

Rapport de stage

Développement d'un joueur à intelligence artificielle basée sur l'algorithme Alpha Bêta

ACHARD--BONNET César
REZENTHEL Mathias

Avril - Mai 2021


Tuteur de stage : STEPHAN Igor

Etablissement : UFR Sciences d'Angers - Licence 3 (2020-2021)



Je, soussigné (e) ACHARD--BONNET César,
déclare être pleinement conscient(e) que le plagiat de documents ou
d'une partie d'un document publiés sur toutes formes de support, y
compris l'internet, constitue une violation des droits d'auteur ainsi
qu'une fraude caractérisée. En conséquence, je m'engage à citer toutes
les sources que j'ai utilisées pour écrire ce rapport ou mon mémoire.

Signature :



ACHARD--BONNET César
le 28/05/21

Cet engagement de non plagiat doit être inséré en première page de tous les rapports,
dossiers, mémoires.



Je, soussigné (e) REZENTHEL Mathias,
déclare être pleinement conscient(e) que le plagiat de documents ou
d'une partie d'un document publiés sur toutes formes de support, y
compris l'internet, constitue une violation des droits d'auteur ainsi
qu'une fraude caractérisée. En conséquence, je m'engage à citer toutes
les sources que j'ai utilisées pour écrire ce rapport ou mon mémoire.

Signature :



Cet engagement de non plagiat doit être inséré en première page de tous les rapports,
dossiers, mémoires.

Sommaire

Introduction	6
I - Présentation de l'Arlecchino	7
II - Présentation de l'algorithme Alpha-Beta	9
1 - Algorithme Min-Max	9
2 - Algorithme ou élagage Alpha-Beta	10
III - Présentation de notre implémentation	12
1 - Premier Algorithme	12
2 - Adaptation de l'arène et développement du jeu Arlecchino	12
3 - Modification de l'arbitre et implémentation des différents joueurs	13
IV - Améliorations apportées à notre joueur Alpha-Bêta	15
1- Pourquoi améliorer l'Alpha-Bêta ?	15
2 - Les améliorations que l'on a utilisé	16
3 - La table de transposition et le hachage Zobrist	17
4 - Résultats obtenus	19
Conclusion	21
Bibliographie et Sitographie	23
Annexes	24
Annexe 1: Diagramme de Gantt prévisionnel	24
Annexe 2 : Guide d'installation	25
Annexe 3 : Manuel d'utilisation	26
Annexe 4 : Diagramme de Gantt réel	29

Introduction

En avril dernier, aux Etats-Unis, une intelligence artificielle du nom de Dr. Fill. à remporté une compétition de mots-croisés contre des joueurs confirmés de ce jeu. Ce n'est pas la première fois qu'une intelligence artificielle devient impossible à battre par un humain, on peut notamment citer sans doute la plus connue d'entre elle, AlphaGo, qui a battu l'humain pour la première fois de l'histoire du jeu de go en 2016.

Depuis, nombreuses sont les intelligences artificielles qui ont vu le jour et ce sur tout types de jeux, qu'ils soient des jeux de plateau ou bien des jeux de cartes par exemple.

Dans le cadre de notre troisième année de Licence à l'Université d'Angers, nous avons souhaité effectuer notre stage en se plongeant dans ce domaine de joueurs à intelligence artificielle et, à notre tour, développer notre propre joueur.

Avant de développer notre joueur et le jeu sur lequel ce joueur s'appuiera, nous devons choisir le jeu en question. Une liste de jeux nous a été présentée parmi lesquels se trouvaient le Katarenga (une version simplifiée des échecs), le Tchag (jeu de dames au plateau mouvant), le Blokus (jeu de stratégie, de placement de pièces géométriques), l'Arlecchino (jeu de stratégie pouvant être comparé aux dames) et deux jeux d'accumulation et/ou de gestion de ressources : Palm Island et Tiki.

Nous avons examiné en détail les règles de chacun de ces jeux et avons choisi de travailler sur un jeu de plateau en écartant les jeux d'accumulation de ressources, avant de choisir l'Arlecchino, jeu pour lequel nous éprouvons tous les deux de l'intérêt.

La dernière étape avant de commencer à développer notre projet était d'organiser notre travail. Nous avons réalisé un diagramme de Gantt (observable en Annexe 1) afin de structurer notre projet. Nous avons décidé de travailler en collaboration sur chaque étape de notre projet, en utilisant l'outil de versionnage collaboratif GitHub.

En amont du développement du jeu choisi et de l'adaptation de l'arène, nous nous pencherons sur une première version de l'algorithme Alpha-Bêta. Ensuite, nous pourrons développer les différents joueurs (manuel, aléatoire et alpha-bêta de base) pour enfin terminer par améliorer notre intelligence artificielle au moyen de différentes techniques (table de transposition, fenêtres alpha-beta minimales...). A la suite de chacune de ces étapes, nous nous réservons une période consacrée aux différents tests et correctifs si nécessaires.

I - Présentation de l'Arlecchino

L'Arlecchino est un jeu tactique imaginé par Bernard Tavitian et publié par sa société *LUD éditions* en 2010. C'est un jeu de plateau jouable à 2, 3 ou 4 joueurs et qui pourrait être comparé au jeu des dames ou aux échecs par exemple.

Arlecchino c'est un plateau de 6 cases par 6, rempli, aléatoirement, par 36 pions (35 pions jouables et 1 pion "maître"). Chacun des 35 pions jouables est séparé en 4 compartiments en forme de triangles qui peuvent être colorés en bleu, jaune, rouge ou vert. Ces quatre couleurs représentent les quatre joueurs de la partie. Chaque pion est différent et toutes les combinaisons de couleurs possibles sont représentées une fois.



Figure 1 : Exemple de pions

Ainsi, vous pouvez observer sur la Figure 1 un pion avec chacune des quatre couleurs (pion "1"), un pion avec 2 compartiments jaunes, 1 vert et 1 rouge (pion "2"), un pion avec 3 compartiments bleus et 1 rouge (pion "3") ou encore un pion présentant 4 compartiments rouges (pion "4").

Le 36ème pion, noté "M" sur la Figure 1, est un pion spécial utilisé uniquement au début de la partie. Il est facilement identifiable puisqu'il est noir et blanc mais également puisque c'est le seul pion qui possède des compartiments en forme de carrés.

Le but de ce jeu est simple : être le joueur qui possède le plus de compartiments de sa couleur à la fin de la partie. Pour se faire, une partie est un enchaînement de tours où chaque joueur (dans l'ordre : bleu, jaune, rouge, vert) joue un coup. Pour jouer un coup, un joueur choisit un pion et effectue une ou plusieurs actions parmi un saut (sauter par dessus un pion adjacent) ou un déplacement (déplacement vers une case adjacente vide). Si le pion choisi présente plusieurs compartiments de la couleur du joueur, alors le joueur peut effectuer plusieurs actions : le joueur peut jouer autant d'actions que son pion n'a de compartiments de sa couleur. Les déplacements et sauts ne s'effectuent uniquement de manière verticale ou horizontale.

Afin de rendre les parties dynamiques, un joueur doit obligatoirement effectuer un saut durant son tour, même si la pièce qu'il doit sauter lui est favorable. S'il ne peut effectuer de saut, avec aucune de ses pièces, il passe son tour.

La partie commence quand le pion "Maître" est enlevé, créant la première case vide du plateau. Le joueur qui commence est désigné aléatoirement.

Règles spécifiques à 3 et 2 joueurs :

Pour jouer à 3, il suffit d'enlever, au début de la partie, la pièce présentant 4 compartiments de la couleur qui n'est pas jouée afin de créer une seconde case vide. Par exemple, si la couleur verte n'est pas jouée, il faut enlever le pion qui a 4 compartiments verts.

En jouant à 2 joueurs, l'un joue les pions bleus et rouges et l'autre les pions jaunes et verts.

II - Présentation de l'algorithme Alpha-Beta

L'algorithme Alpha-Bêta est une version de l'algorithme Min-Max, présentant des modifications dans le but d'améliorer la vitesse et l'efficacité de celui-ci. Il est donc important de présenter l'algorithme Min-Max dans un premier temps.

1 - Algorithme Min-Max

L'algorithme Min-Max est utilisé dans un jeu à deux joueurs et à somme nulle : un jeu équilibré pour les deux joueurs (chaque joueur a autant de chances de gagner que l'autre) où chaque gain du joueur 1 est une perte pour le joueur 2.

L'objectif de cet algorithme est de maximiser les gains possibles d'un joueur tout en minimisant les gains possibles de son adversaire (et donc de minimiser les pertes possibles du premier joueur). Pour se faire, l'algorithme va, pour une situation donnée d'un jeu, parcourir tous les coups possibles et déterminer quel coup est le plus intéressant à jouer grâce à une fonction d'évaluation. Cette fonction d'évaluation donne une valeur quantifiable à chaque configuration possible d'un jeu. Si cette valeur est positive, alors la situation est favorable à notre joueur, sinon elle est, au contraire, favorable à notre adversaire.

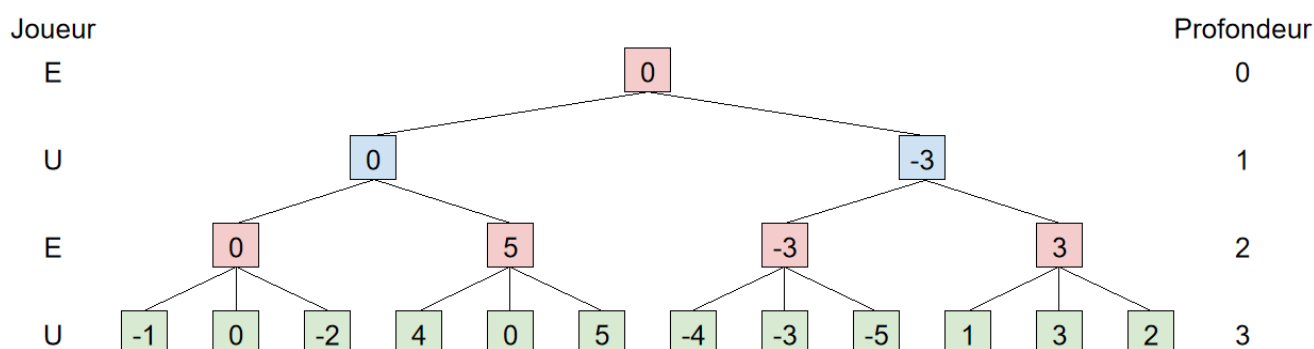


Figure 2 : Exemple d'application de l'algorithme Min-Max

Prenons comme exemple la Figure 2. On nomme Existentiel (E) le joueur que l'algorithme cherche à faire gagner et Universel (U) son adversaire. La Figure 2 est un arbre ayant pour racine le nœud de profondeur 0, représentant la situation dans laquelle se trouve le joueur E. Chaque fils d'un nœud, représente un choix possible pour le joueur E, ou U selon la profondeur de l'arbre / le tour du jeu. Les nœuds verts sont des situations finales (des feuilles), autrement dit des issues possibles du jeu avec pour chacune leur valeur associée.

En partant de la profondeur la plus basse (ici 3), l'algorithme va choisir pour chaque nœud "parent" (qui a des enfants) la valeur de l'un de ses enfants en

fonction de si c'est le joueur U ou E qui doit jouer. Si c'est le tour du joueur E, alors il choisira la valeur la plus élevée parmi celles de ses enfants (application de la fonction maximum). Au contraire si c'est le tour du joueur U, il choisira la moins élevée (fonction minimum).

Sur cette figure on peut donc voir les différents choix de l'algorithme, représentés par des chemins rouges ou bleus en fonction du joueur impliqué. Dans cet exemple, le joueur E a tout intérêt à emprunter le premier chemin, de sorte à optimiser ses gains possibles : il peut espérer, au minimum, arriver dans une situation où il remporte un gain de 0.

2 - Algorithme ou élagage Alpha-Beta

L'algorithme Alpha-Bêta est donc une amélioration du Min-Max. Cet algorithme a pour but d'optimiser l'algorithme Min-Max en temps. On a pu observer un exemple simple de Min-Max, mais, appliqué à une situation plus complexe, comprenant un nombre de choix et une profondeur beaucoup plus importants, il peut être utile de ne pas parcourir chacune des possibilités afin de gagner du temps et être donc plus rapide.

Cet algorithme est aussi appelé "Élagage Alpha-Beta" de par son principe. En effet, l'algorithme coupe certaines branches de l'arbre (il ne les parcourt pas) s'il juge que celles-ci sont "inutiles".

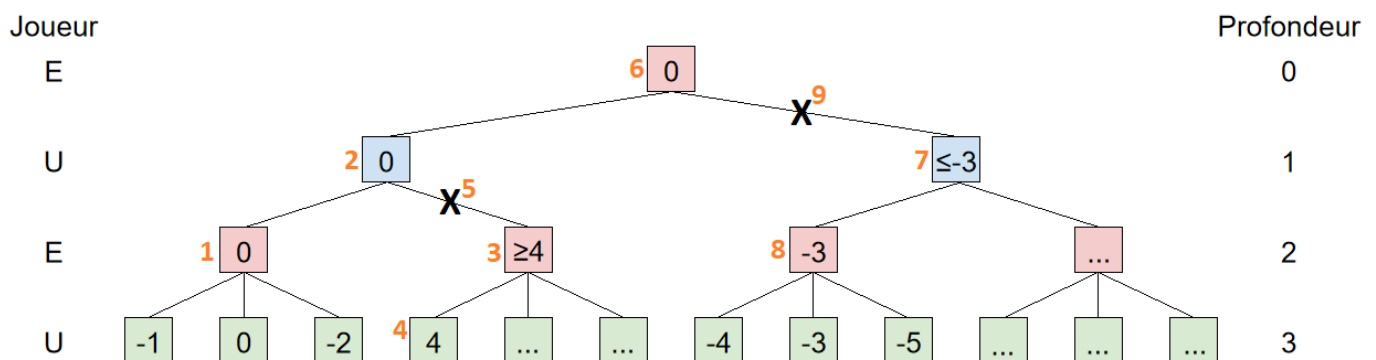


Figure 3 : Exemple d'application de l'algorithme Alpha-Bêta

En reprenant l'exemple précédent et en appliquant ce nouvel algorithme, on peut s'apercevoir que celui-ci n'a pas parcouru certains nœuds.

Pour comprendre comment il fonctionne, on va analyser chacune des 9 étapes numérotées en orange sur la Figure 3 :

- Premièrement, il va descendre en profondeur 2 et calculer le gain espéré du joueur E. Il trouve 0.

- Ensuite, il va remonter en profondeur 1 et évaluer le gain espéré du joueur U. Pour cela, il regarde si ce nœud possède d'autres enfants. La réponse est oui.
- En 3, il va donc descendre en profondeur 2 et évaluer le gain espéré du joueur E.
- Il descend en profondeur 3 et il tombe sur une feuille. Il va faire appel à la fonction d'évaluation qui lui retournera 4.
- Étape 5 : Ici l'algorithme va couper cette branche de l'arbre. En effet, lors de l'étape précédente, il a trouvé une valeur de 4. Ce qui veut dire que le joueur E peut espérer un gain supérieur ou égal à 4 (selon les frères de ce nœud). Or, une profondeur au dessus, c'est au tour du joueur U de jouer (Etape 2), et celui-ci connaît un chemin qui lui assure un gain maximal de 0 (le joueur U cherchant à minimiser le gain possible), il ne va donc pas s'attarder sur un chemin moins avantageux pour lui.
- Le nœud de l'étape 2 n'ayant plus d'enfant, on peut remonter en profondeur 0 et regarder si ce nœud possède d'autres enfants.
- Étape 7 : on descend en profondeur 1 pour évaluer le coup du joueur U.
- Puis on descend pour explorer le coup du joueur E et on trouve un gain maximal de -3.
- Enfin, avec l'évaluation précédente, on sait que le joueur U à la profondeur 1 (étape 7) pourra espérer un gain inférieur ou égal à -3, ce n'est donc pas la peine pour le joueur E de s'engager dans ce chemin en profondeur 0 puisqu'il connaît un chemin lui garantissant un gain au minimum de 0 : l'algorithme coupe donc cette branche également.

Au total, sur cet exemple simple, presque moitié moins de nœuds ont été parcourus. Appliqué à un arbre aux choix plus complexes et à la profondeur plus grande, le gain de temps est très important.

Que ce soit pour l'algorithme Min-Max ou l'Alpha-Bêta, la fonction d'évaluation est très importante. Si cette dernière n'est pas optimale, votre algorithme perdra en vitesse et en efficacité.

III - Présentation de notre implémentation

Pour développer notre implémentation, nous nous sommes appuyés, comme demandé, sur une arène de jeu qui nous a été fournie par M Igor STEPHAN et Mme Caroline DEVRED, développée dans le cadre d'un projet étudié au cours de cette année : une arène servant à faire s'affronter différents joueurs à intelligence artificielle basée sur le Monte Carlo Tree Search (algorithme de recherche arborescente différente de celle de l'algorithme Alpha-Bêta) sur un jeu de plateau nommé Brix.

Cette arène a été entièrement développée en C++ et c'est pourquoi nous avons décidé de développer notre projet en C++ également, afin de conserver une homogénéité des langages utilisés.

1 - Premier Algorithme

Dans un premier temps, nous avons développé une première version de l'algorithme Alpha-Bêta en se basant sur les cours que l'on a suivis. Ce premier algorithme est consultable à la racine de notre projet sous le nom de *alpha_beta_simple.cpp*.

Afin de tester cet algorithme, nous avons utilisé une structure arborescente simple où chaque nœud possède deux enfants ou alors est une feuille. De nombreux tests ont été effectués et l'algorithme fonctionne comme on le souhaitait. Il a été développé de manière à pouvoir être adapté facilement à n'importe quelle structure arborescente, ainsi il sera plus facile de l'implémenter à notre projet plus tard.

2 - Adaptation de l'arène et développement du jeu Arlecchino

Une fois notre premier algorithme terminé, il était temps pour nous de modifier l'arène qui nous a été fournie.

Nous avons supprimé les éléments qui n'allaient pas nous servir, ainsi, tout ce qui était en rapport avec le Monte Carlo Tree Search ou le jeu du Brix a été effacé, parfois partiellement (comme la structure du jeu de Brix) dans le but de conserver l'essentiel de la structure qui avait été mise en place.

Dans un second temps, nous avons développé le jeu de l'Arlecchino en le subdivisant en deux classes : *piece* et *jeu*. Ces classes sont observables dans le répertoire *jeu_arlecchino/* sous les fichiers de *jeu.cpp*, *jeu.hh*, *piece.cpp* et *piece.hh*.

La classe `piece` représente une pièce, un pion du jeu. Chaque pièce est représentée par ses couleurs sous forme d'un string : ex. la pièce observable sur la Figure 1, page 7, numérotée 3 sera identifiée par les initiales de ses couleurs soit "BBBR".

Cette classe contient aussi certaines méthodes utiles dans le fonctionnement du jeu comme "degre", renvoyant le degré d'une pièce, soit le nombre d'occurrence d'une couleur passée en paramètre. Cette méthode permet notamment de définir le nombre de déplacements maximum que la pièce peut effectuer en un tour.

La classe `jeu`, elle, est plus complexe. Elle regroupe la modélisation du jeu et de ses règles mais aussi la modélisation du plateau. Cette classe est représentée par un plateau rempli de différentes pièces (une case vide est une pièce de couleur "----"), l'état actuel de la partie (si elle est nulle, en cours ou si un des joueurs a gagné), le nombre de tours joués et enfin la couleur, sous entendu le joueur, à qui c'est le tour de jouer.

De nombreuses méthodes sont présentes dans cette classe et notamment celles destinées à tester les différents coups possibles à chaque état de la partie, mais aussi les méthodes qui jouent ces coups, actualisent le plateau et l'état de la partie.

Toutes ces méthodes communiquent entre elles et permettent une certaine lisibilité de notre architecture.

3 - Modification de l'arbitre et implémentation des différents joueurs

Le jeu étant développé, nous avons adapté l'arbitre, classe permettant d'organiser un "challenge", nombre défini de parties à la suite, entre deux joueurs. Si, lors d'un coup, un joueur ne rend pas le coup qu'il souhaite jouer dans le temps imparti, l'arbitre fera gagner automatiquement son adversaire. De la même manière, si un coup est invalide (s' il ne contient pas de saut, ou s' il essaie de bouger une pièce qui ne possède aucun compartiment de sa couleur par exemple) l'arbitre donnera la victoire à son adversaire.

Nous avons ensuite modifié et adapté les différents joueurs "de base" aux règles de l'Arlecchino et à notre implémentation.

Le premier joueur est le joueur aléatoire, qui joue un coup aléatoirement choisi parmi l'ensemble des coups qu'il peut potentiellement effectuer, que ce soit le plus mauvais, ou le meilleur possible. S' il ne peut jouer aucun coup, il passe son tour.

Le second joueur est le joueur manuel, autrement dit, un utilisateur humain, choisissant les coups qu'il souhaite jouer en utilisant un clavier d'ordinateur. Si vous souhaitez davantage d'informations sur comment utiliser le joueur manuel, vous trouverez en Annexe 3 un Manuel d'utilisation détaillé.

Le joueur manuel est particulier puisqu'il est le seul qui nécessite une intervention humaine pour fonctionner. De ce fait, les parties sont plus longues puisque le temps d'un tour est adapté pour permettre à l'utilisateur de réfléchir et de saisir le coup qu'il souhaite jouer.

Le troisième et dernier joueur est le joueur alpha-bêta. Ce joueur repose, dans un premier temps, sur la première version de l'algorithme Alpha-Bêta que l'on vous a présenté un peu plus tôt. Ce joueur a été plusieurs fois amélioré, afin d'optimiser son temps de réflexion pour qu'il puisse analyser sur plus de profondeur afin de maximiser son nombre de victoires contre n'importe quel autre joueur. Ces améliorations vont vous être présentées par la suite.

Enfin, l'arbitre peut organiser un challenge entre 2 joueurs différents mais aussi entre 2 versions d'un même joueur : c'est-à-dire qu'il peut faire s'affronter deux joueurs aléatoires ou bien deux joueurs alpha-bêta par exemple. On notera que si l'on veut faire s'affronter deux joueurs manuels, les deux joueurs joueront avec le même clavier, notre application fonctionnant en local.

IV - Améliorations apportées à notre joueur Alpha-Bêta

Parlons maintenant des différentes améliorations applicables à l'Alpha-Bêta.

1- Pourquoi améliorer l'Alpha-Bêta ?

Lors d'une partie, notre joueur sera limité dans sa recherche par le temps qui lui sera imposé pour réfléchir. Ainsi plus il est performant et rapide, plus la profondeur sur laquelle les coups possibles sont évalués sera importante, et donc meilleure sera l'information du meilleur coup à jouer. Ainsi la performance d'un joueur alpha-beta lors d'une partie est fortement liée à ses améliorations.

Il existe trois types d'améliorations de l'algorithme Alpha-Bêta :

Trier :

Cela consiste à modifier l'ordre d'examen des nœuds afin d'effectuer un maximum de coupes

Il existe par exemple la méthode de l'ordonnancement quantitatif : on explore en premier les situations dont le nombre de coups légaux issus est le plus faible, cela permet de rétrécir la fenêtre alpha-beta plus rapidement et donc de faire des coupes plus rapidement.

La méthode la plus connue et la plus efficace est la recherche en profondeur itérative. Alors que l'Alpha-Bêta effectue une recherche en profondeur, si le chemin le plus optimal est situé en dessous du second nœud de la racine, l'Alpha-Bêta explore d'abord tous les nœuds situés sous le premier nœud : il dépense un temps inutile à explorer toutes les profondeurs sous ce premier nœud.

C'est pourquoi la recherche en profondeur itérative appelle l'Alpha-Bêta en profondeur 1, puis 2, etc, en parcourant chacun des nœuds d'une profondeur avant de passer à la suivante tant que du temps de réflexion existe. L'avantage est que tant que du temps reste, on explore la profondeur suivante, si le temps est écoulé, on retourne le coup calculé à l'itération précédente (soit la profondeur précédente).

Réduire :

La réduction est une méthode intéressante puisque plus la fenêtre de recherche ($\beta - \alpha$) est petite, plus il y a de coupes possibles.

La méthode la plus efficace est celle nommée Principal Variation Search (PVS). C'est une recherche qui suppose que les nœuds sont déjà bien ordonnés par

le générateur de coups et donc que la recherche est simplement une vérification. En d'autres termes, cela suppose que le premier nœud fait partie de la variante principale (variante la plus intéressante à jouer). Ensuite, il peut vérifier si cela est vrai en recherchant les nœuds restants avec une fenêtre nulle, ce qui est plus rapide qu'avec une fenêtre Alpha-Bêta normale. Si la recherche échoue, alors le premier nœud n'était pas dans la variante principale et la recherche se poursuit comme une alpha-bêta normal. Par conséquent, PVS fonctionne mieux lorsque l'ordre des nœuds est bon. Avec un ordre de déplacement aléatoire, PVS prendra plus de temps que l'Alpha-Bêta classique.

Réutiliser :

Sauvegarder les évaluations de certaines configurations dans le cas où elles apparaîtraient de nouveau au cours de la partie si une même configuration de plateau est accessible par différents mouvements, ou parce que l'on a relancé l'Alpha-Bêta avec des paramètres différents peut s'avérer être intéressant pour éviter de calculer de nouveau les issues possibles de cette configuration.

Pour cela on utilise une table de transposition : il est possible d'atteindre une même position par plusieurs chemins différents. On stocke alors les positions, leur évaluation et la profondeur associée dans la table de transposition. Retrouver un nœud déjà évalué dans les tables permet alors les optimisations suivantes : si le nœud a déjà été évalué à une profondeur au moins aussi grande que celle désirée, sa valeur devient celle stockée. Pour permettre une recherche rapide sur la table, on l'implémente avec une table de hachage, on reviendra la dessus juste après.

2 - Les améliorations que l'on a utilisé

Nous avons utilisé la méthode MTD(f) car elle a pour avantage de combiner les idées précédentes : réduire la fenêtre au maximum (de taille 1), même si il faut relancer l'Alpha-Bêta plusieurs fois, et utiliser des tables de transposition pour stocker les valeurs déjà calculées (si plusieurs Alpha-Bêta ré-évaluent les mêmes positions). Le nom MTD(f) est une abréviation pour MTD(n,f)(Memory-enhanced Test Driver avec n nœud et la valeur de l'évaluation f).


```

function MTDf(root, f, d) is
    g := f
    upperBound := +∞
    lowerBound := -∞

    while lowerBound < upperBound do
        β := max(g, lowerBound + 1)
        g := AlphaBetaWithMemory(root, β - 1, β, d)
        if g < β then
            upperBound := g
        else
            lowerBound := g

    return g

```

Figure 4 : Pseudo-code de la méthode MTD(f)
 (source : <https://en.wikipedia.org/wiki/MTD-f>)

L'optimisation MTD(f) effectue uniquement des recherches Alpha-Bêta à fenêtre zéro -> $\alpha = \beta - 1$, avec une «bonne» borne (variable beta). Les appels à fenêtre zéro provoquent plus d'élagage, en effet appeler alpha beta avec une fenêtre de valeur très proche de celle de la variation principale est un gain de temps et d'efficacité.

Pour trouver la valeur minimax, MTD (f) appelle Alpha-Bêta un certain nombre de fois, nombre convergeant vers elle et finissant par trouver sa valeur exacte. Pour s'en faire une meilleure représentation, on peut observer le pseudo-code en figure 4 ci-dessus. La variable f représente l'estimation de ce qu' Alpha-Bêta va retourner. Dans notre cas on prend comme valeur initiale 0. Alpha-Bêta est lancé une première fois avec $\alpha = \beta - 1$ et $\beta = 0$.

Dans notre implémentation il n'y a pas de borne supérieure car si la borne supérieure doit être modifiée, c'est que l'on a pas trouvé de coup supérieur à β , ainsi, on peut directement retourner le meilleur coup car on aura parcouru tous les coups possibles (puisque pas d'élagage β). Nous avons également implémenté une table de transposition qui stocke et récupère les parties précédemment recherchées de l' arbre en mémoire pour réduire la surcharge de ré-exploration des parties de l' arbre de recherche.

3 - La table de transposition et le hachage Zobrist

Comme évoqué précédemment pour que la table de transposition soit efficace, il faut qu'elle ne stocke que des informations très légères pour ne pas ralentir le temps d'exécution. On utilise alors le hachage de Zobrist. Cette technique de hachage portant le nom de son inventeur permet de représenter une

configuration, c'est-à-dire l'état du plateau à un instant T mais aussi des éléments de contexte (à qui est le tour de jouer par exemple) en un nombre entier de 20 bits.

On décide de générer un nombre de 20 bits pour éviter les collisions, en effet un même nombre pourrait représenter deux configurations identiques puisque c'est aléatoire, alors on génère de grands nombres pour réduire au maximum cette probabilité. Nous avons par ailleurs estimé qu'un entier non signé de 20 bit était assez grand pour éviter les collisions, surtout que nous créons une nouvelle table entre les coups, cela permet de ne pas ralentir l'exécution avec de très grand nombre de 64 bit par exemple. Pour ce faire on génère un nombre aléatoire pour chaque case et pour chacune de ces cases on génère un nombre aléatoire par pièce pouvant être sur cette case et enfin un nombre aléatoire pour chaque élément de configuration.

Dans notre cas, le plateau comporte 36 cases et il y a 36 pièces différentes possibles, il y a quatre couleurs à qui ça peut être le tour de jouer et on prend également en compte la profondeur notée p , à laquelle la configuration est évaluée. Cela fait donc $36 \times 36 \times 4 \times p$ configurations possibles. On a donc créé un tableau contenant $36 \times 36 \times 4 \times p$ nombres entiers aléatoires qui contient alors toutes les combinaisons de pièce/case/couleur/profondeur possibles.

Pour obtenir le nombre de Zobrist d'une configuration on fait un "ou exclusif" (XOR) de toutes les cases du tableau ayant les indices correspondants aux éléments de contexte de la configuration recherchée, on obtient alors un nombre de 20 bits qui correspond à une configuration unique. Pour accéder rapidement aux données d'une configuration stockée, on utilise le nombre de zobrist comme indice de notre tableau.

Enfin les éléments stockés pour une configuration constituent le nombre de Zobrist de 20 bits. On peut ensuite, si la configuration a été stockée regarde l'évaluation de la configuration qui est stockée en borne inférieure (si son évaluation a dépassé la valeur de β), et/ou celle qui est en borne supérieure (si son évaluation est inférieure à α), de manière à ce que quand on retombe sur cette configuration, si la borne inférieure existe et est supérieure au β actuel, alors on sait qu'il y a un élagage β ; au contraire, si la borne supérieure existe et est inférieure au α actuel, alors il y a un élagage α . Sinon on continue la recherche en affinant α si la borne inférieure a été stockée et qu'elle est supérieure à α et, réciproquement, β si la borne supérieure a été stockée et qu'elle est inférieure à β .

Nous avons également voulu implémenter la recherche en profondeur itérative qui aurait très bien pu s'ajouter au MTD(f) mais après quelques essais, nous avons eu du mal à implémenter la prise en compte du temps dans les recherches, surtout le fait d'estimer combien de temps une recherche allait prendre pour ne pas dépasser le délai restant. Ainsi notre joueur ne jouait parfois pas dans les temps,

aussi nous avons essayé d'implémenter cette fonction vers la fin de notre stage donc le temps nous manquait pour s'attarder la dessus, nous avons décidé de nous en tenir à notre Alpha-Bêta se lançant plusieurs fois avec une fenêtre de recherche à zéro pour trouver plus rapidement la variation principale, couplé à notre table de transposition.

4 - Résultats obtenus

Voici à présent quelques tests destinés à comparer les résultats obtenus par notre Alpha-Bêta amélioré et par notre l'alpha-Beta simple (sans amélioration).

Les tests ont été effectués sur le même PC dont voici les caractéristiques principales (à titre informatif) :

Processeur : Intel Core i5-700HQ , 2,5GHz, 4 coeurs

RAM : 8Go

Nous avons volontairement choisi un temps pour un tour un peu trop court pour que les deux versions de l'alpha beta (améliorée en non améliorée) ne rendent pas systématiquement les coups dans les temps afin de voir lequel dépasserait le délai le plus souvent.

Les matchs se jouent en 30 parties, la profondeur de recherche pour les joueurs Alpha-Bêta est fixée à 4. Le temps maximum pour un coup est de 1000 ms.

Notre joueur amélioré est nommé "Alpha Beta" et le joueur non amélioré est nommé "Simple AlphaBeta".

Comparaison des résultats de ces deux joueurs contre le joueur aléatoire :

Scores :

Simple AlphaBeta 8 - Joueur Aléatoire 22

AlphaBeta 18 - Joueur Aléatoire 11

Défaites par coup non rendus :

Simple AlphaBeta 12

AlphaBeta 7

On constate plus de victoires du côté de notre joueur amélioré mais surtout 2 fois moins de défaites. Aussi, on dépasse moins de fois le délai pour jouer un coup.

Résultat d'un match de notre joueur amélioré contre le joueur alpha beta simple :

Score :

Simple AlphaBeta 8 - AlphaBeta 21

Défaites par coup non rendu :

Simple AlphaBeta 11

AlphaBeta 5

Notre joueur amélioré semble rendre là encore plus de coups dans les temps mais aussi gagner plus souvent contre le joueur Alpha-Bêta simple, sans prendre en compte les défaites ou victoires reliées aux coups pas rendus à temps.

Ainsi de par ces résultats, on peut constater qu'à profondeur égale notre joueur amélioré semble être un peu plus rapide et efficace, c'est ce que nous attendions mais nous espérions une différence plus flagrante. On remarque également que notre joueur semble gagner plus souvent face au joueur Alpha-Bêta simple et ce même à profondeur égale, ainsi les améliorations que l'on a apportées permettent sans doute également une estimation du meilleur coup plus juste.

Résultat d'un match de notre joueur amélioré contre le joueur Alpha-Bêta simple en rééquilibrant les paramètres:

Regardons maintenant le résultat d'un match avec des conditions plus égales: Comme le joueur alpha beta simple rend moins souvent ses coups dans les temps, il descend maintenant à une recherche en profondeur trois, ainsi il est sûr de rendre ses coups à temps, mais notre joueur amélioré reste en profondeur 4, ainsi pour que lui aussi rende tout de même plus de coups dans les temps, on augmente le temps pour un coup mais pas trop non plus pour ne pas avantager notre joueur. Ainsi on passe de 1000 ms à 1400 ms. Le match se déroule sur 15 parties cette fois.

Score :

Simple AlphaBeta 11 - AlphaBeta 4

Défaites par coup non rendu :

Simple AlphaBeta 0

AlphaBeta 1

On constate que notre joueur bat l'Alpha-Bêta simple et qu'il n'a pas rendu son coup à temps une fois. Le score est logique à nos yeux, étant donné que notre Alpha-Bêta recherchait en profondeur 4 alors que le simple était en profondeur 3, ce qui a fait qu'on avait des potentiels meilleurs coups plus précis.

Conclusion

Ces deux mois de stage ont été intenses mais notre organisation et notre méthode de fonctionnement nous ont permis d'avancer rapidement et de mener à bien nos objectifs. Comme on s'y attendait, certaines tâches nous ont pris moins de temps que prévu et d'autres ont, au contraire, nécessité une plus grande attention et plus de temps. Vous pouvez retrouver notre Diagramme de Gantt réel en Annexe 4.

Le développement du jeu et les améliorations de notre joueur Alpha-Bêta ont été les étapes les plus importantes de notre projet et qui ont nécessité le plus de temps et de travail.

Au cours de notre développement et ce à plusieurs reprises, nos tests nous ont permis d'identifier des comportements qui n'étaient pas voulus et de les corriger. Ces tests nous ont permis de perfectionner notre implémentation et d'explorer un maximum de configurations possibles afin de prévoir une solution pour tout éventuel problème.

La modélisation du jeu notamment, a été mise en place en plusieurs fois, en ajoutant au fur et à mesure des configurations que nous n'avions pas envisagées.

Bien que des améliorations ont été apportées à notre joueur à intelligence artificielle, nous aimerions continuer de le développer et nous aimerions améliorer notre fonction d'évaluation. En effet, celle-ci prend en considération le nombre de compartiments de couleur d'un joueur comme indicateur. Cependant, si l'on joue une pièce de degré 3 (par exemple) et que l'on a le choix entre sauter une pièce quelconque pour se retrouver à côté de pièces appartenant à un adversaire et sauter cette même pièce puis se déplacer pour se mettre à l'abri, la fonction d'évaluation donnera la même valeur pour ces deux configurations alors que l'une d'entre elles est avantageuse pour nous et l'autre non. Prendre en considération les voisins de chacune des pièces est donc important mais nous n'avons pas trouvé d'implémentation qui ne soit pas gourmande en temps et avons préféré conserver l'ancienne fonction d'évaluation pour le moment.

Plus encore, nous aurions aimé créer une interface graphique plus accueillante et lisible pour les utilisateurs, que ce soit pour représenter le plateau et son évolution, mais aussi pour choisir les joueurs en créant un menu simple et accessible.

Dans le futur, nous aimerions développer l'aspect multijoueur de notre application pour la rendre aussi attrayante que la version classique physique de l'Arlecchino.

Ce stage a apporté à chacun de nous deux beaucoup d'expérience tant sur le plan personnel en développant notre autonomie et nos compétences (en C++ par exemple) que sur le plan professionnel. En effet, ce stage nous a mis en situation de projet professionnel et nous a permis de développer nos savoir-être et savoir-faire plus particulièrement notre communication et notre organisation afin de d'évoluer et de travailler avec notre binôme. Aussi, nous avons dû nous adapter au cadre du travail en distanciel, imposé par la situation sanitaire de cette période. Heureusement, l'outil de travail collaboratif GitHub est adapté à ce genre de situation et nous avons pu nous y familiariser assez aisément.

Bibliographie et Sitographie

- Article sur Dr. Fill. <https://www.lebigdata.fr/intelligence-artificielle-mots-croises>

L'Alpha-Bêta et découverte de ses divers améliorations :

- Cours de M Igor STEPHAN dispensés lors de notre L3 informatique
- <https://www.lamsade.dauphine.fr/~cazenave/papers/jeux.pdf>
- <https://www-ljk.imag.fr/membres/Jean-Guillaume.Dumas/Enseignements/Polys/Jeux/jeux.pdf>

Table de transposition et hachage de zobrist :

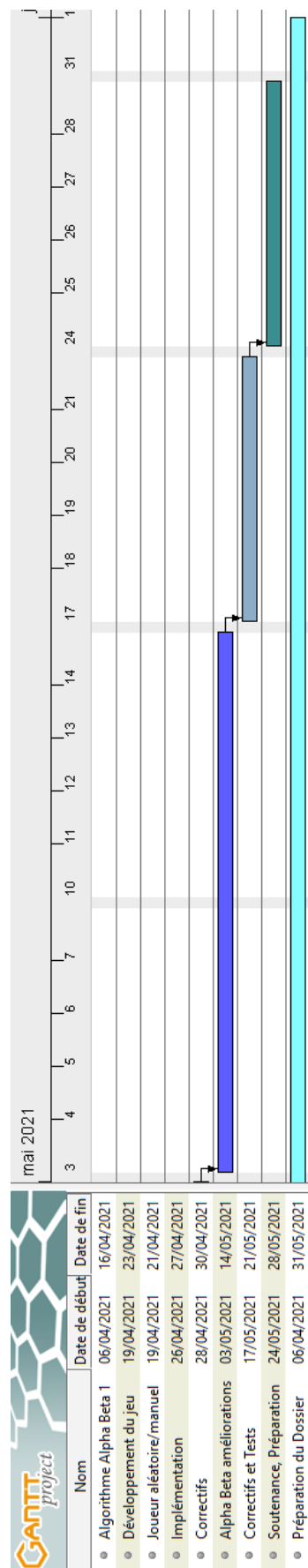
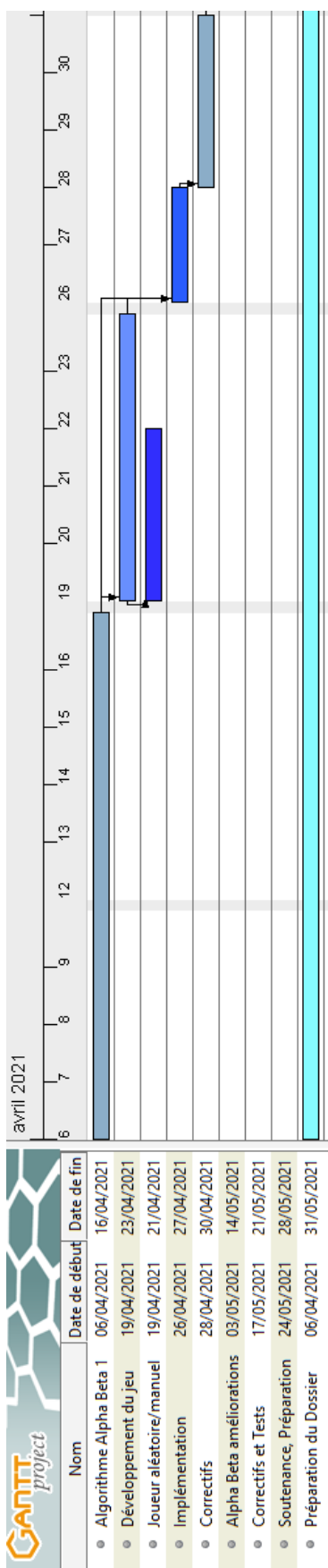
- “Transposition Tables & Zobrist Keys - Advanced Java Chess Engine Tutorial 30” de Logic Crazy Chess
<https://www.youtube.com/watch?v=QYNRvMoIN20&t=2s>
- https://en.wikipedia.org/wiki/Zobrist_hashing
- “Generating Zobrist Keys - Advanced Java Chess Engine Tutorial 32” de Logic Crazy Chess <https://www.youtube.com/watch?v=gyLCFfrLGIM>
- Discussion autour des “Collisions in a Zobrist transposition table”
<https://stackoverflow.com/questions/62541946/collisions-in-a-zobrist-transposition-table>
- Site de Adam Berent
<https://adamberent.com/2019/03/02/transposition-table-and-zobrist-hashing/>

MTD(f) :

- <http://people.csail.mit.edu/plaat/mtdf.html#further>

Annexes

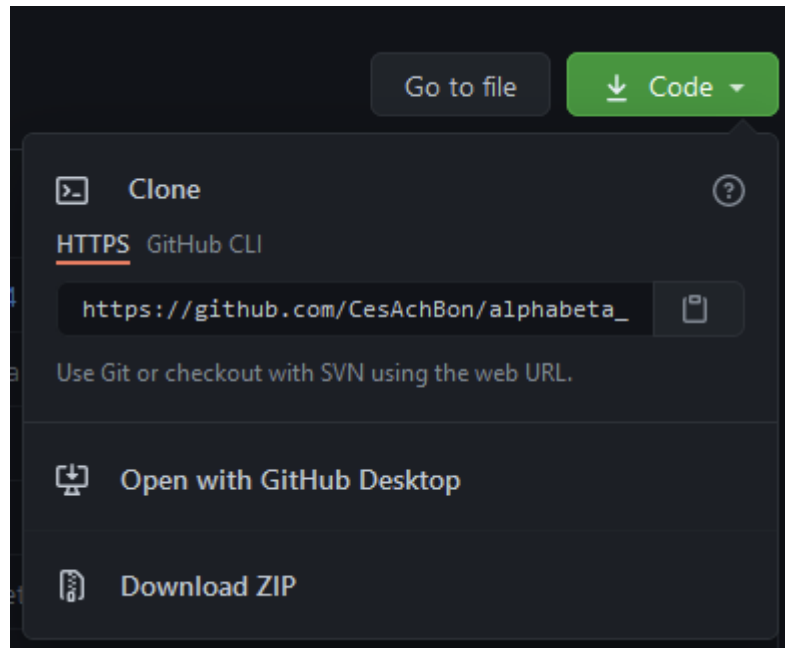
Annexe 1: Diagramme de Gantt prévisionnel



Annexe 2 : Guide d'installation

Afin d'installer notre application il faut tout d'abord se rendre sur notre projet GitHub, disponible grâce au lien ci-dessous et de télécharger notre application directement sous format zip ou en clonant notre projet.

Lien du projet : https://github.com/CesAchBon/alphabeta_project



Structure de notre projet :

A la racine de notre projet vous trouverez un README (qui redirige vers ce rapport), une copie de ce rapport ainsi que la première version de notre algorithme Alpha-Bêta.

Dans le répertoire *jeu_arlecchino/* se trouvent les fichiers relatifs au fonctionnement du jeu ainsi que tous les joueurs dans le répertoire *joueurs/*.

Annexe 3 : Manuel d'utilisation

Choisir les joueurs d'une partie :

Afin de choisir les joueurs que vous voulez voir s'affronter il faut se rendre dans le fichier `jeu_arlecchino/main.cc` et, à la ligne 14, inscrire le nom de deux joueurs exi

```
14 | Arbitre a (player::ALPHABETA, player::RAND2,30);
```

Par défaut, le joueur 1 est le joueur Alpha-Bêta et le joueur 2 un joueur aléatoire.

Pour connaître la liste des différents joueurs jouables, il faut se rendre dans le fichier `jeu_arlecchino/arbitre.hh` ligne 32.

```
32 enum class player {MANUEL , MANUEL2, RAND, RAND2 , ALPHABETA , ALPHABETA2} ;
```

Lancer une partie :

Une fois que vous avez choisi les joueurs qui vont s'affronter, rendez vous dans le répertoire `jeu_arlecchino/` et ouvrez un nouveau terminal.

Effectuez ces 3 commandes :

- `cmake .`
- `make`
- `./stage_alphaBeta`

Des warnings peuvent éventuellement apparaître mais l'application n'en sera pas affectée.

Se familiariser avec l'interface :

Au cours d'un challenge, plusieurs parties vont être lancées à la suite. Au lancement de chacune d'entre elles, le numéro de la partie ainsi que son plateau initial sera affiché : ici c'est la partie n°30 et le pion maître à bien été retiré.

```
Partie n 30 :
| RBRV | BVRV | VBJR | BJBR | JJJJ | RJRR |
| BRBV | BJVJ | BVBB | RRRB | BBBR | JBJR |
| JRBR | BVBV | BJBB | RBRB | VJVR | RRRR |
| RVRJ | VVVV | VJVV | JRJR | JVJV | RRRV |
| VVRV | JRJV | BVVV | JBBJ | RJJJ | JBBJ |
| VBVJ | BBBB | RVRV | VJJJ | ---- | BVBJ |
```

Tour à tour, le nom du joueur qui joue apparaît puis le plateau est actualisé pour refléter le dernier coup joué.

```
tour : 1
A tour passe
|RBRV|BVRV|VBJR|BJBR|JJJJ|RJRR|
|BRBV|BJVJ|BVBB|RRRB|BBBR|JBJR|
|JRBR|BVBV|BJBB|RBRB|VJVR|RRRR|
|RVRJ|VVVV|VJVV|JRJR|JVJV|RRRV|
|VVRV|JRJV|BVVV|JBBB|RJJJ|JJBJ|
|VBVJ|BBBB|RVRV|VJJJ|----|BVBJ|
```

Dans l'exemple ci-dessus, c'est au joueur A (initial du joueur Aléatoire) de jouer, mais comme c'est au tour des pions bleus et qu'aucun coup n'est possible, il passe son tour.

La fin d'une partie est manifestée par ce message ci-dessous mentionnant le nom du joueur victorieux et le nombre de tours joués. Si la partie se termine prématurément (en cas d'un coup non rendu dans les temps), un message est aussi affiché comme ci-dessous.

```
mutex non rendu
Aleatoire gagne ! Nombre de tours : 2
```

Une fois toutes les parties jouées, un récapitulatif du nombre de victoires de chaque joueur vous est affiché.

```
FIN DU CHALLENGE
César gagne 0
Aleatoire gagne 30
```

Utiliser le joueur manuel :

Pour jouer avec le joueur manuel, il faut changer le temps d'un coup afin d'avoir le temps de réfléchir et de saisir le coup que l'on souhaite jouer. Pour cela, il faut se rendre dans le fichier *jeu_arlecchino/arbitre.hh* et saisir le temps souhaité, en l'occurrence, une vingtaine de secondes permet de jouer sans réfléchir, plus il y aura de secondes plus ce sera confortable de jouer, mais aussi plus la partie sera longue (l'adversaire utilisant le même temps que vous).

```
16  const int TEMPS_POUR_UN_COUP(1000); // millisecondes
```

Par défaut, les joueurs ont une seconde pour jouer.

tour : 2
Quel coup voulez vous jouer ?

Enfin, quand c'est au tour du joueur manuel de jouer, vous devez taper sur votre clavier le déplacement que vous souhaitez effectuer sous la forme : "0020" ou "0,0,2,0". Ces chiffres représentent des couples d'indices, le premier couple correspond aux coordonnées de la pièce que vous souhaitez jouer, le second aux coordonnées de votre premier déplacement etc... Une fois votre déplacement saisi, il faut appuyer sur Entrée pour valider le coup.

Modifier la profondeur de recherche du joueur Alpha-Bêta :

Il vous est possible de modifier la profondeur de recherche du joueur Alpha-Bêta dans le fichier *jeu_arlecchino/jeu.hh* à la ligne 8, plus vous mettrez une profondeur élevée plus le joueur Alpha-Bêta sera efficace mais il prendra en contrepartie plus de temps pour jouer. Ainsi si vous mettez une profondeur importante il est possible que vous deviez augmenter le temps pour un coup (vu plus haut). Par défaut la profondeur de recherche est à 4 (ce qui veut dire que Alpha-Bêta regarde si son coup est le bon sur 4 tours).

Annexe 4 : Diagramme de Gantt réel

