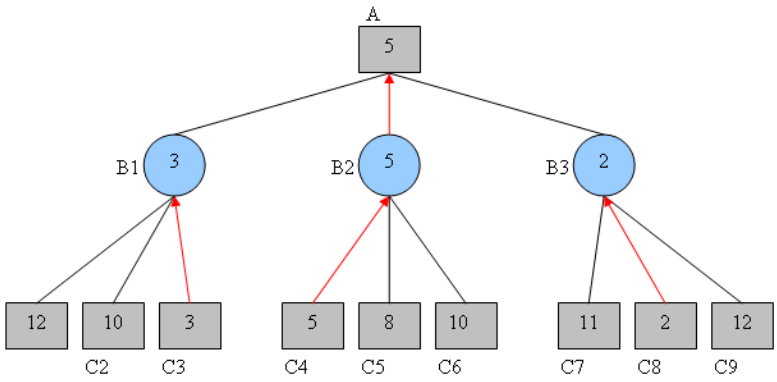
Institut national des sciences appliquées de Rouen

INSA de Rouen

Projet Info GM4 - Vague 1

Algorithme Min-Max



Auteurs :
Thibaut André-Gallis
thibaut.andregallis@insa-rouen.fr
Kévin Gatel
kevin.gatel@insa- rouen.fr
Chloé Mechelinck
chloe.mechelinck@insa-rouen.fr

Enseignants :
Nathalie Chaignaud
nathalie.chaignaud@insa-rouen.fr

04 Janvier 2022

Table des matières

Introduction

Le projet semestriel que nous avons choisi est de créer une intelligence artificielle (IA) grâce à l'algorithme Min-Max.

Le but du projet est de mettre en œuvre les compétences acquises en programmation orientée objet et de nous ouvrir à un domaine tout nouveau pour nous en informatique : l'intelligence artificielle.

Pour ce faire, nous avons décidé d'appliquer l'algorithme à un jeu de Dames et d'implémenter le tout en Java.

Dans un premier temps nous allons expliquer en quoi consiste l'algorithme Min-Max, son principe ainsi que son fonctionnement.

Ensuite, nous allons expliquer la partie implémentation, celle du jeu de Dames puis celle de l'algorithme Min-Max.

Enfin, nous verrons les différents diagrammes UML qui nous ont permis de réaliser ce projet de groupe.

1. Min-Max

1.1 Principe de l'algo

L'algorithme minMax ou miniMax est un algorithme s'appliquant dans le cas d'un jeu (donc dans le cadre de la théorie des jeu) à deux joueurs lorsqu'il s'agit d'un jeu à somme nulle. Son objectif est de minimiser la perte maximum. Un jeu est à somme nulle si la somme des gains et des pertes de tous les joueurs est égale à 0, c'est à dire la perte de l'un est le gain de l'autre. Ce type de jeu répond à plusieurs caractéristiques, démontrées notamment par le théorème du minimax de Von Neumann, dès 1926 (présence de configurations d'équilibre, existence de l'algorithme...).

Le principe est globalement assez simple. L'ordinateur passe en revue tous les possibilités pour chaque pièce sur un nombre limité de coup, créant ainsi un arbre des possibilités (dont nous parlerons d'avantage dans la 4ème partie dans nos exemples). Ensuite chaque noeud se voit affecter une valeur en fonction des bénéfices du joueur et de l'adversaire. Le choix retenu sera la branche partant d'une feuille de cette arbre jusqu'à la racine, indiquant ainsi le coup qui doit être réalisé.

Un problème de mémoire se pose alors car chaque pièce a une liste de coup possible (elle peut se déplacer qu'en diagonale et doit également sauter en diagonale). Si on réalise ce principe sur chaque pièce du joueur (ce qui est nécessaire), l'arbre devient rapidement très vaste. En pratique on explore, lorsque l'algorithme est optimisé, une partie seulement de cet arbre à l'aide de méthodes dites d'élagage.

1.2 Méthode alpha-béta et élagage

L'élagage alpha-bêta est une méthode grâce à laquelle on va pouvoir réduire la taille de l'arbre en enlevant certaines branches à l'aide de conditions choisies. Ainsi, on réduit le nombre de nœuds évalués et donc le temps de calcul de la branche "à choisir". Il s'agit d'une optimisation du minimax sans perdre des informations.

Cet élagage repose sur le fait qu'il n'est pas nécessaire d'examiner les sous arbres dont la configuration et le résultat ne permettra pas une amélioration du gain. On évalue pas les nœuds (et leur sous-arbre) dont la qualité (le gain, l'intérêt) sera inférieur à un nœud déjà évalué. Pour cela on va d'ailleurs travailler sur l'arbre dans un sens fixé, ici de gauche à droite.

2. Jeu de Dames

2.1 Règle du jeu

Afin d'offrir une meilleure expérience, nous laissons libre le choix des paramètres de jeu à l'utilisateur. Ainsi, chaque début de partie l'utilisateur peut choisir s'il veut joueur contre un autre joueur, s'il veut jouer contre une IA ou bien s'il veut regarder jouer deux IA l'une contre l'autre en choisissant leur niveau de jeu. Il peut définir la taille du damier joué, il peut choisir si le saut en arrière est autorisé pour les pions, et également si une pièce doit obligatoirement sauter ou non lorsque cela est possible. Les règles sont ensuite les mêmes pour tout le monde :

Chaque pièce doit se déplacer diagonalement. Un pion se déplace obligatoirement vers l'avant, en diagonale, d'une case sur une case libre de la rangée suivante. Lorsqu'il atteint la dernière rangée, le pion devient Dames. Une Dames doit attendre que l'adversaire ait joué au moins une fois avant d'entrer en action. Une Dames se déplace en arrière ou en avant sur les cases libres successives de la diagonale qu'elle occupe. Elle peut donc se poser, au-delà de cases libres, sur une case libre éloignée.

Lorsqu'un pion se trouve en présence, diagonalement, d'une pièce adverse derrière laquelle se trouve une case libre, il peut sauter par-dessus cette pièce et occuper la case libre. Cette pièce adverse est alors enlevée du damier. Cette opération complète est la prise par un pion. Lorsqu'une Dames se trouve en présence sur la même diagonale, directement ou à distance, d'une pièce adverse derrière laquelle se trouvent une ou plusieurs cases libres, elle peut passer par-dessus cette pièce et occuper, au choix, une des cases libres. Cette pièce est alors enlevée du damier. Cette opération complète est la prise par une Dames.

Lorsqu'au cours d'une prise par un pion, celui-ci se trouve à nouveau en présence, diagonalement, d'une pièce adverse derrière laquelle se trouve une case libre, il doit obligatoirement sauter par-dessus cette seconde pièce, voire d'une troisième et ainsi de suite, et occuper la case libre se trouvant derrière la dernière pièce capturée. Les pièces adverses ainsi capturées sont ensuite enlevées du damier dans l'ordre de la prise. Cette opération complète est une rafle par un pion. Lorsqu'au cours d'une prise par une Dames, celle-ci se trouve à nouveau en présence, sur une même diagonale, d'une pièce adverse derrière laquelle se trouve une ou plusieurs cases libres, elle doit obligatoirement sauter par-dessus cette seconde pièce, voire d'une troisième et ainsi de suite, et occuper au choix une case libre se trouvant derrière et sur la même diagonale que la dernière pièce capturée. Les pièces adverses ainsi capturées sont ensuite enlevées du damier dans l'ordre de la prise. Cette opération complète est une rafle par une Dames. Au cours d'une rafle, il est interdit de passer au-dessus de ses propres pièces. Au cours d'une rafle, il est permis de passer plusieurs fois sur une même case libre mais il est interdit de passer plus d'une fois au-dessus d'une même pièce adverse.

Le jeu se termine lorsqu'un des joueurs ne possèdent plus de pièce ou ne peut plus en bouger aucune. L'adversaire remporte alors la partie.

2.2 Implémentation du jeu

Nous avons donc réalisé notre code en java, langage que nous maîtrisons d'avantage. Nous allons d'abord présenter les classes qui concernent principalement le jeu de Dames :

2.2.1 Case

La classe *Case* représente tout simplement une case du plateau du jeu de Dames. Elle possède donc des coordonnées, une couleur, une pièce qui peut être vide et plusieurs booléens qui ont un rôle dans l'implémentation et la partie graphique du jeu.

2.2.2 Coordonnées

La classe *Coordonnées* possède deux entiers x,y qui représentent juste des coordonnées sur le plateau. Nous avons créé cette classe pour ne pas avoir à transporter deux variables à chaque fois.

2.2.3 Couleur

Cette classe sert à différencier la couleur d'une pièce, d'un tableau de pièce, d'un joueur ou d'une case.

2.2.4 Damier

La classe *Damier* est la classe graphique du jeu de Dames. Elle représente le plateau du jeu. Il contient les tableaux de pièces du jeu, le tableau de cases du jeu, la taille du plateau ainsi que plusieurs booléen jouant un rôle dans les règles du jeu et son déroulement.

2.2.5 Joueur

La classe *Joueur* représente un des deux joueurs du jeu. Elle se décline en deux classes qui héritent de joueur : *IA* et *Humain*. La classe *Humain* n'a pas de différence avec la classe IA si ce n'est qu'il peut choisir son pseudo. En revanche la classe *IA* contient d'avantages de méthodes, notamment la méthode *Min-Max* et *tourOrdi* qui seront expliquées ultérieurement. On retrouve également la procédure *AJoue* qui simule le coup d'un joueur dans le jeu, ainsi que *APerdu* qui retourne vrai ou faux selon si le joueur a perdu ou gagné la partie.

2.2.6 Lanceur

Cette classe constitue le *Main* de notre programme, il est possible d'y modifier certains paramètre de la partie manuellement (selon si le menu est activé ou non), sinon nous utilisons une autre classe que nous allons voir pour laisser le joueur décider des règles du jeu.

2.2.7 Menu

La classe *Menu* permet de laisser l'utilisateur choisir ses règles pour la partie. Elle consiste simplement en un menu à choix multiples que l'on récupère et que l'on transmet au *main* pour lancer la partie.

2.2.8 Piece

La classe *Piece* correspond à une pièce du jeu. Elle se décompose en deux classes qui héritent de celle-ci : Pion et Reine, qui sont les deux types différents de pièces qu'on retrouve dans le jeu de Dames. Dans le fonctionnement de notre jeu, c'est au niveau de la pièce que l'on vérifie si elle peut être mangée ou non, si elle peut manger une autre pièce ou non ainsi qu'afficher les déplacements possibles de la pièce sur le damier qu'elle contient directement.

2.2.9 TableauPiece

La classe *TableauPiece* gère la liste des pièces d'un joueur, d'un camp comme de l'autre.

2.2.10 Souris

La classe *Souris* sert à récupérer les actions de la souris pour permettre au joueur d'utiliser une souris pour jouer au lieu du clavier. Elle permet essentiellement de récupérer les coordonnées de la case sur laquelle le joueur clique.

2.3 Implémentation de la partie IA

Nous allons maintenant nous intéresser d'avantage à la partie IA du projet. Nous allons détailler les classes du programme qui jouent un rôle dans l'IA basée sur l'algorithme Min-Max ainsi que certaines procédures importantes. L'algorithme est appliqué à un arbre de nœuds, expliquons alors ce qu'est un nœud dans notre cas de figure, puis expliquons la création de l'arbre petit à petit :

2.3.1 NoeudDame

La classe *NoeudDame* représente un "nœud" dans le cadre de l'algorithme Min-Max. Elle hérite de la classe abstraite *Nœud* qui contient seulement la signature de la méthode *Heuristique* que l'on détaillera dans la suite.

Nous avons défini la "valeur" du nœud comme un *Damier* du jeu de Dames. Ainsi, on pourra lui attribuer un certain "poids" grâce à la méthode *Heuristique*.

Un objet *NoeudDame* possède une liste de *NoeudDame* correspondant aux successeurs du nœud, on les appellera également ses fils. La classe possède un entier profondeur permettant de savoir à quelle profondeur le nœud se situe dans l'arbre, la profondeur zéro étant celle de la racine.

Détaillons maintenant une des méthodes les plus importantes pour notre IA appartenant à cette classe : la méthode *Heuristique*.

L'heuristique

...

2.3.2 Arbre

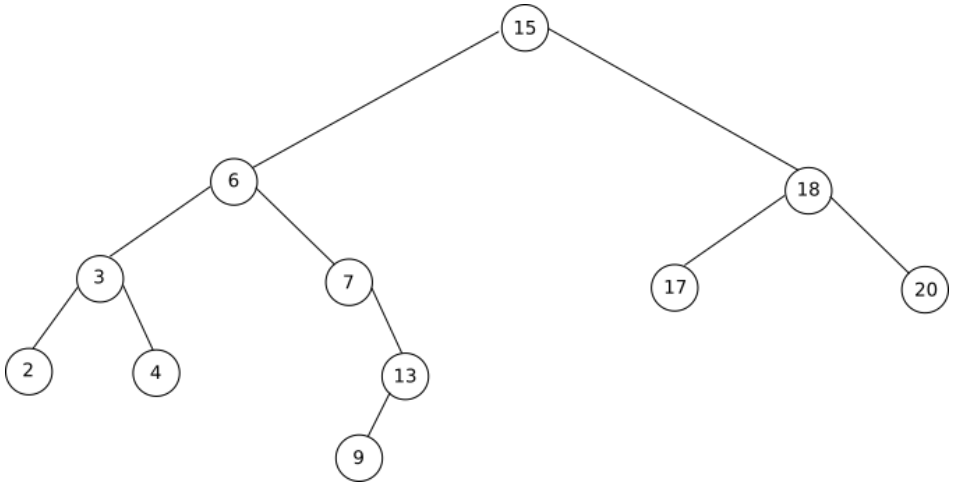


FIGURE 2.1 – Exemple d'un arbre quelconque

La classe *Arbre* représente l'arbre sur lequel se base notre algorithme Min-Max. Dans l'exemple du dessus cet arbre est binaire, le notre ne l'est absolument pas puisque chaque branche représente un damier possible que le joueur peut créer en jouant une de ses pièces. Chaque rond représente un nœud et la valeur à l'intérieur correspond à la valeur du nœud. Cette valeur est le "poids" du damier appartenant au nœud, c'est le résultat de la méthode *Heuristique*.

La classe contient en attribut un nœud qui est en faite le nœud racine de l'arbre. Il est de profondeur 0. On accède au reste de l'arbre grâce à la liste de successeurs que le nœud racine possède. La classe comprend également la profondeur de l'arbre correspondant en pratique au nombre de coup à l'avance que l'IA peut anticiper. Nous allons maintenant détailler la création de l'arbre grâce aux trois méthodes suivantes.

ListeDesCoupsPossibles : Coordonnees[]

Cette méthode permet de retourner un tableau de *Coordonnees*. Chaque coordonnée du tableau correspond à un déplacement possible de la pièce mis en paramètre de la méthode. Ainsi, la méthode retourne l'ensemble des "coups" possibles à partir d'une pièce.

GenererArbreParPiece : void

Cette méthode permet à partir d'un nœud_0 de créer un fils nœud_1 puis de l'ajouter à la liste des successeurs du nœud_0. Elle va en réalité évaluer le déplacement de la pièce grâce à la méthode *ListeDesCoupsPossibles*. Pour chaque valeur du tableau de coordonnées elle va générer un nœud de profondeur +1 avec le damier comprenant le déplacement de la pièce. Si la pièce vient de manger une pièce adverse et peut en manger au moins une autre, alors cette même méthode est appelé sur ce nœud qui vient d'être créé et ainsi de suite (récursivité). Une fois que la pièce ne peut plus manger, si le nœud est une feuille de l'arbre alors ce nœud est ajouté à la liste des successeurs du nœud entré en paramètre de la méthode *GenererArbreParPiece*, sinon on appelle la méthode *GenererArbreParNoeud* (détaillée juste après) sur ce nœud généré pour enfin l'ajouter au nœud entré en paramètre de la méthode *GenererArbreParPiece*.

GenererArbreParNoeud : void

Cette méthode est la méthode principale de la création de l'arbre. Elle est appelée sur chaque nœud dont nous voulons connaître leur fils. Elle est notamment appelée sur l'attribut *racine* de la classe *Arbre* qui est un *NoeudDame*. La méthode va en réalité, selon la profondeur du nœud, appeler pour chaque pièce du tableau de pièce du damier la méthode *GenererArbreParPiece*. Elle permet simplement de découper le problème "en petits morceaux" pièce par pièce. C'est uniquement la méthode *GenererArbreParPiece* qui va ajouter des successeurs au nœud. Les deux méthodes s'appellent mutuellement jusqu'à ce que le nœud soit une feuille de l'arbre.

2.3.3 Coup

La classe *Coup* représente un coup pour une pièce du jeu. Elle possède deux pièces : celle au départ (avant déplacement) du coup et celle à la fin (après le coup). Évidemment ce qui va changer entre les deux pièces sont les coordonnées de la pièce et le statut (un pion peut devenir une reine).

2.3.4 Resultat_minMax

Cette classe a été créée pour exploiter le résultat retourné par notre méthode min-max. Elle n'a pas de signification particulière elle est juste une manière pratique de transporter les données dans ce cas. Pour résumer, elle comprend la valeur d'un noeud et également la liste de coups associée à cette valeur. C'est une classe pratique pour notre implémentation.

2.3.5 IA

La classe *IA* hérite de la classe *Joueur* et représente un ordi doté d'une intelligence artificielle. C'est dans cette classe que l'on retrouve l'algorithme Min-Max que nous allons détailler ci-dessous. Tout d'abord nous allons écrire en pseudo-code la méthode *minMax*, puis nous allons expliciter son fonctionnement. Nous verrons ensuite la méthode *tourOrdiIA*, son pseudo-code et son explication.

minMax : Resultat_minMax

Algorithm 1 minMax(E : NoeudDame noeud ; entier profondeurArbre) : Resultat_minMax

```
if noeud est une feuille then
    retourner l'heuristique du nœud
end if
if profondeur du fils est paire then
    valeur ← -∞
    for chaque fils de noeud do
        temporaire ← minMax(fils,ProfondeurArbre,)
        if temporaire>valeur then
            valeur ← temporaire
        end if
    end for
else
    valeur ← ∞
    temporaire ← minMax(fils,ProfondeurArbre,)
    if temporaire<valeur then
        valeur ← temporaire
    end if
end if
```

L'algorithme minMax à partir d'un arbre dont les nœuds sont des damiers du jeu de Dames doit ressortir le meilleur coup à jouer pour l'ordinateur.

On part de la racine qui représente le damier actuel, chacun des nœuds fils d'un nœud représente une situation possible avec un coup d'écart avec le nœud père. La profondeur de l'arbre qu'on explore va avoir un lien direct avec la difficulté de l'ordinateur. Plus l'arbre est profond plus l'ordinateur fera son choix en fonction des conséquences à plus long termes. Autrement dit la profondeur de l'arbre correspond au nombre de coup d'avance avec lequel joue l'ordinateur.

L'algorithme consiste à évaluer les feuilles de l'arbre grâce à la méthode *Heuristique* en leur donnant une valeur représentative de l'intérêt de la situation pour le joueur. Ensuite en fonction de si le niveau auquel on se situe est paire ou impaire on remonte la valeur maximal ou minimal des fils au nœud père. A la fin on obtient le coup qui maximise l'intérêt pour l'ordinateur tout en minimisant l'intérêt pour l'adversaire. Plus la profondeur de l'arbre va être grande plus on va pouvoir voir cela à long terme.

L'algorithme semble parfait et imbattable si on pousse la profondeur au maximum, cependant le coût de calcul et la consommation de mémoire est exponentielle et on arrive rapidement aux limites de l'ordinateur même pour un jeu de Dames. Il existe donc certaines méthodes pour réduire le coût de calcul comme l'élagage alpha-bêta que nous n'avons pas eu le temps de mettre en place.

L'autre faille importante est que la méthode *Heuristique* ne peut être parfaite et résulte entièrement de l'interprétation humaine. Intéressons nous maintenant à la méthode *tourOrdiIA*.

tourOrdiIA : void

La méthode *tourOrdiIA* représente le déroulement du tour de l'IA, concrètement ce que fait l'IA lorsque c'est à elle de jouer. On mettra ci-dessous le pseudo-code de la méthode puis nous allons expliquer en quoi elle consiste :

Algorithm 2 tourOrdiIA(E : int difficulte ; boolean tourBlanc) : void

```
Variables : Arbre arbre, ArrayList<Coup> meilleurCoup, Entier indice,x,y

arbre ← Arbre(difficulte,this.couleur,this.damier,this.peutMangerEnArriere,this.obligerLesSauts)
meilleurCoup ← minMax(arbre,tourBlanc,this.peutMangerEnArriere,this.obligerLesSauts).getListeDeCoup()
indice ← 0
x ← meilleurCoup.get(0).getPieceAvantD().getC().X()
y ← meilleurCoup.get(0).getPieceAvantD().getC().Y()
this.Ajoue(x,y,tourBlanc,this.peutMangerEnArriere, this.obligerLesSauts)
while indice < taille du tableau meilleurCoup do
    x ← meilleurCoup.get(indice).getPieceApreD().getC().X()
    y ← meilleurCoup.get(indice).getPieceApresD().getC().Y()
    this.Ajoue(x,y,tourBlanc,this.peutMangerEnArriere,this.obligerLesSauts) ;
    indice ← indice+1 ;
end while
```

Dans un premier temps l'arbre est généré avec une profondeur *difficulte*. C'est ainsi qu'on règle la difficulté de l'IA, il s'agit en effet de la profondeur de l'arbre. Concrètement, la difficulté de l'IA se résume au nombre de coups d'avance qu'elle a sur le damier actuel. Ensuite le *meilleurCoup* est récupéré grâce à la méthode *minMax* où l'on va appeler l'accesseur *getListeDeCoup* pour ainsi récupérer le tableau dynamique du ou des meilleur.s coup.s à jouer. On va alors "simuler" un clique sur la pièce

initiale en récupérant ses coordonnées puis cliquer sur la liste des coups à jouer en utilisant le tableau de coups *meilleurCoup*. Ce dernier est de dimension 1 lorsqu'il y a un déplacement sans prise de pièce, il sera de dimension supérieure s'il est dans une situation de saut multiple.

2.4 Application de la méthode à des exemples

Pour mieux comprendre cette méthode et les différents cas possibles, voici plusieurs exemples (simplifier car on a choisit de représenter deux pions sur plusieurs coups en ayant attribué des valeurs aux noeuds).

3. UML

3.1 diagramme de cas d'utilisation

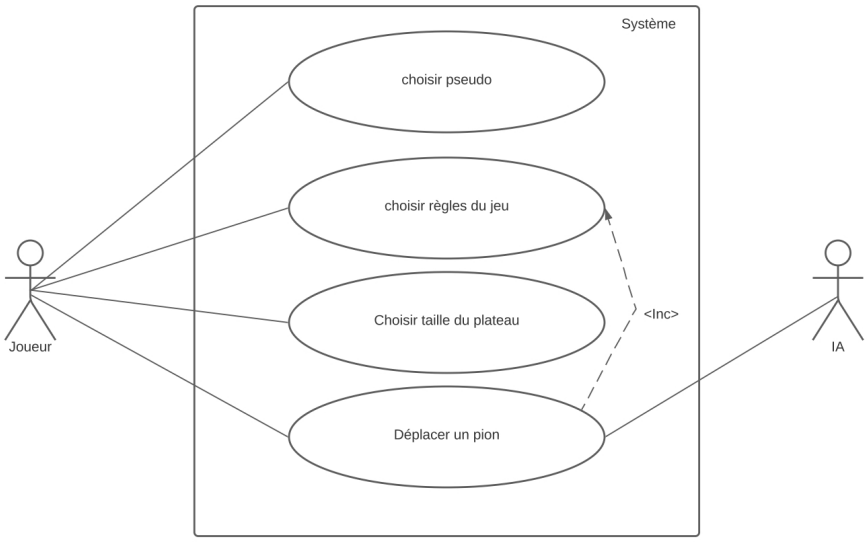


FIGURE 3.1 – Diagramme use case du jeu de Dames

L'utilisateur peut ainsi choisir de jouer à 2, de jouer contre l'ordinateur ou afficher les règles, stockées dans un fichier texte extérieur.

3.2 diagramme de séquences

Nous avons réalisé plusieurs diagrammes de séquences pour détailler la partie algorithme Min-max. Les voici :

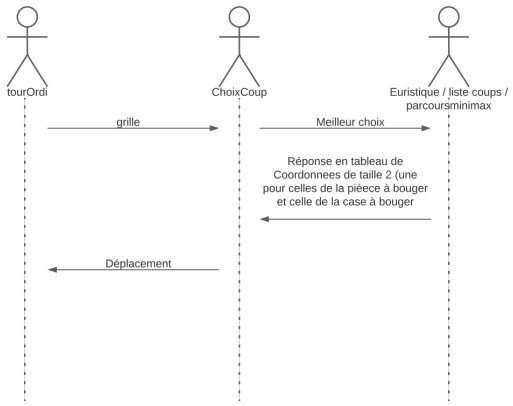


FIGURE 3.2 – Diagramme de séquence choix coup ordi

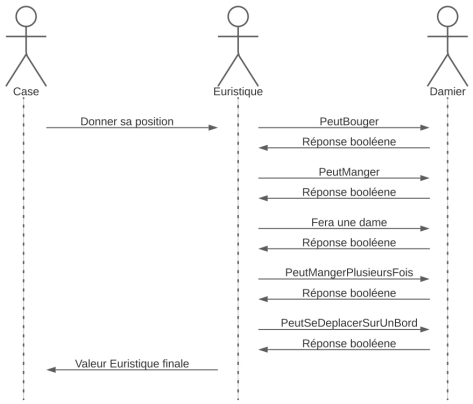


FIGURE 3.3 – Diagramme de séquence heuristique

3.3 diagramme de classes

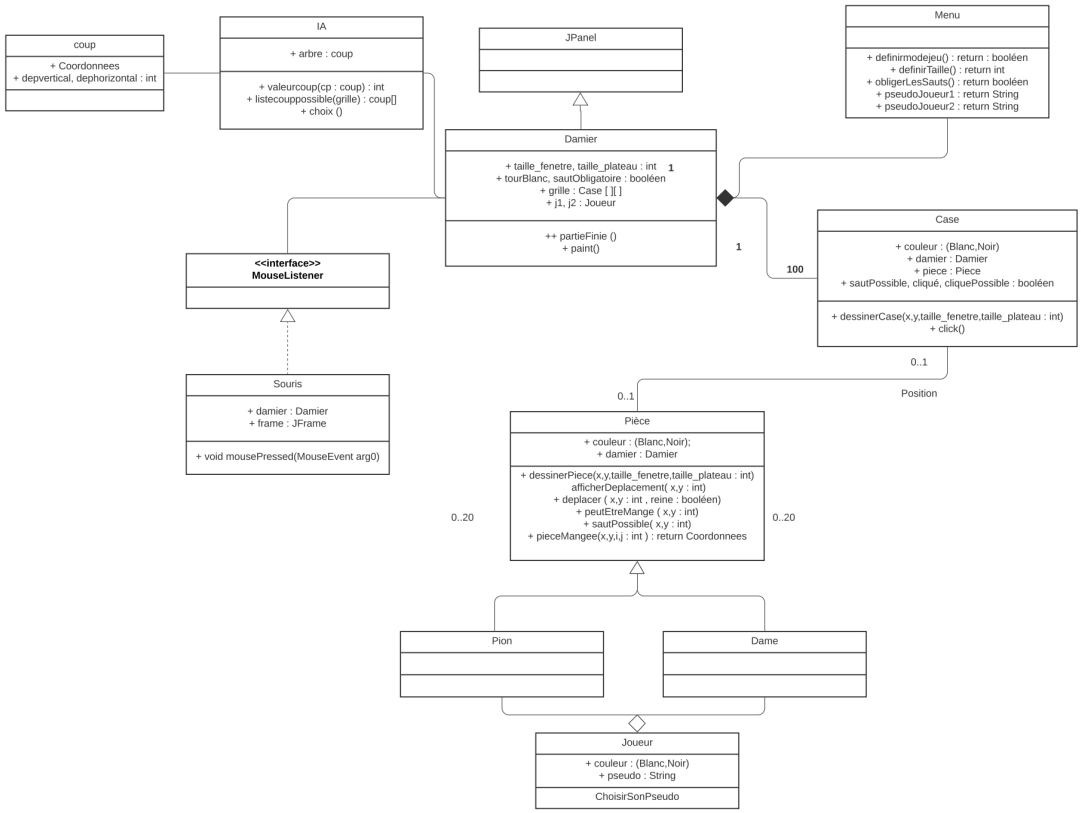


FIGURE 3.4 – Diagramme des classes

Voici le diagramme de classe de notre projet. Celui-ci est susceptible d'évoluer jusqu'à la fin de notre code, il se sépare en deux parties : la première pour l'implémentation du jeu de Dames la seconde pour celle de l'algorithme Minimax.

Un certain nombre de fonction seront détaillées dans les parties suivantes.

4. Problèmes rencontrés

Le premier problème rencontré est lors de l'ajout d'un joueur IA. Notre programme n'était pas du tout destiné à jouer automatiquement. Le programme devait attendre un "clique" souris pour continuer à s'exécuter. Ne pouvant pas simuler un "clique souris" il nous a fallu revoir tout le corps du *Main* ainsi que de nombreuses méthodes.

Le second est lors de la création de l'arbre. La manipulation d'objet étant en réalité une manipulation d'adresse, lorsque nous voulions travailler sur une copie du damier en affectant directement un damier à un autre, nous nous retrouvions à modifier le damier de base, alors que ce n'était pas le but. Grâce à l'utilisation d'une classe implémentée *Cloneable*, cela nous a permis de "cloner" nos objets, de sorte à les copier et ne modifier que les copies de manière à laisser intact les originaux. Une fois cela compris la profondeur 1 a été plus ou moins facile à faire. En effet, lors des sauts multiples il a fallu faire appel à une récursivité de loin triviale. Les problèmes se compliquent pour générer un arbre de dimension n. En effet, la gestion de couleur des pièces change, et nous remarquons que nous devons appliquer une récursivité non pas seulement sur les pièces mais sur chaque nœud. Une réorganisation du code avec la création de méthodes fût nécessaire afin de permettre le bon déroulement de la méthode. Il a également fallu trouver un moyen d'afficher le damier des nœuds d'une certaine profondeur de manière à pouvoir observer et remarquer les éventuels oublis. La création de l'arbre reste de loin le problème le plus dur rencontré dans ce projet.

Le troisième problème rencontré est lors de la méthode *minMax*. L'originale consiste à seulement renvoyer la valeur de la feuille de l'arbre qui maximise les coups du joueur tout en minimisant ceux des adversaires, alors que ce qui nous intéresse vraiment c'est le meilleur coup à partir du damier actuel. Il a fallu adapter un peu l'algorithme original pour ainsi renvoyer la valeur des nœuds et la meilleure liste de coups.

Enfin le dernier problème rencontrer est le choix de l'heuristique. Une infinité peut en être créée. Quelles situations privilégier ? Pourquoi une plus qu'une autre ? Un mini débat s'est installé naturellement, mais nous avons fini par en choisir une. Notre choix s'est porté sur celle qui nous semble la plus adapté au jeu de Dames selon nous.

Conclusion :

Cette première partie nous a permis de nous renseigner sur la théorie des jeux, les spécificité du jeu de Dames (les différents types de positions des pions...). Nous avons aussi commencé à coder après avoir réalisé nos diagrammes en essayant de rendre notre code le plus portable possible, tout cela en travaillant en équipe à trois. Nous allons poursuivre le codage de notre programme, tout en réalisant certaines parties en pseudo-code d'abord car la fonction euristique est très importante mais aussi assez importante.

Pour aller plus loin :

Comme dit précédemment nous ne sommes qu'au milieu de notre projet et nous n'avons donc pas fini les fonctions de la partie implémentation du mininmax.

Nous aimerions réaliser la fonction élagage, ainsi que plusieurs euristiques que nous comparerions pour obtenir la meilleur (il s'agit de la difficulté à laquelle sont confrontées les personnes ayant travaillé sur cet algorithme etc : choisir la bonne euristique).

Si le temps qu'il nous reste nous le permet nous aimerions pouvoir afficher à la demande du joueur, l'arbre des possiblités de sa composition actuelle et la branche choisie.

Nous voulons aussi réaliser un menu pour notre jeu avant le début de la partie où l'on peut choisir de jouer à deux, seul contre l'ordinateur ou bien d'afficher les règles.

Liste des algorithmes

Table des figures

Sources

Règle du jeu de Dames officielle : `http://www.ffjd.fr/Web/index.php?page=reglesdujeu#top`