE 1 – Une classe en UML

Les classes peuvent être reliées par divers relations, voici un florilège des plus communes :

- Relation d'agrégation : il s'agit de décrire la relation entre une classe qui contient un attribut appartenant à une autre classe. Par exemple, un plateau a des pièces, donc il y a une relation d'agrégation entre les deux classes.

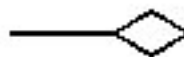


FIGURE 2 – Représentation graphique d'une agrégation

- Les relations d'héritage sont représentées par des flèches classiques.

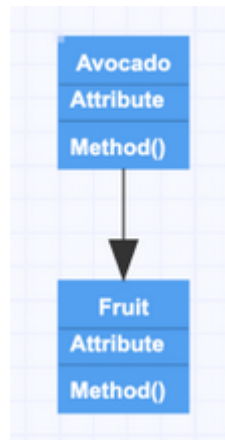


FIGURE 3 – Relation d'héritage

- Lorsqu'une classe implémente une classe abstraite ou une interface, la flèche devient alors pointillée.

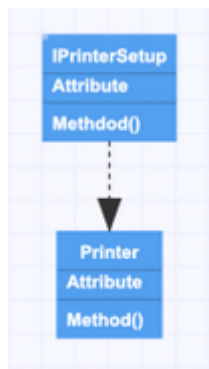


FIGURE 4 – Relation marquant l'implémentation

Avec ces informations, vous devriez pouvoir débiter le TP sans trop de problème. Cependant, si vous souhaitez plus d'information sur les différentes relations entre les classes, cette discussion [StackOverflow](#) contient des schémas et exemples plus explicites.

## 2.2 Le jeu de Dames

Pour ce TP, vous devez vous familiariser avec le jeu de Dames. Pour cela, nous vous conseillons de regarder quelques vidéos sur Youtube pour en comprendre les règles. Nous vous suggérons celles-ci :

- [Vidéo 1](#) (1m53)
- [Vidéo 2](#) (4m05)

### Modification des règles

Les pions et les dames peuvent manger plusieurs pièces en un seul coup mais quelques restrictions sont appliquées. Elles sont explicitées dans une partie ultérieure.

### Les positions ne sont pas des index !

Le plateau suit une notation de 1 à 32 avec une case sur deux jouable. La cellule n°1 se situe en haut à gauche et la 32 en bas à droite. La lecture se fait donc de gauche à droite.

### 2.2.1 Notation de Forsyth-Edwards (FEN)

Durant ce TP, vous devrez vous habituer à une notation classique pour représenter chacun de vos plateaux de jeu. Historiquement, elle est associée au jeu d'échecs mais nous l'avons adaptée au contexte de ce TP.

Tout d'abord prenons un exemple avec ce plateau :

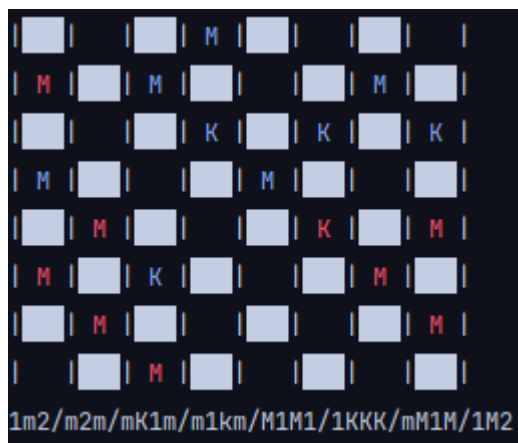


FIGURE 5 – Le plateau "1m2/m2m/mK1m/m1km/M1M1/1KKK/mM1M/1M2"

Vous pouvez remarquer que chaque ligne est séparée par un /, et que nous avons huit lignes au total sur un plateau.

Cette représentation se fait par convention en commençant par le bas gauche du plateau. Le sens de lecture est de **gauche à droite** du **bas vers le haut**.

Pour savoir à qui appartient chaque pièce, nous mettons par convention les lettres majuscules pour le joueur ayant les pièces noires et les lettres minuscules pour le joueur ayant les pièces blanches.

Dans les dames, nous n'avons que deux types de pièces ce qui simplifie grandement la notation :

1. Les pions (*Men*) : représentés par le caractère 'M' ou 'm' sur le plateau
2. Les dames (*King*<sup>1</sup>) : représentés par le caractère 'K' ou 'k' sur le plateau
3. Les espaces vides ne sont pas représentés directement par une lettre mais plutôt par leur nombre. Ainsi, '4' indique une ligne vide et '1m1m' indique la présence d'un espace avant un pion puis un espace entre les deux pions de la ligne.

Nous vous mettons quelques exemples supplémentaires à votre disposition pour vous familiariser avec cette notation au coeur du jeu :

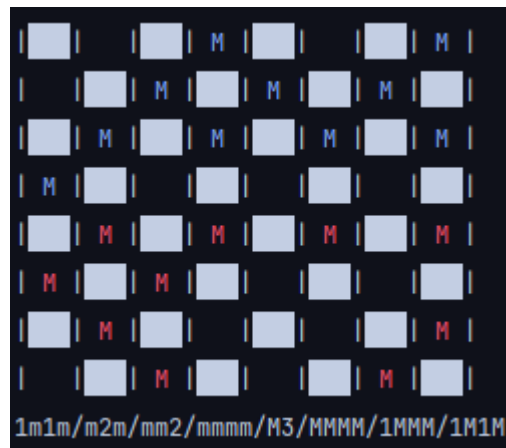


FIGURE 6 – Le plateau "1m1m/m2m/mm2/mmmm/M3/MMMM/1MMM/1M1M"

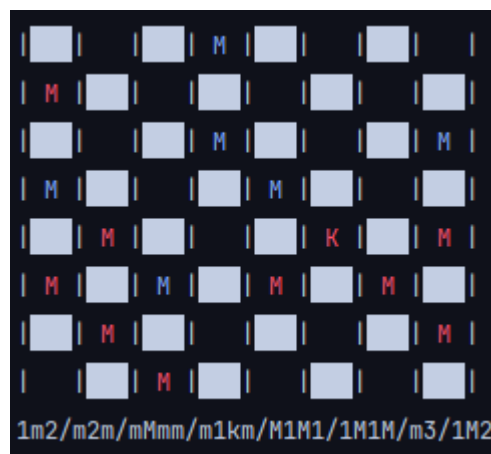


FIGURE 7 – Le plateau "1m2/m2m/mMmm/m1km/M1M1/1M1M/m3/1M2"

1. La version anglaise du jeu n'utilise pas le terme *Queen*

### 2.2.2 Notation des coups

Pour rendre vos coups lisibles, vous allez devoir utiliser un format spécifique. Commençons par deux exemples pour illustrer les concepts principaux : (5-9) et 15x6

Tout d'abord, vous pouvez remarquer la présence de deux nombres, le premier représente le point de départ du coup tandis que le second représente sa destination. Ainsi, on peut lire que la pièce en 5 va vers la case 9 pour le premier exemple et de 15 vers 6 pour le second.

Les parenthèses du premier exemple indiquent que le coup est effectué par les noirs. Inversement, leur absence sur le second indique que le coup est pour les blancs.

Finalement, le tiret central indique un coup simple (pas de pièce mangée). Et à l'opposé le 'x' indique que la pièce mange des adversaires sur son chemin.

Normalement, vous avez tout ce qu'il vous faut pour manipuler la notation durant ce TP. Voilà un dernier exemple pour la route : 27x18<sup>2</sup>

#### Attention

Vous l'avez sûrement remarqué, la croix dans la notation ne donne pas d'information sur le chemin parcouru. Les pièces peuvent enchaîner plusieurs mouvements en un seul tour pour manger plusieurs pièces ennemies. Nous utiliserons un tri par la distance puis par le sens des mouvements en sens horaire pour choisir un unique coup. Le sens horaire commence par la diagonale droite vers le haut puis diagonale droite vers le bas comme une horloge classique.

---

2. Coup des blancs de 27 vers 18 qui mange au moins une pièce

## 2.3 L'algorithme MiniMax

MiniMax est un algorithme qui s'applique à la théorie des jeux. Il se distingue des algorithmes d'intelligence artificielle que vous connaissez par son aspect non-génétique. Il entre plus dans une catégorie proche des automates<sup>3</sup>. Cet algorithme bien que moins *mainstream* s'applique à de nombreux jeux dont :

- Les Échecs
- Les Dames (quel hasard !)
- Le Poker
- Pierre-feuille-ciseau
- et tant d'autres...

Pour en expliquer le principe, nous allons passer par votre imagination : imaginez que vous êtes en train de jouer aux échecs et que vous vous apprêtez à faire un coup. Vous vous demandez alors ce qu'il va se passer chez l'adversaire au tour suivant pour voir si vous n'allez pas vous retrouver dans une situation désavantageuse.

MiniMax applique ce schéma de pensée à vos ordinateurs pour en décupler la puissance<sup>4</sup> sur un petit intervalle de temps. Il explore des chemins (des branches) d'un arbre de possibilités (proche de vos arbres binaires généraux en algo).

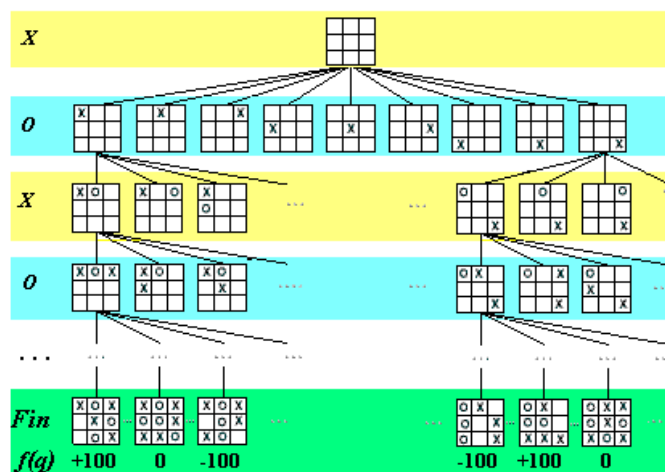


FIGURE 8 – Un exemple d'arbre décisionnel sur un morpion

3. Plus d'information sur ce [lien](#)

4. Un humain en plus efficace sur le papier



En ayant ce principe en tête, on peut décrire un pseudo-processus pour le jeu de Dames :

1. Déterminer tous les coups disponibles pour le joueur J1
2. Effectuer chacun de ces coups dans le cadre d'une simulation tout en demandant un coup à l'adversaire à chaque fois.
3. Nous répétons cet échange plusieurs fois<sup>5</sup> et nous cherchons le résultat optimal.

Ce premier squelette est intéressant. Cependant, vous devez vous demander comment on détermine le meilleur coup.

Pour cela, l'algorithme représente la réussite ou l'échec d'un joueur par un nombre flottant qui se calcule grâce à une fonction d'heuristique. Elle a pour but d'évaluer le résultat d'une feuille de notre arbre de choix en prenant divers facteurs en compte. Ces facteurs peuvent être en plus du gain et de la perte possible d'une partie, le nombre de pièces restantes pour chaque joueur et leur valeurs. Une fois calculé, ce nombre remonte dans tous les noeuds afin de déterminer si un choix doit primer sur un autre : c'est la minimisation ou la maximisation de la valeur d'heuristique.

En suivant ce principe, on peut établir un premier pseudo-code :

```
1  function minimax(node, depth, minimizingPlayer) is
2      if depth = 0 or node is a win or a loss then
3          return the heuristic value of node
4      if not minimizingPlayer then (* maximizing player 1 *)
5          value := -inf
6          for each possible move by player 1 do
7              value := max(value, minimax(child, depth + 1, FALSE))
8      else (* minimizing player 2 *)
9          value := +inf
10         for each possible move by player 2 do
11             value := min(value, minimax(child, depth + 1, TRUE))
12     return value
```

Sur ce pseudo-code, vous pouvez remarquer quelques éléments non mentionnés dans les paragraphes précédent. Tout d'abord le paramètre *depth*, il s'agit en fait du nombre de coups permis à notre IA afin de limiter la durée de traitement. D'autre part, le booléen *minimizingPlayer* permet de mettre des mots sur un principe de base de l'algorithme : on cherche à maximiser la réussite du camp allié et à minimiser celle du camp adverse. Il sert donc à alterner la logique de maximisation dans l'algorithme un tour sur deux.

Un problème de temps se pose, les jeux ont des limites de temps par tour et une recherche en profondeur avec cet algorithme coûte beaucoup en temps par sa nature récursive non terminale (pas de transformation possible en boucle while ou for). De plus, la plupart des jeux ont un nombre de coups en facteur exponentiel, on parle de l'ordre de  $10^{40}$  coups possibles et  $10^{10}$  plateaux possibles dans une partie pour le jeu de dames. Limiter le nombre de tours joués est une possibilité avec le paramètre *depth*, mais n'est pas satisfaisant car votre IA perdra en capacité de prédiction. Il faut donc tenter d'améliorer les performances de l'algorithme de base.

---

5. entre 5 et 10 fois au plus

## 2.4 Elagage alpha-bêta (Alpha-Beta pruning)

Pour remédier à ce problème, nous vous proposons d'implémenter une version optimisée de l'algorithme précédant.

Le but de cette extension est de couper les récursions évoquées dès que possible pour limiter son impact sur les performances de l'algorithme.

Pour illustrer concrètement l'algorithme, nous allons prendre l'exemple de coupure le plus à gauche sur le schéma suivant :

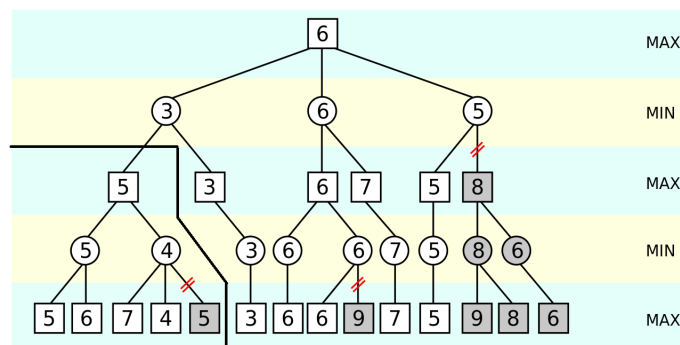


FIGURE 9 – Un exemple d'élagage alpha-bêta

Voici une explication de ce qui se passe lors du parcours profondeur du sous-arbre dont la racine est 5 (délimité en noir) :

- Lors de l'évaluation, le nœud à gauche de 4 retourne 5 (rond). En effet, il s'agit d'un nœud minimisant sa valeur de retour (MIN), et ses deux enfants sont composés de 5 et 6. Nous avons donc  $\min(5, 6) = 5$
- En remontant du parcours profondeur, le nœud MAX a pour valeur 5 et appelle son enfant à droite. Celui-ci sait alors que le nœud au dessus de lui veut une valeur de retour plus grande que 5 car il souhaite maximiser sa valeur de retour.
- Par conséquent, le nœud sait qu'il doit retourner une valeur plus grande que 5 pour que son résultat soit utile. Mais, nous remarquons qu'il rencontre d'abord un 7. Cette valeur est plus grande que 5 donc potentiellement utile.
- Puis, il explore la feuille ayant pour valeur 4. Cette valeur remplace 7 car  $\min(4, 7) = 4$ . Cependant, cette valeur est inférieure à 5. De plus, nous avons l'égalité suivante :  $\min(4, x) \leq 4$ . Ainsi, il n'est plus possible d'obtenir des autres feuilles une valeur pouvant augmenter la valeur du parent (5). Nous pouvons arrêter l'évaluation là.
- Finalement, le nœud MAX garde la valeur 5.

Ainsi nous arrêtons l'évaluation plus tôt que dans l'algorithme *minimax* classique et cela n'a pas d'impact sur le résultat de l'algorithme. Si vous en êtes pas convaincu,  $\min(4, 5) = 4$  avec la dernière feuille, ce changement n'a donc pas d'impact. Cet arrêt en pleine récursion est appelé coupure  $\alpha$  ou  $\beta$  selon des critères que nous allons vous expliciter.

Dans notre algorithme, nous devons introduire deux nouvelles variables que nous nommerons :  $\alpha$  et  $\beta$ . Elles servent à garder en mémoire l'état<sup>6</sup> des parents lorsque l'enfant est appelé. Elles sont initialisées avec les valeurs suivantes :  $\alpha = -\infty$  et  $\beta = +\infty$ .

Selon le type de nœud traité, l'algorithme va mettre à jour l'une des deux variables.

6. la valeur heuristique

Par convention, nous avons  $\alpha$  qui est maximisé et  $\beta$  qui est minimisé. D'autre part, dans le parcours profondeur, pour qu'un noeud soit considéré comme utile, il est obligatoire d'avoir  $\alpha < \beta$ . Ainsi, vous pouvez couper votre traitement sur un noeud dès que  $\beta \leq \alpha$ . Ce sont les fameuses coupures alpha bêta.

#### Attention

Nous vous demandons de respecter ces conventions de nommage pour avoir une bonne lisibilité sur vos codes. **Tout code ne respectant pas ces conventions sera pénalisé.**

Pour conclure, cette optimisation vous permet d'économiser un bon nombre de récursions dans vos parcours profondeurs grâce aux approximations présentées ci-dessus. Cependant, elles n'ont pas d'impact sur la décision prise par l'IA au final.

Nous vous conseillons de faire vous-même les autres exemples sur l'arbre ci-dessus et de consulter des pseudo-codes<sup>7</sup> pour vous familiariser avec ces principes fondamentaux.

## 2.5 Fonction d'heuristique

Nous avons évoqué dans la partie précédente que le succès ou l'échec de l'algorithme devait être représenté par un nombre. Dans cette partie, nous allons vous expliquer comment fonctionnent les fonctions devant calculer ce nombre.

L'heuristique est une méthode d'optimisation dans le domaine de l'informatique. Elle est appliquée à la résolution de problèmes complexes par approximations.

On définit généralement des fonctions pour calculer l'heuristique pour un problème donné. Ainsi, il n'y a pas **une** mais **des** fonctions pouvant résoudre un problème puisqu'elles doivent être adaptées au problème traité. Le résultat d'une telle fonction est un nombre devant servir à la prise de décision d'un algorithme.

Cette fonction se doit donc d'être **non-aléatoire**<sup>8,9</sup>, rapide et complète.

Une fonction heuristique naïve pour les Dames peut être de retourner le nombre de pièces vivantes pour un joueur donné, ou sinon selon la situation retourner un nombre très grand quand un joueur a gagné ou très petit quand il a perdu. C'est une base sur laquelle vous pouvez vous appuyer pour construire des conditions plus complexes en relation avec d'autres facteurs relatifs aux pièces (valeur, position, ...).

Cette fonction sert au final à manipuler votre algorithme décisionnel et donc si votre heuristique n'est pas assez précise, votre IA prendra de mauvaises décisions. À vous de déterminer dans ce TP quelle fonction peut être la plus optimale et quels facteurs peuvent être pris en compte pour obtenir les meilleurs choix.

#### Conseil

Vous serez évalué sur l'efficacité de votre IA et votre score en dépendra. Il est donc conseillé de prendre le temps de réfléchir, essayer, et réessayer pour trouver une fonction qui convient.

7. [https://fr.wikipedia.org/wiki/%C3%89lagage\\_alpha-b%C3%AAta](https://fr.wikipedia.org/wiki/%C3%89lagage_alpha-b%C3%AAta)

8. Toujours générer le même résultat pour une situation précise

9. Une part d'aléatoire peut être ajoutée dans MiniMax mais jamais dans cette fonction

## 2.6 La boîte à outils de Sherlock

### 2.6.1 Liens utiles

Documentation en ligne + Q&A : <https://vertven.github.io/checkers-doc>

Leaderboard : <https://checkmycheckers.engineer>

Sujet + Squelette + Fichiers du client : <https://moodle.cri.epita.fr/course/view.php?id=578>

Il est essentiel que vous compreniez parfaitement la relation entre les classes et le fonctionnement de leurs méthodes. Pour cela, nous vous avons fourni une documentation complète pour toutes les fonctions que vous devez implémenter et toutes celles déjà implémentées. Cette documentation se trouve dans votre squelette ou en ligne. En plus de cela, vous pouvez trouver sur la documentation en ligne un UML décrivant les relations entre toutes les classes. Tout ce qui concerne l'utilisation du client ou du leaderboard se trouvera dans la documentation en ligne.

#### Important

Ce client est la propriété intellectuelle des ACDCs. Toute tentative de décompilation du binaire ou de rétro-ingénierie de l'API sera sévèrement punie. De même pour toute tentative de tricherie.

#### Attention

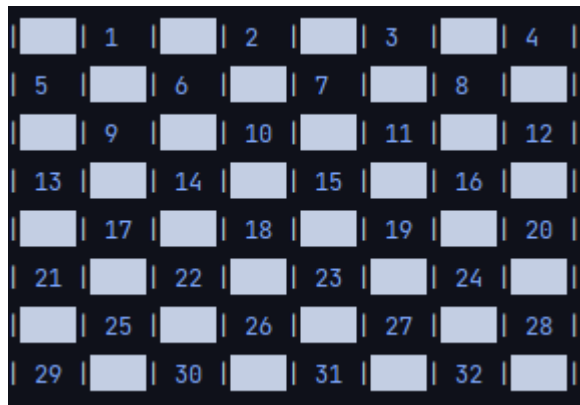
Il est également important de vous rappeler que le client est susceptible d'être modifié pendant toute la durée du TP. Vous devrez retélécharger le client depuis la page moodle à chaque mise à jour.

## 2.6.2 Notations

Prenons le temps de clarifier les quatre notations différentes utilisées dans ce TP.

- Les positions sur le plateau :

Comme indiqué précédemment dans la partie cours, toutes les positions possibles se sont vues attribuer un numéro de 1 à 32 inclus. Ces numéros sont utiles pour communiquer les coups à l'adversaire et pour représenter facilement une case sans devoir passer par les coordonnées 2D. La cellule n°1 est située en haut à gauche et la cellule n°32 en bas à droite. La numérotation va de **gauche à droite, de haut en bas**.

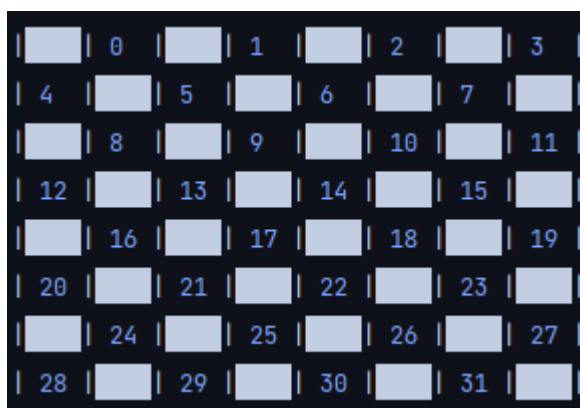


	1		2		3		4
5		6		7		8	
	9		10		11		12
13		14		15		16	
	17		18		19		20
21		22		23		24	
	25		26		27		28
29		30		31		32	

FIGURE 10 – Numérotation des positions du plateau

- La gestion des index :

Si vous avez suivi un peu vos cours d'algorithmique et de programmation, vous savez que le premier indice d'un tableau est 0. Pour coïncider avec cette convention, la notation des tableaux suit les mêmes règles que celle des tableaux mais au lieu de commencer à la cellule 1, la cellule 0 est la cellule en haut à gauche. Un soupçon d'arithmétique vous donnera facilement accès la position de la dernière cellule.



	0		1		2		3
4		5		6		7	
	8		9		10		11
12		13		14		15	
	16		17		18		19
20		21		22		23	
	24		25		26		27
28		29		30		31	

FIGURE 11 – Numérotation des indices des positions du plateau

— Notation FEN :

Tout ce que vous devez savoir sur cette notation a été traité dans la section cours. Cependant, vous devez garder en tête que la notation FEN représente un tableau ligne par ligne, de **bas en haut** !

	29		30		31		32
25		26		27		28	
	21		22		23		24
17		18		19		20	
	13		14		15		16
9		10		11		12	
	5		6		7		8
1		2		3		4	

FIGURE 12 – Ordre d'apparition des cellules en notation FEN (de 1 à 32)

— Le hachage de Zobrist :

Cette notation sera un bonus, mais il est bon de clarifier certaines choses. Alors que les trois notations ci-dessus sont déterministes, la notation Zobrist, par sa nature aléatoire, ne l'est pas. Il est donc possible que deux plateaux différents soient encodés avec le même hachage. De plus, pour des raisons de mémoire, nous ne pouvons pas stocker une quantité infinie de valeur de hachage. Deux hachages différents pourraient pointer vers le même élément du tableau.

### 2.6.3 Des tests à votre disposition

Nous savons que ce TP n'est pas le moins complexe de tous les TP que vous avez effectués cette année. Pour éviter une progression à l'aveugle dans le TP, jusqu'à ce que vous parveniez à lancer votre IA, une série de tests par seuil vous est fournie. **Ces tests ne sont pas exhaustifs.** Ils permettent tout juste de survoler ce que l'on attend de vous, mais ils fournissent un bon indicateur pour savoir si votre fonctionnement correspond ou non au comportement attendu.

Pour exécuter les tests, allez dans le projet *SafetyNet*, ouvrez le fichier correspondant au seuil que vous voulez tester et appuyez sur l'une des flèches vertes à gauche. Vous pouvez choisir d'exécuter un test particulier en appuyant sur la flèche verte à côté de la déclaration de fonction ou d'exécuter tous les tests du seuil en appuyant sur la flèche verte à côté de la définition de classe en haut du fichier.

Vous devriez exécuter ces tests au moins chaque fois que vous terminez un seuil pour vous assurer que tout ce que vous avez fait jusque-là a un sens.

Si vous avez besoin d'aide pour lancer les tests, n'hésitez pas à lire la [documentation JetBrains](#), et si nécessaire demander de l'aide à vos assistants.

#### Attention

Les tests fournis ne testent pas vos fonctions pour l'IA. Cela veut dire que les étapes 3 et 4 sont pas ou partiellement testées.

### 2.6.4 Visualisation du format Dot

Quand vous aurez terminé de coder les fonctions de l'IA, vous rencontrerez sûrement des comportements inattendus de votre algorithme. Heureusement, nous avons conçu un moyen pour vous permettre de voir tous les plateaux que votre IA a pu visiter avec leur valeur heuristique associée. Lorsque vous déboguez en utilisant la classe `DotRunner`, nous vous recommandons de réduire la profondeur maximale de votre IA pour avoir un niveau de performance acceptable. Le nombre de mouvements possibles simulés est exponentiel et vous serez rapidement incapable de comprendre le sens de tous les noeuds.

Le `DotRunner` gère la plupart de la logique pour vous, mais vous devez remplir vous-même la valeur du `DotNode`. Voici un petit guide sur comment l'utiliser :

Tout ce qui est présenté ici fait sens que dans le contexte de `GenerateMove` dans le fichier `MiniMax.cs`. Vous devez toujours vérifier que `root` n'est pas `null`. `root` sera toujours `null` quand votre IA sera appelée en dehors du contexte d'un `DotRunner`.

```
1 // You can assign a board like this
2 if (root is not null)
3     root.Board = FenEncryption.Encrypt(currBoard);
4
5 // A value like this
6 if (root is not null)
7     root.Value = HeuristicFunction.Invoke(currBoard, Color);
8
9 // Create a child like this
10 // If `root` is null, child will be as well. This is ensured by the '?'
11 var child = root?.AddChild();
12
13 // And assign to the child a move like this
14 if (child is not null)
15     child.Move = MoveEncryption.Encrypt(simulatedColor, move);
16
17 // If you have reached the transposition bonus, this is how you assign a
18 // transposed node
19 root.TransposedNode =
20     TranspositionTable.Entries[TranspositionTable.Index(currBoard)].Id;
```

Voici comment lancer le `DotRunner` :

```
1 var blackBrain = new AiProperties(PlayerColor.Black, new Board(), 7,
2     Heuristic.ComputeHeuristic, true);
3 var blackAi = new MiniMax(blackBrain);
4
5 var whiteBrain = new AiProperties(PlayerColor.White, new Board(), 5,
6     Heuristic.ComputeHeuristic, true);
7 var whiteAi = new MiniMax(whiteBrain);
8
9 var runner = new DotRunner(blackAi, whiteAi);
10 runner.Run();
```

Les résultats sont stockés dans le dossier `DotHelper`. Pour utiliser `dot`, nous vous conseillons de lire [la documentation dot](#). Vous pouvez aussi utiliser un outils de visualisation en ligne : [GraphViz online](#).



## 3 Exercices

### 3.1 Etape 1 : découvrez le plateau de jeu

Afin de réduire de manière significative le nombre de fonctions pour ce TP, nous avons décidé de vous donner la plupart des fonctions du moteur de jeu. Cela vous permettra de vous plonger directement dans l'IA sans avoir à vous occuper du jeu en lui-même. Toutefois, on ne peut pas programmer une IA pour un moteur de jeu si l'on ne comprend pas comment fonctionne le plateau pour celui-ci. C'est l'objet de cette première étape : découvrir les principales fonctionnalités du plateau.

Attention : il faut lire le sujet et la documentation attentivement

Beaucoup de code donné implique beaucoup de lecture de votre part ! Assurez-vous de lire **attentivement** la documentation qui vous est fournie et de vous référer au cours pour des exemples appropriés ! Il y a beaucoup de notions différentes à assimiler, vous devez donc lire les fichiers fournis et essayer de comprendre les principaux concepts.

Pour vous faciliter la tâche, voici quelques-uns des éléments que vous devriez vraiment étudier avant de commencer :

- Tous les attributs de classe avec leurs interdépendances.  
Cela inclus : `Board`, `Cell`, `BasePiece`, `Man`, `King`, `PieceManager` et la classe `PathObject`
- Placer des pièces sur le plateau : `Place` dans `BasePiece.cs`
- Pour la génération des chemins : `CheckPathing` et `CreateCellPath` dans `BasePiece.cs`
- Vérification des coups : `CanMove` et `CheckCouldHaveCaptured` dans `BasePiece.cs`
- Les fonctions exécutants les coups : `Move` avec sa surcharge dans `BasePiece.cs` et sa redéfinition `Move` dans `Man.cs`

#### Les dictionnaires

Chaque plateau de jeu contient un `PieceManager`. Celui-ci dispose d'un dictionnaire pour stocker les pièces de chaque joueur. Ceux-ci sont nouveaux pour vous mais il ne sont pas si différents de vos tableaux habituels. En effet, vous êtes normalement habitués aux tableaux indexés à partir de 0. Les dictionnaires fonctionnent d'une manière similaire aux tableaux mais prennent en index des objets. Pour ce TP, vous avez à utiliser un dictionnaire qui est indexé par la couleur du joueur :

```
1 // PieceManager.cs
2 public readonly Dictionary<PlayerColor, List<BasePiece>> PiecesDictionary;
```

Ce qui peut se lire comme étant un dictionnaire associant à couleur (`PlayerColor`) une liste de pièces (`List<BasePiece>`).

Pour accéder à la liste de pièces de chaque joueur du plateau vous pouvez écrire :

```
1 PiecesDictionary[PlayerColor.Black]; // for black player
2 // or
3 PiecesDictionary[PlayerColor.White]; // for white player
4 // etc... it can also be a variable, an attribute in the brackets
```

Pour plus d'information, veuillez consulter [la documentation MSDN](#).

### 3.1.1 Encodage FEN

Avec toutes vos connaissances fraîchement acquises, vous devriez être en mesure d'implémenter assez facilement un encodeur FEN. Veuillez vous reporter au cours pour plus de détails sur le format.

```
1 public static string Encrypt(Board board); // FenEncryption.cs
```

Cette fonction prend un plateau en paramètre que vous devez transformer en code FEN. Pour faire cela, parcourez le plateau, si une pièce se trouve sur la cellule que vous visitez actuellement, encodez cette pièce, sinon incrémentez le compteur de cellules vides. À la fin de la ligne, vous devriez avoir une chaîne de caractères capable de représenter cette même ligne. Il suffit d'ajouter simplement un délimiteur pour pouvoir passer à la ligne suivante.

#### Attention au piège !

La notation FEN d'un plateau et son orientation sont **opposées**. C'est à vous de trouver un moyen de faire coïncider les deux notations. En effet, la notation FEN commence en bas à gauche, de gauche à droite, et bas vers le haut. Tandis que `cells` dans `Board.cs` débute par la cellule en haut à gauche, en sens de lecture de gauche à droite, du bas vers le haut. Pour plus d'exemples, veuillez relire la partie cours.

#### Informations

- Vous n'avez pas à gérer les formats invalides ! Le plateau sera toujours considéré comme valide.
- Nous vous donnons des tests pour cette fonction. Ils ne sont **pas** exhaustifs, ils servent de sécurité. Vous êtes libres d'ajouter les vôtres en suivant notre modèle. <sup>a</sup>
- Une cellule ayant son attribut `CurrentPiece` à `null` est une cellule **vide**.
- Veuillez regarder [la documentation sur le pattern matching](#). Elle pourra vous être utile quand il faudra distinguer les différents types de pièces.
- Pour ce TP, vous avez le droit d'utiliser la bibliothèque `LinQ` dont la documentation est [ici](#). A vous de trouver les fonctions utiles pour ce TP.

<sup>a</sup>. Les tests que vous ajoutez sont personnels. Les partager c'est tricher.

## 3.2 Etape 2 : L'IA avec son plateau de jeu

Cette partie va vous guider dans la connexion de l'IA au plateau.

### 3.2.1 Est-ce déjà la fin ?

L'IA est pour l'instant incapable de savoir si une partie est gagnée ou perdue. Ajoutez cette fonctionnalité en commençant par cette fonction :

```
1 public bool HasWon(PlayerColor playerColor); // Board.cs
```

Pour savoir si un joueur a gagné, vous devez vérifier qu'il a au moins une pièce vivante et que son adversaire n'a plus de pièces.

```
1 public bool HasLost(PlayerColor playerColor); // Board.cs
```

Pour déterminer si un joueur a perdu, vous devez vérifier qu'aucune des pièces du joueur n'a de coup disponible.

```
1 public bool IsEoG(Board board); // AiBase.cs
```

Cette fonction retourne un booléen qui permet de savoir si l'IA a fini sa partie. Plus concrètement, il s'agit de vérifier si le joueur simulé, et uniquement celui-ci, a gagné ou perdu. La couleur pour effectuer cette vérification est stockée dans les attributs du joueur. Gardez à l'esprit que gagner et perdre sont deux conditions distinctes qui ne sont pas liées sémantiquement entre elles.

### 3.2.2 La guerre des clones

Pour faire les différentes simulations, vous devez cloner l'ensemble du plateau. Pour ce faire, vous avez plusieurs fonctions *Copy* à implémenter dans différents fichiers.

```
1 public override Man Copy(); // Man.cs
```

Pour copier un pion, il suffit d'en instancier un nouveau.

```
1 public override King Copy(); // King.cs
```

Pour copier les dames, le processus est identique.

```
1 public Board Copy(); // Board.cs
```

Enfin, vous devriez être capable de copier un plateau entier. Tout d'abord, vous devez créer un nouveau plateau **vide**. Ensuite, vous devez parcourir toutes les cellules du plateau que vous souhaitez cloner. S'il y a une pièce sur une cellule, vous devez la cloner, sinon, passez à la suivante. **N'oubliez pas d'ajouter vos nouvelles pièces au gestionnaire de pièces du plateau cloné et de les placer dessus correctement !**

#### Complexité

Attention, cette fonction est évaluée aussi sur sa complexité. Utiliser les fonctions pour générer et décoder un code FEN ne sont pas des implémentations valides.

### 3.2.3 Transmettre un coup à un autre plateau

Pour cette fonction, vous allez manipuler les `PathObject`. Nous vous recommandons de lire sa documentation associée dans votre squelette.

Un `PathObject` est composé de deux listes :

1. La première contient une liste de `Cell` sur lesquelles une pièce va passer. Ces `Cell` sont les même objets que ceux dans votre `Board`.
2. La seconde contient une liste d'ennemis sur son chemin. Elle peut être vide. Les ennemis ont chacun une référence a la cellule sur laquelle il est.

Ainsi, ces deux listes contiennent des références **aux objets d'un plateau en particulier**.

Un problème se présente donc au moment de la copie des plateaux. En effet, un `PathObject` fait référence aux cases d'un plateau, mais que se passe-t-il si l'on clone un plateau? Le `PathObject` reste associe a l'ancien plateau et est alors désynchronisé si vous essayez de l'utiliser sur une copie. Il faut mettre a jour cet objet pour pouvoir l'utiliser de nouveau.

Donc, pour chaque cellule dans le `PathObject`, vous devez trouver sa jumelle dans le nouveau référentiel (c'est-à-dire la cellule ayant la même position) et ajoutez-la au `PathObject` translaté. Une approche similaire peut être adoptée pour traduire la liste d'ennemis.

Implémentez la fonction `Translate` qui permet de transmettre un coup d'un plateau vers un autre :

```
1 public PathObject Translate(Board targetBoard); // PathObject.cs
```

### 3.3 Etape 3 : le coeur de l'IA

Pour cette avant-dernière étape, vous allez pouvoir implémenter enfin votre IA et commencer à la tester.

Pour pouvoir faire cette partie dans de bonnes conditions, vous devez lire ces classes :

- **AiProperties** : une classe regroupant de nombreuses propriétés pour l'IA.
- **AiBase** : une classe abstraite sur laquelle sont basées vos IA.
- **MiniMax** : la classe dédiée à l'implémentation de votre version de l'algorithme *MiniMax*.
- **Heuristic** : une classe statique contenant les fonctions d'heuristique que vous allez implémenter.

La classe **AiProperties** définit un grand nombre d'attribut. Son constructeur est déjà implémenté pour vous ! Vous ne devriez pas changer quoique ce soit ici. Cependant, si vous décidez d'implémenter des bonus supplémentaires, vous pouvez ajouter de nouveaux attributs.

Similairement, la classe **Heuristic** contient des fonctions déjà déclarées pour vous et une implémentation de l'heuristique aléatoire pour mener des tests. Vous pouvez (et c'est encouragé) ajuster les valeurs du tableau **PositionWeights** et ajouter de nouvelles fonctions d'heuristique. Tout ce que vous ajoutez ici ne sera pas directement évalué lors de la correction. Cependant, faire votre propre fonction d'heuristique peut améliorer votre IA que l'on évalue sur ses performances.

#### 3.3.1 Générer une liste de coups possibles

La famille des algorithmes *MiniMax* repose sur le principe d'explorer tout les chemins possibles pour trouver le meilleur. La fonction **GeneratePaths** une liste de **PathObject**. Ces coups sont utilisés pour générer les différents plateaux possibles et leur score.

```
1  protected static List<PathObject> GeneratePaths(Board currBoard,  
2                                          PlayerColor currPlayer,  
3                                          bool enemyFlag = false)
```

Pour chaque pièce d'un joueur donné, vous devez essayer d'obtenir tous les coups possibles et les rassembler dans une liste. Il est intéressant de noter qu'il existe une fonction toute prête à faire exactement cela pour vous.

#### Attention

- Si vous trouvez un coup pouvant prendre des pièces, tous les autres coups possibles doivent être filtrés pour garder que ceux qui capturent des ennemis.
- Pour voir si un **PathObject** est un coup de capture, vous devriez regarder le nombre d'élément dans la liste d'ennemi du **PathObject**.
- Enfin, souvenez vous que **CheckPathing** a un paramètre optionnel **prevEnemies** dont le sens varie selon sa valeur : *null* ou une liste vide.

### 3.3.2 Heuristique : fonctions par défaut

Implémentez ces fonctions dans la classe `Heuristics` :

```
1 // Heuristic.cs
2 public static double SimpleHeuristic(Board board, PlayerColor currPlayer);
3 public static double ComputeHeuristic(Board board, PlayerColor currPlayer);
4 public static double PositionWeightedHeuristics(Board board,
5                                                    PlayerColor currPlayer);
```

Ces fonctions ont pour but d'additionner la valeur des pièces de chaque joueur et d'appliquer une soustraction aux sommes. Vous trouverez de plus amples instructions dans la documentation fournie.

### 3.3.3 Trouver un coup

Cette fonction est le cœur de notre IA. La fonction `GenerateMove` implémente l'algorithme *MiniMax*. Avec toutes les fonctions implémentées précédemment, vous devriez être en mesure de terminer les bases de l'algorithme en peu de temps.

```
1 // MiniMax.cs
2 protected override Tuple<double, PathObject> GenerateMove(Board currBoard,
3                                                            int currDepth,
4                                                            bool minimize = false,
5                                                            double alpha = Min,
6                                                            double beta = Max,
7                                                            DotNode root = null,
8                                                            bool quietMode = false
9                                                            );
```

Cette fonction renvoie une paire. Le premier élément est la meilleure valeur heuristique des enfants du nœud actuel, l'autre est le *PathObject* qui permettra d'obtenir cette valeur heuristique, c'est-à-dire le déplacement le plus efficace trouvé par votre IA. Veuillez vous référer au cours et à la documentation pour obtenir plus d'informations sur l'implémentation de cette fonction.

#### Conseil

Vous devriez peut-être commencer par l'algorithme *MiniMax* de base, puis introduire l'élagage alpha-bêta sur ce dernier. Il est plus facile d'ajouter des fonctionnalités à partir d'une base saine.

Vous devriez aussi ignorer le paramètre *quietMode* pour le moment.

### Attention

Pour rendre vos IA plus flexibles, la classe `AiBase` contient un attribut `HeuristicFunction`. Il s'agit d'une référence variable à une des fonctions de la classe `Heuristic`. Elles suivent un prototype spécifique qui prend en paramètre un `Board`, la couleur du joueur `PlayerColor` et retourne un `double`.

Vous **devez** utiliser cet attribut pour obtenir la valeur heuristique d'un noeud. Pour ce faire, vous pouvez utiliser la syntaxe suivant :

```
1 // replace with your own variables
2 double val = HeuristicFunction.Invoke(yourBoard, aPlayerColor);
3 // or like a regular function
4 double val = HeuristicFunction(yourBoard, aPlayerColor);
```

Cette manière de stocker et d'utiliser des fonctions reposent sur la notion d'encapsulation de fonction. Pour de plus amples informations sur cette notion, vous pouvez vous référer à la [la documentation MSDN](#).

Cet attribut vous permet de changer facilement la manière dont vous testez vos IA sans avoir à changer tout votre code.<sup>a</sup>

---

a. Vous pouvez vous référer à la partie 3.4.4 pour customizer vos IA

### 3.4 Etape 4 : votre IA contre le monde

#### 3.4.1 Encoder un coup

Comme nous vous le disions précédemment, les déplacements sont représentés par un `PathObject`. Cependant, votre IA doit renvoyer une chaîne de caractères pour représenter un tel déplacement. Ainsi, vous devez encoder cet objet en suivant les règles définies dans la partie cours.

```
1 // MoveEncryption.cs
2 public static string Encrypt(PlayerColor playerColor, PathObject path);
```

Vous pouvez considérer que les valeurs données sont toujours correctes.

#### 3.4.2 Décoder un coup

Vous pouvez générer un mouvement et l'encoder pour le partager avec le monde entier ! Mais ce serait triste de jouer seul. C'est pourquoi nous avons maintenant besoin de décrypter les coups envoyés par l'adversaire.

```
1 // MoveEncryption.cs
2 public static bool Decrypt(Board board, string input);
```

Cette fonction doit en plus de décrypter le coup l'exécuter directement sur le plateau. La fonction renvoie donc un booléen qui indique si le coup a bien été exécuté.

Vous pouvez considérer que les valeurs données seront toujours correctes si la `string` donnée n'est pas `null`. Si `input` est `null`, cela signifie que l'IA est la première à jouer, et qu'aucun coup n'a été effectué avant. Dans ce cas, vous devez renvoyer `true` sans modifier le plateau de jeu.



### 3.4.3 Fonction pilote de l'IA

Implémentez cette fonction :

```
1 // AiBase.cs
2 public string GetThenCreate(string move, DotNode root = null);
```

Elle prend en paramètre un coup adverse et l'exécute sur son plateau. Ensuite, elle demande à l'IA de chercher le coup optimal. Finalement, vous devez encoder le coup et le renvoyer.

Vous pouvez considérer que le coup donné en paramètre est toujours bien formé.

Cette fonction renvoie généralement des coups valides. Cependant, dans certains cas, elle peut renvoyer des chaînes de caractères :

1. "WON" : si l'adversaire a effectué un coup qui est illégal
2. "LOST" : si l'IA n'est pas capable de générer un coup valide. Le plus souvent, cela se traduit par un *PathObject* vide ou *null* selon votre implémentation de l'algorithme.

Vous devez aussi mettre à jour diverses statistiques dans cette fonction :

- Réinitialiser *ExploredPaths* à 0 avant de générer un nouveau coup.
- Mettre à jour et remplacer *ElapsedTime* avec sa nouvelle valeur (pour plus d'informations, les structures *DateTime*<sup>10</sup> et *TimeSpan* vous seront utiles<sup>11</sup>).
- Attribuer à *WithHeuristic* une valeur.

#### Attention !

N'oubliez pas d'effectuer le coup généré par votre IA sur votre plateau de jeu !

#### Les fonctions à paramètres optionnels

Il existe un moyen de transmettre un argument spécifique à l'appel d'une fonction tout en conservant des valeurs par défaut pour les paramètres précédents. Voici un exemple de syntaxe appropriée :

```
1 public bool func(int a, int b = 0, int c = 1, int d = 2)
2 {
3     return a + b + c + d;
4 }
5 Console.WriteLine(func(1, c:4)); // stdout: 7
```

10. Documentation MSDN

11. Documentation MSDN

### 3.4.4 Instantier et tester l'IA

Pour instancier une nouvelle IA, vous devez initialiser un ensemble de propriétés. Une fois fait, vous pouvez les donner au constructeur de l'IA comme ceci :

```
1 var newBrain = new AiProperties((int)PlayerColor.Black, new Board(), 5,  
2                               Heuristic.ComputeHeuristic, true);  
3 var blackAI = new MiniMax(newBrain);
```

Vous pouvez personnaliser les propriétés pour qu'elles correspondent aux capacités de votre IA. Cependant, nous vous conseillons vivement de maintenir la profondeur de recherche entre 5 et 9 pour obtenir des performances acceptables. Vous pouvez également choisir la fonction heuristique de votre choix en utilisant cette syntaxe :

```
1 Heuristic.YourFunction;  
2 // For instance, for the given random heuristic  
3 // You can this function to test your AI's performance  
4 Heuristic.RandomHeuristic;
```

Nous vous avons fourni de nombreux moyens pour tester vos fonctions. Si vous êtes arrivé jusqu'ici, c'est que vous avez passé tous les tests pour toutes les étapes. Si ce n'est pas le cas, corrigez d'abord votre code car les outils suivants ne fonctionneront pas si vous n'avez pas passé les tests fournis.

Si vous avez besoin d'aide pour déboguer votre IA, nous vous suggérons de lancer le **DotRunner** et de jeter un coup d'œil à l'arbre qui a été créé. Vous pouvez trouver plus d'informations sur le visualiseur d'arbres généraux dans la section cours.

Si vous souhaitez tester les performances de votre IA, pensez à utiliser la classe **Runner** avec l'impression sur la console désactivée :

```
1 var runner = new Runner(blackAi, whiteAi, print: false);  
2 runner.Run();
```

Enfin, lorsque vous êtes prêt, vous pouvez soumettre votre IA au test ultime : la référence ACDC. Pour ce faire, lancez un affrontement en mode *leaderboard* sur votre client !

#### ATTENTION

- Nous vous demandons de faire preuve de bon sens. Ne soumettez pas votre IA au classement si vous ne l'avez pas testée au préalable. **Le classement n'est en aucun cas un moyen pour tester votre IA mais plutôt un moyen d'évaluer son efficacité.** Les ressources utilisées pour faire fonctionner le classement sont limitées, et tout abus sera sévèrement puni.
- Afin que votre IA soit correctement instantiée lors de son utilisation par le client, le classement ou lors de la correction de ce TP, il est essentiel que vous mettiez à jour les variables présentes dans la classe **AiTemplate**

#### Encore un peu d'attention s'il vous plaît

Nous tenons à vous rappeler que le classement sur le *leaderboard* n'est pas obligatoire dans ce TP. Il doit être considéré comme un moyen amusant d'évaluer l'efficacité de votre IA par rapport à la référence et à vos camarades de classe, mais rien de plus. Vous ne serez pas notés sur les résultats du *leaderboard* !

### 3.5 La FAQ

Pour vous aider avec vos IA, nous vous avons concocté une petite FAQ :

- Que faire lorsque la partie est finie pour l'un des joueurs ? Vous devez arrêter vos récursion là. Il faut également renvoyer une valeur heuristique qui traduit la victoire ou la défaite de l'IA. Pour cela, les valeurs  $\mp\infty$  pourraient vous servir.
- Que faire en cas de coupure alpha-bêta ? Vous pouvez renvoyer la valeur heuristique du noeud en cours de traitement sans vous soucier de l'exactitude de celle-ci.
- Mon IA met du temps à renvoyer un coup, que faire ? Vous devriez vérifier que votre IA a bien les cas d'arrêts et une profondeur de recherche raisonnable (entre 3 et 7 pour commencer). Vous devriez vérifier vos fonctions *HasWon*, *HasLost* et *IsEoG*.
- Après un coup, le **Board** considère que les coups sont illégaux ou son état n'est pas celui attendu, que faire ? Pour effectuer les simulations de coup, vous devez copier le plateau et ensuite effectuer le coup adapté pour cette copie. Relisez les parties précédentes pour savoir quelles fonctions utiliser. Le **PathObject** contient des références vers des cellules d'un **Board**. Lors de la copie, ces références ne sont plus les mêmes.
- Dois-je toujours retourner un **PathObject** dans ma pair ? Pour économiser de la mémoire, nous vous conseillons de retourner un **PathObject** que pour la première récursion. Ce **PathObject** sera le seul encodé et envoyé à l'adversaire car il s'agit du coup du tour à jouer.
- Comment savoir si mon IA est bonne en local ? Vous pouvez utiliser le **DotRunner** fourni pour lancer une partie entre deux IA. L'une des deux IA devrait avoir une profondeur plus faible ou une heuristique déterminée au hasard. Si cette dernière gagne la partie, c'est généralement que vous avez commis des erreurs dans votre implémentation. Attention, ce test requiert d'avoir fini les autres étapes correctement.

## 4 Bonus

Cette partie contient une liste de bonus que vous pouvez faire. La liste des bonus est très diversifiée, vous devriez donc en choisir un ou deux qui vous intéressent. Ils sont censés améliorer considérablement votre IA. Vous pourrez ainsi découvrir de nouvelles structures de données et des optimisations intéressantes.

Hormis le bonus de la table de transposition, aucun des exercices énumérés ci-dessous ne sera testé. Cependant, ils amélioreront grandement l'efficacité de votre IA, qui sera testée.

Vous trouverez peu ou pas d'aide sur la façon de mettre en œuvre ces bonus. Ils sont là pour vous obliger à lire et comprendre le cours, la documentation écrite pour ce TP, mais aussi la documentation que vous pouvez trouver en ligne !

### Attention

Avant de vous lancer dans toute tentative d'optimisation bonus, faites la partie obligatoire avec la méthode suggérée pour avoir les clefs de compréhension nécessaires pour améliorer votre IA.

### Activer les bonus

Pour activer les bonus, vous devez changer les booléens présents dans la classe `AiTemplate`. Veuillez également inscrire dans votre `README` quels bonus ont été faits.

## 4.1 Cours

### 4.1.1 Les tables de transposition

#### Attention

Ces parties servent d'introduction aux sujets des bonus et ne sont pas complètes pour vous pousser à une réflexion sur votre implémentation !

Comme vous l'avez sûrement remarqué à présent, votre IA peut être particulièrement lente pour effectuer l'ensemble des coups sur une profondeur donnée. De plus, certains de ces coups mènent à une situation identique de celle rencontrée précédemment<sup>12</sup>. Ainsi, pour réduire le nombre de ces doublés, nous vous proposons d'introduire un cache pour stocker le résultat de chacun des coups effectués par l'IA.

Pour cela, il est nécessaire de rassembler des informations dans une structure de donnée :

1. L'état du plateau de jeu
2. Le coup effectué pour y arriver
3. Le résultat associé à ce coup (valeur numérique, profondeur de recherche, le type du noeud avec sa coupure alpha-bêta éventuelle)

Cependant, le nombre de coup augmente très vite. Il faut stocker efficacement chacun des éléments. Le meilleur moyen d'y parvenir est de stocker les données dans une table de hachage. Une table de hachage associe à un objet une clé (un index) pour le stocker et ainsi le retrouver rapidement.

La représentation d'un plateau de jeu en index se fait grâce à l'utilisation d'une fonction de hachage. La plus commune pour le jeu de Dames est la fonction de *Zobrist*. En ce qui concerne son implémentation, nous vous conseillons de regarder la page Wikipedia sur le sujet : [ici](#).

---

12. Lors de la génération d'un coups précédant

#### 4.1.2 Quiescence search (Recherche de sérénité)

Cet ajout à l'algorithme MiniMax permet de rendre votre code encore plus réaliste en quelques sortes. En effet, quand vous simulez des coups et que vous arrivez à un stade qui vous semble très avantageux, vous vérifiez si ce n'est pas un piège en regardant les possibilités suivantes.

Notre algorithme par analogie lui regarde qu'un certain nombre de coup et certains peuvent donc sembler très avantageux alors que l'adversaire pourrait faire un coup meilleur au tour suivant. L'IA est dans l'incapacité de voir cela, c'est l'effet d'horizon.

Pour palier ce défaut, une pratique courante est d'ajouter une recherche supplémentaire lorsque l'algorithme atteint sa profondeur maximale. Cette recherche sur une profondeur très limitée sur certains nœuds permet de voir si un nœud est stable. C'est-à-dire, qu'un nœud qui semble bon, le reste, ou qu'un nœud mauvais reste mauvais.

Au jeu de dames, la stabilité d'un nœud dépend entre autres de la possibilité d'effectuer un coup pouvant manger une pièce. Lancer cette recherche supplémentaire lorsqu'un coup de ce genre est disponible est une manière d'implémenter l'amélioration. Cependant, il faut effectuer divers ajouts pour éviter de faire une recherche en mode *silencieux* qui en lance une autre (une boucle infinie).

Pour illustrer cette partie, voici un exemple simplifié :

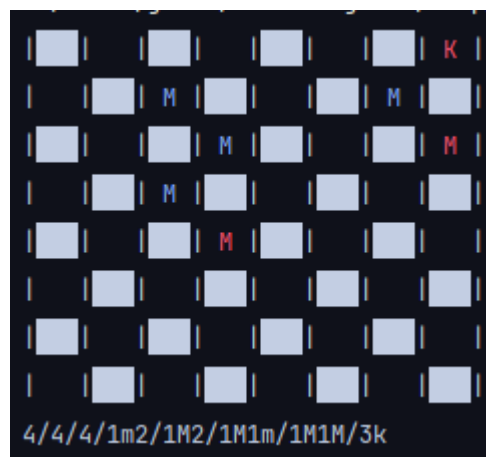


FIGURE 13 – Etat initial pour la dernière recherche de l'IA



## 4.2 Bonus 1 : La sérénité s'offre à vous

Pour ce bonus, vous devez implémenter la *quiescence search*. Vous devez l'ajouter à votre IA MiniMax dans *MiniMax.cs* et utiliser les attributs de l'IA pour l'activer ou désactiver le bonus.

### Conseil

Regardez le booléen `QuiescenceSearch` dans `AiProperties` pour activer ou désactiver le bonus.

## 4.3 Bonus 2 : NegaStonks

Créez la nouvelle classe `NegaMax` héritant de `AiBase` dans le dossier AI. Vous devez implémenter l'algorithme `NegaMax` tel que décrit sur [Wikipedia](https://fr.wikipedia.org/wiki/NegaMax).

## 4.4 Bonus 3 : Table de transposition

Ce bonus est le plus long et assez difficile. Cependant, nous vous avons fourni du code pour commencer plus facilement votre implémentation et nous vous guidons pas-à-pas.

### 4.4.1 Fonction de hachage Zobrist

Vous avez eu l'occasion d'encoder vos plateaux en notation FEN dans la partie obligatoire. Cependant, celle-ci est trop coûteuse en ressource. Il faut donc trouver une alternative pour représenter les plateaux. Vous allez donc implémenter la fonction de hachage de Zobrist. Vous avez dans le cours les liens nécessaires pour la réalisation de l'algorithme.

```
1 // ZobristHashing.cs
2 static ZobristHashing();
3 public static uint CalculateZobristKey(Board board);
4 private static void WriteRandNum();
5 private static uint RandUInt32BitNum();
6 private static Queue<uint> ReadRandNum();
```

Le tableau `PiecesArray` contient des clés que vous pouvez utiliser pour appliquer la fonction *XOR* avec votre résultat. La première colonne de ce tableau représente le type de pièce (0 pour les pions et 1 pour les dames). La deuxième colonne sert d'index pour la couleur des pièces (0 pour les noirs, 1 pour les blancs). Enfin, le dernier index représente la position de la pièce dans le tableau du plateau de jeu.



#### 4.4.2 La classe Entry

Pour stocker l'ensemble des données nécessaire, la classe **Entry** doit être complétée avec son constructeur :

```
1 // TranspositionTable.cs
2 public Entry(int id, uint key, double value, int depth, Status nodeType,
3             PathObject pathObj);
```

Pour prendre en compte les coupures  $\alpha$ - $\beta$ , les noeuds peuvent être classés selon 3 types :

1. Une valeur exacte : quand ils sont stockés dans un contexte normal (non-terminal).
2. Une borne inférieure : si une coupure a eu lieu sur un noeud de maximisation.
3. An borne supérieure : s'il y a une coupure sur un noeud de minimisation.

Les deux derniers types de noeuds mentionnés ont des valeurs approximatives. Leur validité est donc très limitée.

#### 4.4.3 Table de transposition

Premièrement, vous devez implémenter les fonctions suivantes :

```
1 // TranspositionTable.cs
2 public TranspositionTable();
3 public static uint Index(Board board);
4 public PathObject GetStoredMove(Board board);
```

Ensuite, vous devez implémenter la fonction qui stocke vos coups dans la table de transposition :

```
1 // TranspositionTable.cs
2 public void StoreEvaluation(int id, Board board, double eval, int depth,
3                           Status evalType, PathObject move);
```

Pour cela, vous devez créer une **Entry** et la stocker dans la table au bon index.

Ensuite, vous devez implémenter la fonction de recherche d'**Entry** dans la table :

```
1 // TranspositionTable.cs
2 public double LookupEvaluation(Board board, int depth, double alpha,
3                               double beta);
```

S'il existe une entrée correspondant au plateau donné, vous devez vérifier certaines conditions avant de renvoyer sa valeur. Premièrement, une entrée n'est valide que jusqu'à une certaine profondeur. Si cette profondeur n'est pas au moins égale à celle donnée en paramètre, il n'y a pas de cache valide pour ce coup. N'oubliez pas que plus la profondeur actuelle est élevée, plus on se trouve en amont dans l'arbre de recherche. Pas l'inverse !

De plus, vous devez filtrer les valeurs pour les entrées approximatives :

- Une borne supérieure représente une borne pour  $\alpha$  dans l'algorithme.
- Une borne inférieure représente une borne pour  $\beta$  dans l'algorithme.

Si l'entrée n'est pas valide, vous devez renvoyer la valeur de **LookupFailed**.

#### 4.4.4 Intégrer la table de transposition à votre IA

Pour intégrer ce bonus, vous devez comme pour les autres bonus initialiser les propriétés de l'IA pour que l'on puisse activer et désactiver cette fonctionnalité.

Dans MiniMax et ses alternatives, le cache doit être consulté avant d'effectuer un coup pour voir s'il retourne une valeur valide. Dans ce cas, vous pouvez simplement ignorer ce coup et passer au suivant. Il faut également stocker chacune des valeurs retournées dans le cache, un peu comme lorsque vous avez intégré le **DotRunner** à votre IA.

Normalement, si bien exécuté, il ne vous reste plus qu'à lancer votre IA et voir les améliorations de performances données par ce bonus.

## 5 Pour aller vers l'infini et au-delà

Nous avons quelques fonctionnalités supplémentaires intéressantes à implémenter. Cependant, nous ne vous expliquerons rien pour ces bonus.

- Tri des coups : trier la liste des coups possibles du plus intéressant au moins intéressant selon le critère de votre choix afin de forcer une coupure alpha-bêta aussi vite que possible.
- Historique des coups : garder en mémoire les  $n$  derniers coups pour éviter des boucles infinies dans votre IA (qui sont sanctionnées dans le tableau des scores). Ce bonus requiert d'avoir déjà implémenté la fonction de hachage *Zobrist*.
- Retour en arrière : pouvoir revenir sur un état précédent permettrait de grandement optimiser votre IA. En effet, la copie de vos plateaux consomme vite beaucoup de mémoire et de temps CPU. Ainsi, pouvoir garder un plateau unique pour vos simulations est un moyen de réduire le nombre d'opérations coûteuses.

Enfin, ce bonus n'est pas disponible car sa longueur est déjà suffisante, mais une implémentation d'une IA sur un plateau en logique binaire permet une amélioration considérable des performances. On parle ici d'un temps de recherche de coup divisé au moins par deux comparativement à la version orientée-objet. Cependant, l'implémentation d'un tel moteur est longue et peu lisible.

L'ensemble d'optimisation pour ces IA est vaste. Si cela vous intéresse, vous devriez consulter le site [Chess Programming](#).

**There is nothing more deceptive,  
than an obvious fact.**