Lyes Meghara , Marine Belot , Solaina Cassamaly & Caroline Dakouré

# Licence 2 Informatique - 2015/2016


# Rapport de projet TER-L2 - GOMOKU -

## Encadré par : Mr *Lionel Ramadier*

24 avril 2016

# Table des matières

**Nous tenons à remercier Mr *Lionel Ramadier* qui a supervisé notre projet tout au long du semestre. Ses nombreux conseils nous ont permis de mener à bien notre jeu.**

# I Introduction :

Dans le cadre de notre parcours en informatique, il nous a été demandé de choisir un projet dans l'UE HLIN405. Notre choix s'est porté sur le jeu du Gomoku, l'objectif était de réaliser le jeu en langage orienté objet (C++. La programmation orientée objet simplifie l'organisation du code et facilite sa réutilisation. Grâce à cela nous avons découvert pour la première fois l'interface graphique ainsi que des algorithmes sur l'intelligence artificielle.
Après avoir obtenu l'accord de notre encadrant Mr ***Lionel Ramadier***, nous avons décidé de coder une variante officielle du Gomoku qui est le Renju, ce-dernier se joue comme le Gomoku mais sur une grille de 15x15 ce qui permet à l'IA d'évaluer moins de possibilités. Pour l'interface graphique, nous avons suivi les conseils de notre encadrant et choisi d'utiliser SFML sous l'IDE Code : :Blocks. SFML est une librairie orientée objet qui est très bien documentée.
Nous avons vu deux algorithmes traitant de l'intelligence artificielle : le *min-max* et *l'alpha-beta*
Dans l'optique d'apprendre un nouveau langage qui nous sera utile par la suite, nous avons décidé de faire notre rapport sur ***LaTex***.

# II Le Renju :

Nous avons trouvé plus simple de réaliser cette variante du gomoku car premièrement la taille de la grille est réduite. Une grille de Renju est de taille 15x15 contrairement à celle du gomoku qui est de taille 19x19. Le Renju se joue sur le même principe que le gomoku. En début de partie, chaque joueur possède 60 pions de couleur (noir ou blanc), le but est que l'un des 2 joueurs aligne 5 pions de la même couleur pour gagner la partie. Si chacun des joueurs vient à poser ses 60 pions sur la grille sans pour autant qu'il n'y ait de vainqueur potentiel alors on déclare la partie nulle.

L'ouverture du jeu diffère également du gomoku, le renju est composé d'une séquence de règles d'ouverture de jeu qui ne sont pas présentes dans le Gomoku. Par manque de temps, nous n'avons pas pu toutes les implémenter.

Nous avons choisi d'implémenter les 2 premières règles de la séquence d'ouverture, c'est à dire :

1. Le premier joueur place deux pierres noires et une pierre blanche sur le plateau.
2. Le second joueur choisit alors de savoir s'il veut jouer en noir ou blanc.

Il nous a semblé logique de suivre le déroulement de la séquence d'ouverture pour l'implémentation des règles.

### Pourquoi SFML ?

Une multitude de bibliothèques sont mises à disposition afin de concevoir des jeux vidéo, il n'a donc pas été facile d'en choisir une parmi les autres.
Après maintes réflexions notre choix s'est porté sur la SFML qui est une bibliothèque spécialisée pour les programmes en 2D. Comme c'était nouveau pour nous, il a fallu apprendre à l'utiliser.
Nous avons choisi la SFML plutôt que la SDL car la SFML est codée en C++ et tous les modules pour gérer les images, l'audio, les évènements - et bien d'autres - sont regroupés en une seule bibliothèque plutôt qu'éclatés en plusieurs parties, ce qui fait qu'elle est très facile à prendre en main.
La SFML a aussi l'avantage d'être multiplateforme, elle fonctionne sur Windows, Linux et Mac OS X. Aussi, nous avons pu installer SFML sur nos ordinateurs à la fac en faisant la liaison dynamique avec Code : :Blocks.

Exporter le chemin manuellement était nécessaire car nous n'avions pas l'accès root.
Pour conclure, la SFML est fréquemment mise à jour et dispose d'une communauté dynamique ainsi que d'un forum francophone.

⇨ Nous détaillerons les différentes parties de notre jeu avec intelligence artificielle tout au long du rapport.

# III Joueur contre Joueur :

## III.1 Joueur

Nous avons créé une classe « Joueur »afin de définir le login du joueur, ses scores.

**➢ Attributs :**

Il était indispensable de définir un attribut « Score_j »qui correspond au score réalisé par le joueur après un nombre de parties non défini.

**➢ Méthodes :**

Nous avons créé des accesseurs en lecture et écriture pour le pseudo et le score du joueur.
– l'accesseur en lecture « GetPseudo »nous est utile pour les méthodes « qui_commence »« draw_score »de la classe « Menu », ainsi que la méthode « printvictoire »de la classe « Grille ».
Il nous permet de lire le pseudo du joueur pour ensuite l'afficher à l'écran.
– accesseur en écriture « SetPseudo »permet lors de la saisie au clavier du pseudo du joueur « Entrer_Pseudo »de stocker le login.

Nous avons créé une méthode « AjoutScore »qui permet d'incrémenter le score du joueur.

**➢ Les logins :**

Afin que les joueurs puissent entrer leur login, nous avons mis en place une méthode « Entrer_Pseudo »qui lors de l'exécution du jeu ouvre une nouvelle fenêtre (avant que la partie commence) qui demande au(x) joueur(s) de saisir leur/son pseudo(s) au clavier.
Une erreur survient lorsque le joueur efface avec la touche back la dernière lettre qu'il vient de saisir, on constate un problème d'affichage : un espace inutile vient séparer le mot saisi précédemment et la suite du mot que l'utilisateur continue de saisir.

**➢ Les scores :**

La méthode « Meilleur_score() »permet de calculer les 3 meilleurs scores de toutes les parties et renvoie le maximum de ces scores.
Afin de sauvegarder les scores, la création du fichier « scores.txt »a été necessaire.
La méthode fonctionne de la manière suivante : grâce à un flux d'entrée on récupère les 3 scores que l'on stocke dans 3 variables différentes, puis on compare chacune de ces valeurs avec le score réalisé par les joueurs à la dernière partie.
Si les résultats ont dépassé certains scores contenus dans le fichier « score.txt », alors on ouvre le fichier en flux de sortie puis on procède à un remplacement des valeurs (mais qui conserve tout de même le classement).
Exemple : un joueur réalise un score de 23, à ce moment-là le fichier score contient : 15, 6, 3. Alors le nouveau fichier « score.txt »contiendra : 23, 15, 6.

➢ **Reprise de partie :**

On a également souhaité qu'un joueur puisse poursuivre une partie qu'il avait démarré à un autre moment. Nous avons donc crée une méthode « Enregistre_score() »qui stocke le login du joueur et son score actuel dans un fichier « partie prec.txt ». Elle utilise le flux de sortie pour stocker les données dans un fichier où nous enregistrons les 2 logins et les 2 scores des 2 joueurs. Nous avons dû mettre le flux de sortie en ios : :app, afin que les données du deuxième joueur soient ajoutées à la suite du fichier (pour ne pas qu'il y ait de confusion lors de la réutilisation du fichier « partie_prec.txt »dans la méthode « Partie_Précédente() »contenue dans la classe Menu).

Pour que les joueurs puissent voir les scores en cours de partie, la méthode « Print_score »a été créé. Elle permet d'afficher le login du joueur et son score actuel.
Cependant lors de la conception de cette méthode, nous avons rencontré un problème :
lorsque l'on affichait le score du joueur à l'écran, son score apparaissait sous forme de code.
Nous avons dû créer une méthode « intConvertString »qui nous a permis de convertir les types entiers du score en String afin que le score du joueur soit réellement affiché à l'écran.

## III.2 Menu

Cette classe est responsable de l'affichage graphique du menu principal, elle propose divers choix ; le joueur peut naviguer avec les flèches de son clavier, une fois son choix fait il a juste à appuyer sur la touche ENTREE

➢ **Mode 1V1 : Nouvelle partie/Poursuivre l'ancienne partie :**

Concernant la méthode Joueur1 VS Joueur2, nous avons fait en sorte qu'après avoir sélectionné son type de partie, une nouvelle fenêtre s'ouvre demandant au joueur s'il souhaite poursuivre ou non la partie précédente, le joueur sélectionne son choix en cliquant sur oui ou non.
Si le joueur clique sur oui, on ouvre le fichier « partie_prec.txt »en lecture pour récuperer les 2 noms et les 2 scores des différents joueurs, puis on stocke ces valeurs dans des variables et on les utilise avec les accesseurs en écriture « Set_Pseudo() »et « Set_Score() ».
Le joueur commencera sa partie avec les scores précédents. Si le joueur clique sur non, on efface le contenu du fichier « partie_prec.txt »avec le flux de sortie avec l'option ios : :trunc, la fênetre pour entrer les logins s'ouvre alors et les scores sont mis à zero.

➢ **Options :**

L'élément option se compose d'une fenêtre avec des images pour pouvoir activer, désactiver et changer le volume du son. Le plus dur a été de réussir à lier le clic de la souris à l'image.
Pour cela, on a fait une méthode « spriteClick »qui définit un rectangle autour de l'image et qui renvoie vrai si on a cliqué dans le rectangle. Nous avons également rajouté ces images sur la fenêtre du jeu pour pouvoir modifier le son pendant une partie.

➢ **Les évenements :**

Afin de gérer l'évènement qui permet la navigation avec le clavier, on a choisi de faire une seule boucle d'évènement dans le « main »plutôt que plusieurs boucles d'évènements dans les différentes méthodes de la classe Menu. Cela nous a permis d'alléger un peu le programme

➢ **Autres éléments du menu :**

En passant par le menu, on peut aussi voir les règles du jeu qui apparaîtront dans une nouvelle fenêtre, consulter la liste des meilleurs scores qui a été stockée dans un fichier texte ou quitter le jeu.


FIGURE 1 – Menu

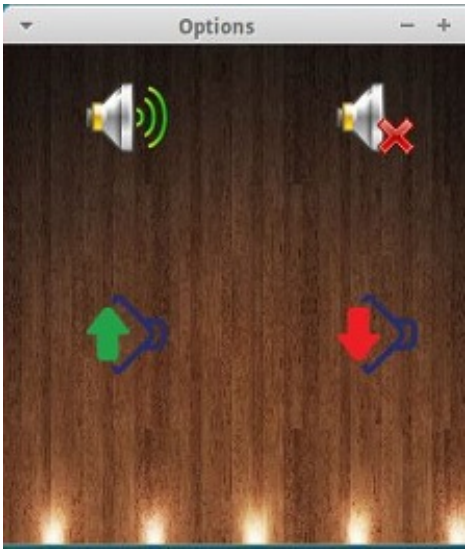



FIGURE 2 – Options

On a également rajouté une rubrique « Un Joueur »qui permet de jouer contre l'IA. Pour ce faire, on a utilisé l'algorithme min-max qu'on a par la suite amélioré avec l'algorithme alpha-bêta pour avoir une meilleure complexité et diminuer le temps d'exécution du programme.
Aussi, à chaque fois qu'un mode de jeu est sélectionné, on nettoie la fenêtre et on l'agrandit pour l'affichage de la grille.

## III.3 Curseur

Cette classe est responsable de l'interaction entre le joueur humain et l'interface graphique de la grille.

**➢mouseButton/getPosition :**

Premiers choix pour détecter le clic de la souris :

sf : :Mouse : :isButtonPressed( sf : :Mouse : :Left )

*et*

event.mouseButton.button == sf : :Mouse : :Left

On préfèrera la deuxième option car si l'on maintient la souris dans le premier cas alors cela placera plein de cellules partout. La deuxième option ne permet que le clic par clic.
De plus, la gestion des évènements (sf : :event ) permet l'interaction avec l'utilisateur. Cette classe dispose d'une panoplie de méthodes adaptées et les méthodes sont plus spécifiques.

**➢Méthode Pion_cree_apres_click :**

On teste si la souris est bien sur le sprite représentant une cellule (rectangle), pour cela on utilise la méthode déjà existante « contains », car elle est simple d'utilisation (elle prend des coordonnées en paramètres).
Si la souris est bien sur une cellule du tableau alors on valide l'action. L'attribut pion_present de la cellule vaut désormais « true », on affiche le pion puis on joue un son.
Amélioration de cette méthode : par la suite on a jouté une variable qui enregistre le dernier pion_posé afin de pouvoir le colorer.

## III.4 Cellule

Le Renju se joue sur une grille de 15x15. Une grille est composée de cellules, une classe portant ce nom nous a donc paru indispensable.

**Contenu de la classe Cellule :**

**➢Attributs :**

Une cellule est déterminée par sa position vis-à-vis de la grille, des unsigned int permettent d'éviter un gaspillage de mémoire (comparé aux int).
Un attribut de type booléen nous indique si la cellule contient un pion.
Par la suite, on divisera cet attribut en deux autres afin de savoir si la cellule contient un pion noir ou un pion blanc.

**➢ Methodes :**

Nous ne détaillerons pas mais la classe possède de nombreux accesseurs ainsi qu'un constructeur par défaut. Une méthode qui affiche graphiquement une cellule « Print_Cellule() »est essentielle.
⇨Divers tests ont été effectués sur la cellule avant d'obtenir un résultat concluant. La cellule a été liée par inclusion de « headers »(.h) à plusieurs classes, dont la classe Grille et la classe joueur.
Schéma utilisé tout au long du développement :
« un joueur pose un pion sur une grille qui est composée de cellules. »

➢ **rectangle_transparent :**

Cette méthode réinitialise la couleur de la cellule précédemment colorée.

➢ **méthode lumineux_rectangle :**

Cette méthode colore la cellule actuelle.

➢ **méthode get_entity :**

Cette méthode utilise :
sf : :Sprite : :getGlobalBounds() qui retourne la position et la dimension du sprite.
Elle est utilisée par la méthode pion_cree_apres_click afin de savoir si la souris se trouve sur une cellule.

## III.5 Grille :

Comme expliqué précédemment, une grille est un tableau bidimensionnel (statique) de 15x15 cellules. Par conséquent la classe possède un attribut qu'on a nommé « Gri »qui est de la forme Cellule[NMAX][NMAX]. NMAX est défini par une macro préprocesseur qui vaut 15.

**Contenu de la classe Grille :**

➢ **Attributs :**

L'attribut tableau de cellules est le seul de la classe Grille.

➢ **Methodes :**

En revanche, de nombreuses méthodes ont du etre rajouté, en commençant par printGrille qui affiche une grille à l'écran. Il y a également diverses méthodes liées aux jeu comme : qui_commence qui détermine aléatoirement ( grâce au rand ) le joueur qui débute la partie mais également la méthode choix qui demande au joueur (selon les règles officielles du Renju) de choisir la couleur de ses pions, ainsi que la méthode etat_jeu et la méthode print_victoire.

➢ **Création de la grille (constructeur) :**

On fait un tableau de cellules de taille 15x15, on définit au préalable une position fixe à laquelle la première cellule sera créé. Les autres cellules seront positionnées en fonction de la cellule initiale grâce à une addition.

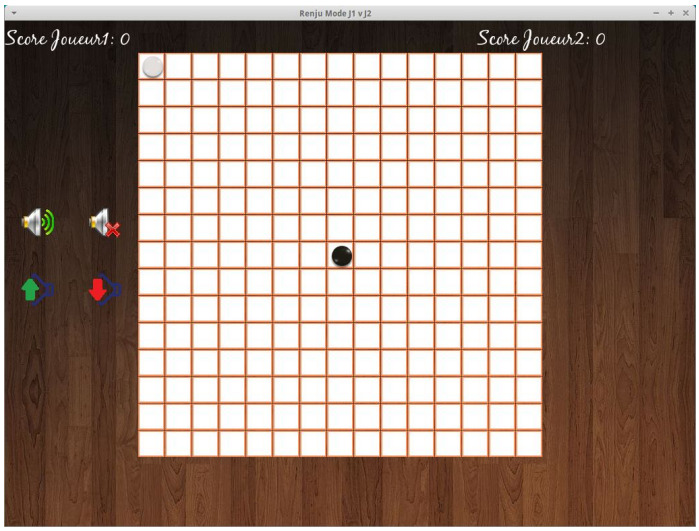



FIGURE 3 – Grille

➢ **méthode : verif_click**

Afin de savoir si le joueur a bien cliqué sur une case de la grille, on parcourt la grille entièrement et on appelle « Pion_cree_apres_click ».

➢ **méthode état jeu :**

Le principe de cette méthode est de parcourir toute la grille en vérifiant l'entourage (voisins) de chaque cellule.
On compte par exemple le nombre de pions de la même couleur alignés à gauche et à droite. Si ce nombre est supérieur à 4 (la cellule initiale n'est pas prise en compte) la méthode retourne le chiffre 1 ou le chiffre 2 selon la couleur du gagnant.
Sinon, elle retourne le chiffre 3 ce qui correspond à une partie qui n'est pas terminée. Il faut également vérifier l'horizontale, la verticale et la diagonale.

Amélioration prévue : La méthode prendra une cellule en paramètre et non la grille entière afin de pouvoir vérifier à chaque fois que l'on pose un pion l'entourage de la cellule.

⇨ D'autres méthodes ont été ajoutées suite à l'ajout de l'IA, nous les détaillerons dans la partie réservée.

# IV Intelligence Artificielle

## IV.1 Algorithme Min-Max :

Comme nous n'avions aucune base sur les algorithmes de l'intelligence artificielle, nous avons choisi l'algorithme min-max pour sa simplicité.
Cet algorithme est utilisé pour des jeux se jouant à tour de rôle comme dans notre cas le renju.
L'algorithme min-max permet à un ordinateur de jouer de manière performante en anticipant la situation plusieurs coups à l'avance.
Le principe consiste a minimiser la perte maximum, c'est-a-dire choisir la meilleure situation dans le pire des cas.
L'algorithme anticipe tous les coups possibles pour l'IA et son adversaire puis assigne une note à chacune des situations obtenues.
Le meilleur des coups choisi correspondra à la meilleure situation possible, tout en supposant que l'adversaire jouera le meilleur de ses coups possibles.
Le déroulement des parties d'un jeu peut être représenté par un arbre de jeu. L'état actuel du jeu est la racine de l'arbre.
La hauteur de l'arbre équivaut au nombre de coups a simuler et chaque niveau contient tous les coups possibles qui peuvent être joués par un joueur.
Les feuilles de l'arbre sont des états finaux du jeu où aucun mouvement ne peut-être réalisé parce qu'un joueur a gagné ou il y a eu match nul.

Avec le Renju, comme on a une grille de 15x15, il n'est pas possible d'aller explorer toutes les possibilités qui sont en nombre trop grand (pour des raisons de temps de calcul et de mémoire).
La profondeur de l'arbre est donc passée en paramètre à la méthode « IA_jouer », ainsi on peut interrompre l'exploration de l'arbre avant d'avoir atteint des fins de partie et par conséquent évaluer l'état du jeu à un instant quelconque.
Plus la profondeur est importante, plus il sera difficile de gagner contre l'IA.

Le problème étant que la notion de score n'existe que pour les états finaux.
Nous avons donc défini une méthode « eval »capable d'évaluer les différentes possibilités et donc permettre à l'ordinateur de choisir le prochain coup avec plus ou moins d'efficacité. Elle servira également a étiqueter les feuilles de l'arbre
Si c'est l'IA qui a gagné on renvoie une très grande valeur et à l'inverse si l'IA a perdu, on renvoie une très petite valeur. On renvoie 0 s'il y a match nul et si le jeu n'est pas fini et que personne n'a gagné, on calcule alors le nombre de séries de 2 pions alignés de l'IA moins celui de l'adversaire.

## IV.2 ➢ Sources et remarques :

Le fait de consulter plusieurs sources nous a permis de comprendre certaines de nos incompréhensions et en observant les diverses illustrations nous avons pu nous faire une idée claire sur le principe du min-max et de la fonction « eval » ;
Nous avons donc implémenté l'algorithme en c++ et transformé les fonctions en méthodes.
De ce fait, les méthodes étant publiques, elles avaient accès aux attributs publiques/privés.
Nous avons donc pu ôter un paramètre de chaque fonction du min-max car nous avons pu accéder à cet attribut librement par des accesseurs en lecture/écriture.

Une première implémentation s'est d'abord faite selon les recommandations de Mr Lionel Ramadier sur le jeu de Morpion qui se joue sur une grille 3x3.
Le fait de procéder ainsi nous a été d'une très grande utilité pour la suite car nous avons pu nous tromper sur le Morpion ce qui nous a permis d'éviter de refaire les mêmes erreurs sur le Renju.
Concernant la profondeur choisie sur le Morpion, il n'y a eu aucun souci en complexité (temps), même avec un passage de profondeur 12.

Par contre pour le Renju, le nombre d'intersections augmente considérablement et nous avons remarqué qu'au-delà d'une profondeur = 3, l'IA prenait beaucoup de temps pour jouer.
Algorithmiquement, nous avons décidé de suivre une logique assez simple et cela en ajoutant les méthodes directement dans la classe Grille.

Le principe suivi est le suivant :

Le joueur « humain »débute et incrémente la variable globale nb_tours, vient alors le tour de l'IA.
Cette dernière grâce au min max calcule les positions max i et max j qui indiquent où poser le pion associé à sa couleur (ici l'IA joue en noir).
Une fois la position trouvée, il suffit de faire appel à la méthode supplémentaire créé « poser_pion_IA »avec « max_i »et « max_j »comme paramètres pour que l'IA dépose un pion à l'endroit indiqué.
L'IA trouve les coordonnés d'une cellule dans la grille (de 0 à 14), il nous a donc fallu les convertir en coordonnées graphiques adaptées à l'affichage de la grille sur l'écran.

Pour cela, nous avons créé les méthodes : « reverseX(int) »et « reverseY(int) ».

Une fois le pion posé, la variable « nb_tours »se voit incrémentée et c'est au tour du joueur humain.

```
int Grille::reverserX(int a){

if (a<=0) {return 220;}

return 40+reverserX(a-1);

}

int Grille::reverserY(int a){

if (a<=0) {return 65;}

return 65+reverserY(a-1);

}
```

## IV.3 Algorithme Alpha-Beta :

L'algorithme du Min-Max s'avère être très lent dès que le jeu a beaucoup de cases, c'est pourquoi on implémente son amélioration nommée Alpha-Beta.

Dans un premier temps, pour avoir un aperçu du fonctionnement et des performances de l'algorithme Alpha-Beta, nous avons réalisé comme pour le min-max un morpion.

Nous avons choisi le morpion car la taille de la grille est plus petite (3*3) et parce que c'est le jeu qui nous as semblé le plus ressemblant au principe du jeu du Gomoku.

Principe : Il n'est pas nécessaire d'évaluer toutes les évolutions possibles lors de la simulation du prochain coup. L'arbre suit le schéma suivant : la racine est un nœud Max, ses fils sont des nœuds Min et ses petits-fils sont des nœuds Max... Si l'on est sur un nœud Min et qu'on a déjà évalué son fils gauche par exemple (qui est le maximum de ses petits-fils à gauche) et que l'on n'a pas encore évalué son fils droit (qui est le maximum de ses petits-fils à droite), il n'est pas forcément nécessaire de regarder tous les nœuds des petits-fils à droite.

Si par exemple, le nœud du petit-fils à droite qu'on regarde est supérieur au fils gauche, il n'est pas nécessaire d'aller plus loin car l'on recherche un nœud Min.

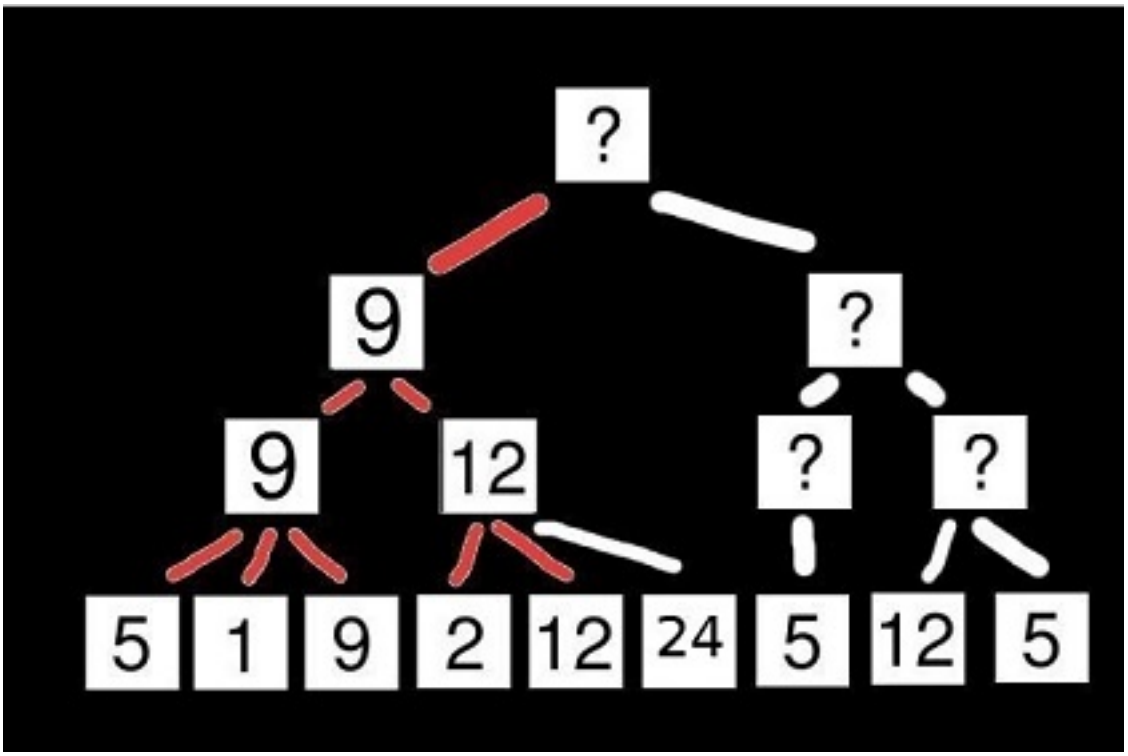



Photo extraite du site : `http://fearyourself.developpez.com/tutoriel/sdl/morpion/part7/` modifiée pour les besoins de l'exemple.
*Il n'est pas nécessaire de regarder le 24 car le nœud en dessous de la racine est un nœud Min.*

## IV.4 ➢ IA :

Nous n'avons pas modifié les classes précédentes afin d'y ajouter les méthodes propres à l'IA.
Pour faciliter la lecture du code, nous avons créé une classe IA.
On introduit deux variables alpha et beta (alpha<beta) qui nous permettront de réaliser plusieurs coupures afin d'évaluer moins de combinaisons.

### ➢ Méthodes de la classe IA :

Nous n'avons pas trouvé utile d'insérer des attributs dans notre classe « IA »car nous utilisons des objets créés avec d'autres classes.
Exemple : la grille → Classe Grille.

#### *Méthodes Min et Max :*

Afin de permettre le calcul des valeurs de chaque case pour l'IA, nous avons mis en œuvre 2 méthodes « Min »et « Max », « Min »calcule la valeur minimale de la grille.

Le principe de ces méthodes est de parcourir l'ensemble de la grille, on teste si chaque case ne contient ni pion noir, ni pion blanc (afin d'éliminer le calcul des cases déjà remplies), on simule un placement de pion de l'IA et on évalue la valeur de la case selon le résultat trouvé par la méthode « Max »(dans le cas de la méthode « Min »).

On compare ensuite cette valeur avec notre valeur de variable beta : si cette valeur est moindre on remplace la valeur de la variable beta par la valeur trouvée.

C'est le même principe de fonctionnement pour la méthode « Max »sauf que « Max »calcule la valeur maximale de la grille (alpha).

*Changement dans la méthode IA_jouer :*
Rappel : dans l'algorithme du « min-max »la méthode « IA_jouer »regarde tous les coups possibles puis appelle la méthode « Max ».
Ici on a effectué quelques modifications afin d'évaluer moins de coups.

Afin de lancer l'IA on appelle la méthode Min

```
tmp = Min(profondeur-1,window, alpha, beta,A);
                        if(alpha<tmp)
                        {
                        alpha = tmp;
                        maxi = i;
                        maxj = j;
                        }
```

➢ **Méthode jouer :**

Dans un premier lieu cette méthode fait en sorte que le joueur puisse jouer, puis vérifie également si le clic est sur la grille.

Nous n'avons pas converti les coordonnées d'une cellule du tableau en coordonnés graphiques car les méthodes « contains »(méthode incluse dans SFML) et « get_entity »(méthode définit précédemment) permettent de faire sans.

*Faire un lien avec l'interface graphique :*
Pour lancer une partie Joueur contre IA avec l'algorithme Alpha-Beta, nous avons rajouté une méthode dans la classe « Menu » : « drawjeuIA »qui permet au joueur de sélectionner l'option « 1 joueur »du Menu et de lancer sa partie contre l'IA.

# V Classe de complexité :

La complexité totale du Renju que nous avons créé diffère selon deux cas :

## V.1 Le mode J1 VS J2 :

Dans le pire des cas comme en moyenne ou dans le meilleur des cas, la complexité ne change pas ( multiplication par une constante donc ignorée).
Le meilleur des cas est quand l'un des joueurs réussit à finir la partie en 5 tours.
Si après 60 tours les deux joueurs ont épuisé leurs 60 pions et que personne n'a gagné alors c'est le pire des cas. Le cas moyen revient à résoudre une équation mathématique qui fait intervenir les probabilités en moyenne, un vainqueur est décidé après un nombre "n" de tours, où n = 19 tours .

La complexité se calcule suivant les méthodes appelées, certaines méthodes ne sont appelées qu'une fois et leur complexité est englobée dans une autre plus grande (comme la méthode choix). En revanche, certaines méthodes sont là du début à la fin - comme la méthode Draw_Grille()-. Après calcul, la complexité totale du Renju dans le mode J1 VS J2 est en O(n^4 + n^2 + n ) , n^4 pour la méthode état jeu() qui parcourt et recherche 5 pions alignés à chaque fois qu'un joueur pose un pion. On multiplie tout cela par le nombre de tours qui est entre 5 et 60, la complexité totale est donc en O(n^4).

## V.2 Le mode J1 VS IA :

Soit « **j** »la complexité en espace mémoire que nécéssite le joueur humain (définie précédemment ).
Soit « **i** »la complexité en espace mémoire que nécéssite l'IA .

**i se définit comme tel :**

$$i = O((c-1)!/((c-1)-p)!) \tag{1}$$

Avec c=15x15 ( grille de 15x15) et p = 2 ( profondeur ) .

Nous pouvons estimer la complexité en espace mémoire à O(nb-tours*(j+i)) où j correspond à la complexité du joueur et i correspond à la complexité de l'IA.
Le nombre de tour est constant.
Nous pouvons dire que la complexité du jeu est O(j+i).
De plus, nous avons choisi une profondeur de 2 ce qui étend la simulation (Suivant la formule donné en haut ) à 38 220 états différents.
Le jeu est donc stable sur les machines actuelles en terme d'espace mémoire.
Cependant, le défaut de l'IA - min-max - est sa lenteur au niveau du temps.
Nous avons pu remarquer que plus nous augmentions la profondeur du jeu, plus l'IA mettait du temps à jouer. Nous avons donc choisi une profondeur assez petite.

# VI Conclusion

Nous pouvons d'ores et déjà dire que ce projet nous a appris ce qu'est le travail de groupe, nous avons eu à prendre des décisions et des initiatives afin de réussir au mieux notre objectif. A chaque étape du projet, nous nous sommes répartis des tâches afin que chacun d'entre nous fournisse une part de travail personnel.
Malgré les difficultés rencontrées lors de la réalisation du projet, nous avons pu aller au-delà des erreurs rencontrées pour ainsi atteindre notre objectif. A ce stade, nous pouvons dire que notre programme Renju fonctionne, que ce soit une partie Joueur contre Joueur ou une partie Joueur contre Intelligence Artificielle.
Avant la réalisation de ce projet nous n'avions aucune base en Intelligence Artificielle. Nous avons donc pu acquérir des connaissances sur le fonctionnement, le calcul des coups réalisés par une Intelligence Artificielle, ainsi que sur l'estimation du temps de calcul de ces coups.
Par la suite, nous souhaiterions améliorer le programme. Dans un premier temps, nous aimerions améliorer la partie IA contre Joueur. Il serait souhaitable que les coups joués par l'IA soient différents à chaque partie car nous avons pu remarquer qu'ils pouvaient être répétitifs. Nous souhaiterions également revoir la complexité de certaines méthodes. En dernier lieu, des améliorations graphiques sont envisageables, l'une de ces améliorations serait de permettre à l'utilisateur de changer la taille de la fenêtre du jeu.
Il serait intéressant d'adapter d'autres algorithmes tels que le « NegaScout » qui sont plus optimisés mais difficiles avec nos connaissances actuelles. Le NegaScout ressemble à l'Alpha-Beta mais on part du principe que le premier noeud exploré est le meilleur. Ensuite, on vérifie que cela est vrai en recherchant les nœuds restants en prenant alpha = beta. Cet algorithme est moins rapide que l'Alpha-Beta dans le cas où la recherche du prochain coup est aléatoire. Il faut donc avoir défini un bon ordre de recherche, c'est pourquoi il semble plus difficile à implémenter.

# VII Bibliographie :

## VII.1 Les icônes :

https://www.iconfinder.com/search/?q=volume

## VII.2 Le graphisme :

http://www.sfml-dev.org/tutorials/2.3/index-fr.php
http://www.sfml-dev.org/faq.php
https://openclassrooms.com/courses/creez-des-applications-2d-avec-sfml/installation-de-la-sfml
http://stackoverflow.com/questions/24354423/moving-rectangle-in-sfml-how-to-make-it-smoother

## VII.3 Intelligence Artificielle :

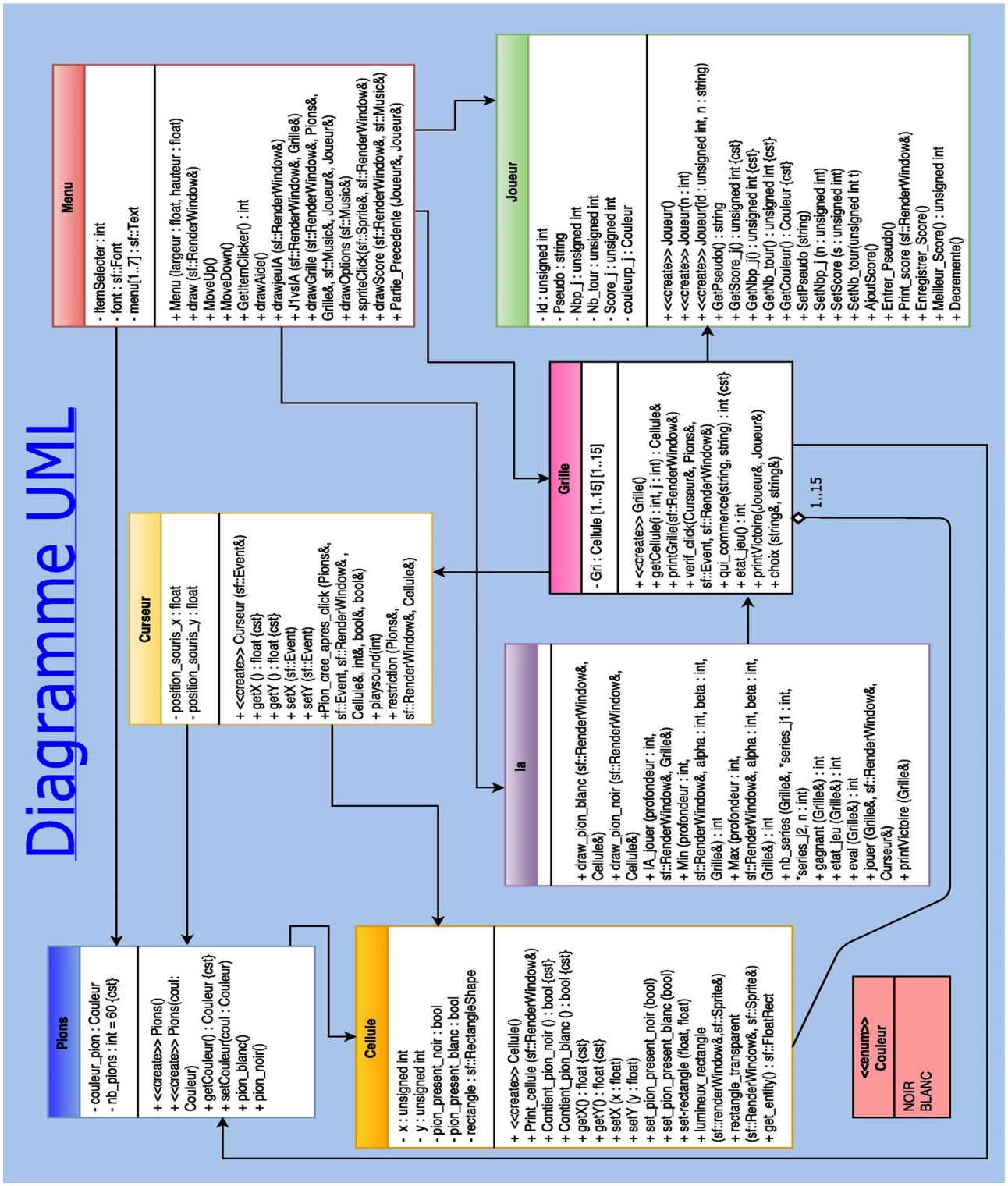
https://openclassrooms.com/courses/1-algorithme-min-max
https://fr.wikipedia.org/wiki/%C3%89lagage_alpha-b%C3%AAta
https://fr.wikipedia.org/wiki/Algorithme_minimax
http://www.grappa.univ-lille3.fr/~torre/Enseignement/Cours/Intelligence-Artificielle/jeux.php
http://neverstopbuilding.com/minimax
http://stackoverflow.com/questions/3437404/min-and-max-in-c
http://fearyourself.developpez.com/tutoriel/sdl/morpion/part6/
http://fearyourself.developpez.com/tutoriel/sdl/morpion/part7/
http://pageperso.lif.univ-mrs.fr/~liva.ralaivola/lib/exe/fetch.php?id=teaching%3A20122013%3Aprojetalgo&cache=cache&media=teaching:20122013:minmax.pdf
https://www.youtube.com/watch?v=KS2QkHe-hpE

## VII.4 LaTex :

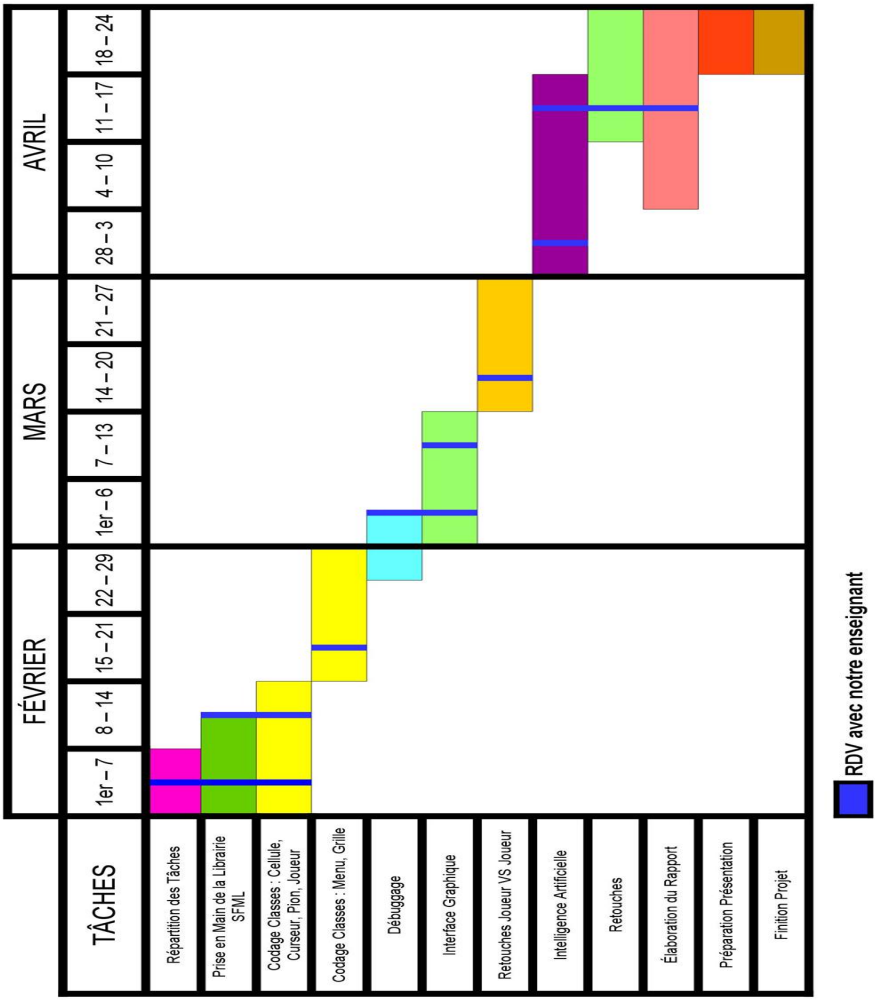
http://tex.stackexchange.com/questions/37581/latex-figures-side-by-side
http://latex-community.org/forum/viewtopic.php?f=44&t=11435
http://tex.stackexchange.com/questions/48772/how-do-i-keep-a-string-of-text-together-without-it-doing-a-line-break
https://openclassrooms.com/courses/redigez-des-documents-de-qualite-avec-latex/maitriser-sa-mise-en-page-1-2
https://openclassrooms.com/courses/redigez-des-documents-de-qualite-avec-latex/maitriser-sa-mise-en-page-2-2
https://en.wikibooks.org/wiki/LaTeX/Special_Characters#/media/File:LaTeX-dingbats.png

# VIII Annexes :

## VIII.1 Diagramme UML :



## VIII.2 Diagramme Gantt :




## VIII.3 Version LaTex :

**Texnic center version 2.02 64 bits**
**Xubuntu 15.10, Wily Werewolf**