Projet C++
Puissance 4

Henri Qiu & Matthieu Palumbo

Sommaire

I- Raisons du choix du sujet

Nous avons choisi le Jeu du Puissance 4 car nous voulions nous focaliser sur l'aspect formateur du projet. Voulant tous les deux travailler dans le domaine financier plus tard, nous pensions au début faire un sujet relatif à la finance, estimant que ce serait plus en accord avec notre parcours. Nous avons finalement réalisé que développer un Jeu de Puissance 4 serait plus utile en terme de compétences acquises en programmation C++ et certainement plus ludique aussi. C'est pourquoi nous avons finalement opté pour ce sujet.

II- Règles du jeu du Puissance 4

Le « Puissance 4 » est un jeu de plateau en deux dimensions, contenant 42 cases agencées en 7 colonnes et 6 lignes. Il se joue à deux joueurs, chacun disposant de pions à positionner dans une colonne. A la différence d'un jeu comme le morpion, toutes les cases ne sont pas accessibles instantanément au Puissance 4. En effet, le joueur ne peut choisir que la colonne et non la ligne, l'effet gravitation étant une composante spécifique de ce jeu. Cela veut dire que le pion déposé dans une colonne tombe directement dans la ligne la plus basse non occupée. Ainsi pour pouvoir jouer dans une case vide, il faut que ce soit une case de la première ligne ou que la case juste en dessous soit déjà occupée. Le but du Puissance 4 est de constituer une suite de 4 pions d'affilée appartenant au même joueur. L'alignement peut être horizontal, vertical ou diagonal; il existe deux types de diagonales: la diagonale montante et la diagonale descendante. Le premier joueur a avoir constitué un alignement de 4 de ses pions a gagné la partie. Si lorsque toutes les cases ont été jouées, il n'y a pas de gagnant, il y a match nul.

III- Définition de la structure de jeu avec 2 joueurs humains

Notre première tâche a été de créer un programme qui permette à deux joueurs humains de jouer sur la console, ce qui est possible dans notre rendu final grâce à la classe **TwoPlayers**. Pour cela nous avons programmé les deux joueurs, en prenant soin de définir un type de marquage différent pour matérialiser les pions des deux joueurs. Cette étape est implémentée dans le constructeur de la classe, de même que dans **Connect4_AI** (mais avec un marquage pour le Bot cette fois-ci). Nous avons ensuite défini les règles du jeu vues précédemment, à savoir que le joueur choisit une colonne et le programme doit tenir compte de « l'effet gravitation » pour placer correctement le pion. Lorsque le joueur joue, nous avons donc pris en compte le numéro de la colonne qu'il entrait dans la console et affiché l'état du jeu à chaque tour grâce à notre interface graphique et la fonction ***OnRender*** (détail plus loin). Nous avons également une fonction ***IsColFull*** qui détermine si, oui ou non, la colonne sélectionnée était pleine. Une variable ***watchdog*** nous permet de ne pas faire jouer l'autre joueur si le joueur 1 a cliqué sur une colonne déjà pleine.

Nous avons ensuite défini l'état du jeu à la fin de chaque tour. Si la partie est finie, le programme doit s'arrêter, sinon il doit continuer. Il existe deux manières de terminer une partie. La première correspond à un match nul: si plus aucune case n'est disponible, l'ordinateur doit retourner match nul. La deuxième correspond à une victoire. Le programme doit vérifier tous les types de victoire possible et s'il en trouve un, il doit retourner le nom du gagnant. Les fonctions ***IsBoardFull***, ***IsGameWon*** et ***IsGameOver*** permettent de définir les différents états du jeu possible.

Cette première phase a été réalisée assez rapidement, les problèmes survenant étant principalement des fautes de syntaxe avec une mauvaise traduction des règles, une erreur dans les

caractéristiques des éléments (comme la numérotation des cases d'un tableau, commençant par les lignes et les colonnes 0), ou encore dans l'affichage du tableau (mettre les espaces correspondant pour aligner les lignes et les colonnes).

Une fois cette première phase achevée, nous pouvions jouer à deux joueurs humains dans la console, le programme exécutant correctement nos demandes et retournant les instructions correspondantes lorque la partie était finie ou qu'un coup n'était pas possible.

IV- Intelligence artificielle

1-Paramètres de base

Nous avons alors commencé la deuxième phase qui correspond à l'implémentation de l'intelligence artificielle, qui est nécessaire pour qu'un humain puisse jouer seul face à l'ordinateur. Cette phase est beaucoup plus complexe que la première car il faut que l'ordinateur prenne en compte les actions du joueur humain, les analyse et retourne un coup qui soit "intelligent". Pour que l'ordinateur puisse jouer en alternance avec un joueur, il faut qu'il sache quand c'est à lui de jouer et quand c'est au tour de son adversaire et alterner successivement les tours de chaque joueur. Dans la suite, nous avons défini des fonctions permettant d'altérner le tour de chaque joueur en définissant le tour d'un joueur une fois que son adversaire a joué et de différencier le type de marqueur selon les joueurs (par exemple mettre une croix au premier joueur et un cercle au second, ou les numéros 1 et 2). Ces fonctions permettent donc que chaque joueur joue une et une seule fois et que leurs pions soient de type différent pour permettre de reconnaître qui a joué et dans quelle case.

Ensuite, il nous faut définir l'effet gravitation lorque l'ordinateur joue pour qu'il comprenne qu'il n'a le choix que de la colonne et que la ligne lui est imposée par les règles du jeu. La fonction ***RowForPlay*** fait jouer l'ordinateur dans la ligne la plus basse libre de la colonne qu'il a choisi.

2- Mise en évidence des stratégies

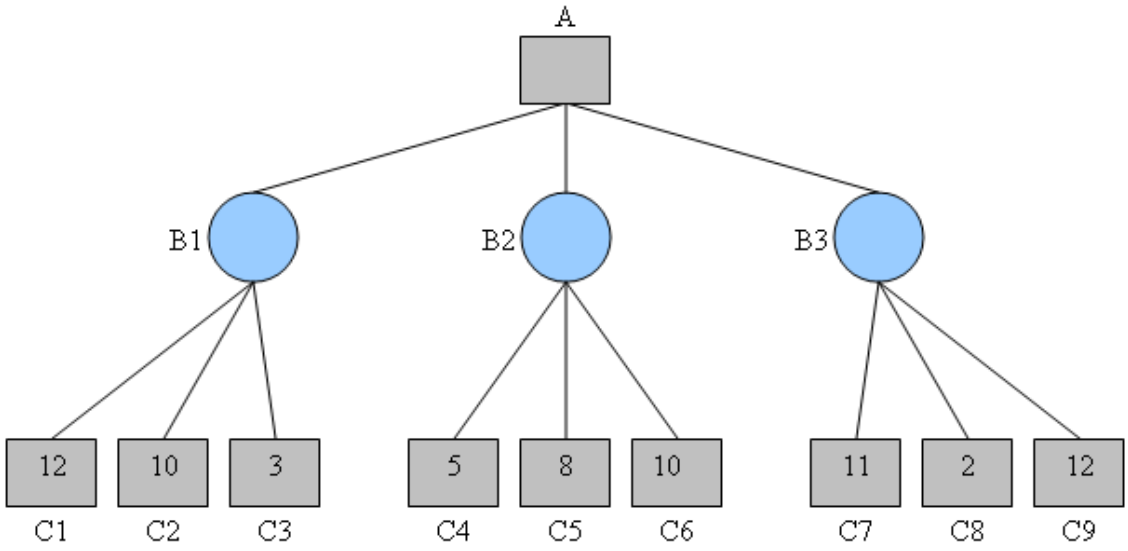
Vient ensuite le problème des stratégies qu'il faut coder pour que l'ordinateur soit efficace. Il doit pouvoir en quelque sorte réfléchir, tel un humain en ordonnant ses choix. Lorsque qu'un humain joue au « Puissance 4 », quand c'est à son tour de jouer, il analyse les différents coups possibles (7 en l'occurence) et joue celui qu'il juge le plus efficace. Mais toute la complexité réside dans la définition de l'efficacité. Chaque joueur logique regarde d'abord s'il peut gagner immédiatement, puis si ce n'est pas le cas, s'il peut perdre au prochain tour. Si ce n'est toujours pas le cas, il joue soit un coup d'attaque pour se donner la possibilité de gagner dans un futur proche, soit un coup de défense pour éviter que son adversaire gagne dans un futur proche. Mais ces derniers choix sont propres aux joueurs et varient selon leur niveau ou leur style de jeu. Ainsi, certains joueurs vont adopter une stratégie offensive, car ils préfèrent imposer leur jeu et mettre la pression sur leur adversaire et d'autres joueurs vont s'avérer plus défensifs et contrer leur adversaire en attendant que ce dernier joue un « mauvais coup ».

Mais dans tous les cas, l'analyse des différents coups possibles est une étape cruciale dans le processus de réflexion d'une partie de Puissance 4. L'humain vérifie instinctivement les choix qui s'offrent à lui, en plaçant virtuellement un de ses pions dans chaque colonne pour observer le résultat. Mais si l'humain peut faire cette expérience de pensée, l'ordinateur, lui, doit pouvoir matérialiser ces différentes possibilités. Cette étape de matérialisation des choix possibles se fait au travers de deux fonctions dans notre code. La première, ***SetCell*** permet d'ajouter un pion dans une colonne et ***EraseCell*** permet d'effacer le marquage d'un pion. Ces deux fonctions entrent en jeu lorsque la méthode de *scoring* est définie et donc que chaque coup potentiel se définit par un score qui correspond à un nombre de points. Lorsqu'il commence à jouer, l'ordinateur veut maximiser son

score. Pour ce faire, on va d'abord créer une copie du jeu sur laquelle l'ordinateur ajoute un de ses pions dans une colonne avec la fonction ***SetCell***, calcule ensuite le score qu'il obtient, puis efface ce pion, qui correspond au dernier pion joué dans la colonne choisie, avec la fonction ***EraseCell***. Il procède ainsi pour toutes les colonnes et joue finalement la colonne qui a obtenu le score le plus élevé. Le principe est le même lorsque c'est le joueur humain qui commence, sauf que l'ordinateur minimise le score de son adversaire. Cette démarche est effectuée dans la fonction ***MakeBotPlay*** et finalisée par la fonction ***BotPlay*** qui joue le coup.

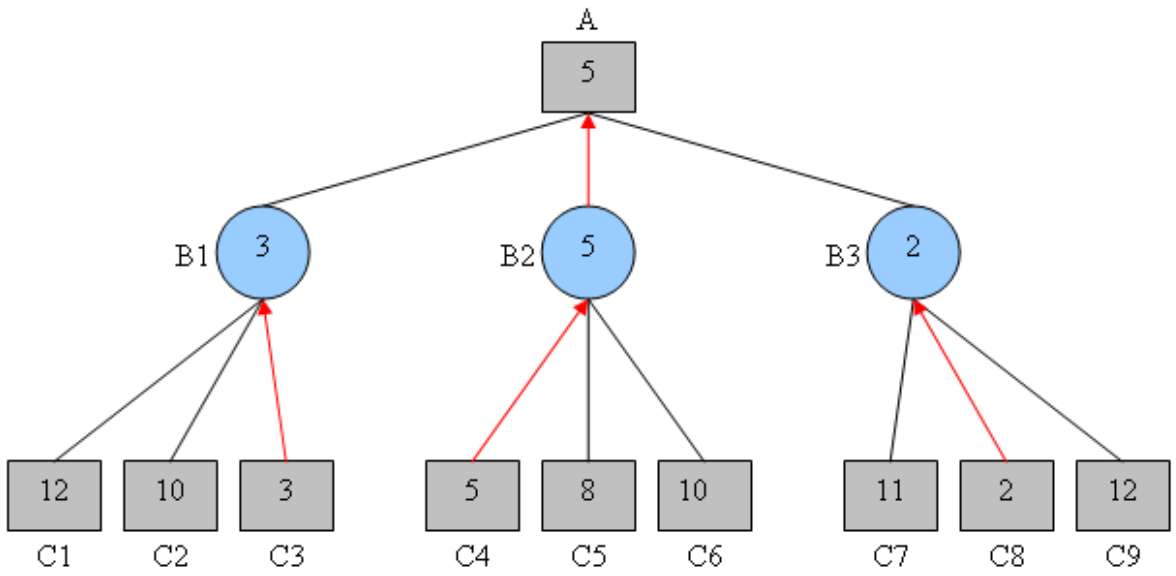
3- Algorithme minimax

Pour plus de clareté, nous définirons dans la suite du raisonnement le joueur 1 comme celui qui commence et le joueur 2 comme celui qui joue en deuxième. Ainsi, lorsque le joueur 1 joue, pour chaque coup possible, il analyse les répliques éventuelles du joueur 2, qui lui-même analyse pour chacune d'entre elles celles que le joueur 1 peut effectuer, et ainsi de suite. Ce processus de réflexion peut être mis en oeuvre avec un algorithme minimax, particulièrement adapté pour des jeux se jouant à tour de rôle. Il peut également se réprésenter sous la forme d'un arbre avec une certaine profondeur. La profondeur correspond au nombre de tour qu'on est capable de prévoir en avance. Pour un humain, cette profondeur dépend de son expérience du jeu mais surtout de sa capacité intellectuelle. Pour un ordinateur, la profondeur est limitée par son temps de calcul. Pour illustrer nos propos, représentons un arbre de profondeur 2 avec 3 coups possible à chaque tour (nous rappelons que dans le cas du Puissance 4, suivant que les colonnes sont remplies ou non, il peut y avoir jusqu'à 7 coups possibles). Les différents coups possibles sont représentés par des noeuds, un noeud père donnant plusieurs noeuds fils.



Le noeud A, de profondeur 0 est appelé racine et représente l'état actuel du jeu. Le noeud A possède 3 noeuds fils B1, B2 et B3, de profondeur 1 qui sont les coups possibles du joueur 1. Chacun de ces 3 noeuds possède 3 noeuds fils de profondeur 2 (de C1 à C9) qui sont les coups possibles de l'adversaire du joueur 1, c'est-à-dire le joueur 2. Les noeuds de profondeur correspondant à la profondeur de l'arbre sont appelés feuilles et leur valeur est calculée à l'aide d'une fonction d'évaluation (nous expliciterons celle que nous avons choisi dans la suite). Les cercles bleus correspondent aux coups du joueur 1 et les rectangles gris à ceux du joueur 2. Le principe de l'algorithme minimax réside dans le fait que le joueur 1 va maximiser son score, tout en sachant que son adversaire, le joueur 2, va le minimiser. Ainsi, si le joueur 1 joue B1, le joueur 2 va choisir le coup qui minimise le score du joueur 1, à savoir C3 de valeur 3 et donc le score de B1 est 3. De même, les

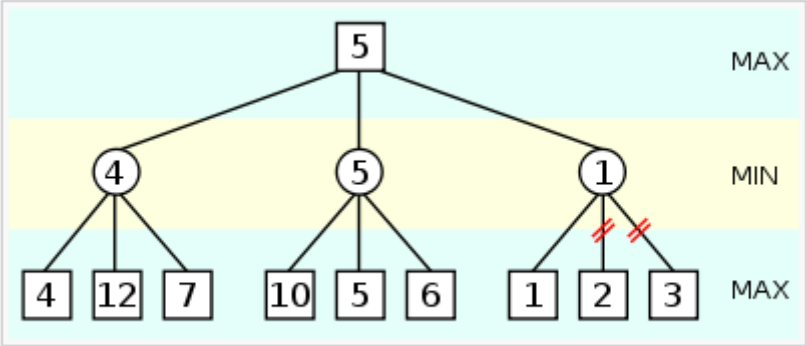
scores de B2 et B3 sont respectivement 5 et 2. Le joueur 1 choisit alors le coup B2 qui maximise son score; la valeur du noeud A est donc 5. Graphiquement, cela donne:



L'implémentation de l'algorithme minimax étant une des deux parties centrales de l'intelligence artificielle, il nous a fallu un certain temps afin de le comprendre et de le mettre en œuvre correctement. Dans notre code, la fonction correspondante est nommée ***Alpha_Beta*** nous verrons dans la section suivante l'amélioration concernant l'élagage alpha-beta). Une des difficultés réside dans le fait que cet algorithme est récursif, ce qui n'est pas toujours évident à comprendre, en faisant diminuer de 1 la profondeur de la fonction à chaque étape récursive. La fonction ***Alpha_Beta*** va de paire avec la fonction ***MakeBotPlay*** qui définit la colonne que l'ordinateur doit choisir et qui représente la deuxième partie centrale de l'intelligence artificielle. La difficulté, que nous n'avions pas perçue au début, réside dans le fait qu'il y a un joueur qui maximise (le premier joueur) et un qui minimise (le deuxième joueur). Ce n'est qu'une fois que nous avons compris et mis en place cette notion que l'intelligence artificielle commençait à retourner des coups qui paraissaient raisonnables.

4- Elagage alpha-beta

Un problème qui s'est ensuite posé était la profondeur de calcul. Plus elle est élevée et plus l'intelligence artificielle anticipe les coups et donc est "intelligente". Mais à partir d'une certaine profondeur, le temps de calcul était vraiment trop long. En effet, dans un « Puissance 4 » avec 7 coups possibles, augmenter la profondeur de n-1 à n nécessite le calcul de 7^n scores supplémentaires, qui eux-mêmes peuvent s'avérer plus ou moins long. Nous avons donc décidé de mettre en place la technique d'élagage alpha-beta, ou alpha-beta pruning en anglais. L'algorithme minimax effectue une exploration complète de l'arbre à la profondeur demandée, mais cela n'est pas nécessaire. Une exploration partielle est le plus souvent suffisante. Pour cela, il faut déterminer des seuils maximum et minimum à partir desquels la génération de noeuds ou de parties de l'arbre est inutile. Pour clarifier nos propos, nous proposons d'étudier un cas simple, constitué d'une racine avec trois noeuds fils qui eux-même possèdent trois noeuds fils potentiels. La profondeur de l'arbre est donc de 2 et le premier joueur maximise le score.



Élagage alpha-beta

A la profondeur 1, c'est le joueur 2 qui joue, il va donc minimiser les scores des feuilles de profondeur 2. Pour le troisième noeud de profondeur 1, comme la première feuille vaut 1 et que le joueur 2 minimise, sa valeur sera inférieure ou égale à 1. Or comme les valeurs des deux autres noeuds de profondeur 1 sont respectivement 4 et 5, il est inutile de prendre en compte les feuilles suivantes (représenté par des traits rouges sur le graphique). En effet, le premier joueur maximise et donc le troisième noeud de profondeur 1 ne sera jamais choisi par ce joueur car sa valeur est inférieure ou égale à 1. Ainsi, dans cet exemple, l'élagage alpha-beta permet de ne pas calculer les feuilles de valeur 2 et 3. Mais imaginons que la profondeur soit plus grande, les sous arbres partant de ces noeuds n'ont pas besoin d'être calculés, ce qui peu représenter une économie de calcul assez conséquente. Ainsi, l'élagage alpha-beta permet d'optimiser le temps de recherche par rapport à un algorithme minimax classique. Cette technique est bien évidemment implantée dans notre fonction ***Alpha_Beta***.

5- Evaluation heuristique

Pour que l'intelligence artificielle puisse attribuer des scores aux différents coups, il ne reste plus qu'à définir une évaluation heuristique, qui est définie par la fonction Eval dans notre code. Sa détermination est essentielle car des évaluations différentes conduisent à des choix différents. Il est donc important que ces choix soient logiques. En raisonnant de manière mathématique, la technique de scoring la plus adaptée semble être symétrique. Un coup qui nous est bénéfique va mettre notre adversaire dans l'embarras. Par exemple, un coup qui me rapporte n points va rapporter -n points à mon adversaire. Ainsi, soit un coup me donne l'avantage, soit il le donne à mon adversaire, soit dans quelques cas, il est neutre.

La méthode la plus simple à laquelle nous avons pensé au début est une évaluation heuristique gagné / perdu, c'est-à-dire qui me rapporte un nombre de points très grand si j'ai gagné (10000 par exemple), un nombre de point très négatif si j'ai perdu (-10000) et 0 sinon. Cette technique ne permet d'être efficace seulement si une victoire est possible pour la pronfondeur de jeu choisie et dans les autres cas, elle est inutile, le score renvoyé étant toujours 0.

Nous avons ensuite décidé de développer un peu plus notre fonction heuristique en ajoutant un décompte des séries de 2 ou de 3 jetons pour chaque joueur, ce qui est fait respectivement par les fonctions ***CheckTwos*** et ***CheckThrees*** dans notre code. Ainsi, si j'ai 3 jetons d'affilée, que ce soit horizontalement, verticalement ou diagonalement (en montée ou en descente), l'évaluation retourne une certaine valeur. On effectue le même raisonnement avec les séries de 2 jetons.

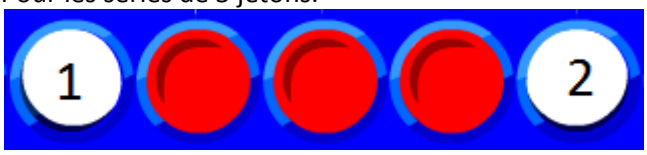
Cependant, on voit bien qu'il faut ordonner ces différentes séries. Un classement logique en terme de score est série de 4 (victoire) > série de 3 > série de 2. Nous avons donc décidé d'attribuer un score de 10000 en cas de victoire, -10000 en cas de défaite, pour que la victoire ou l'empêchement de la défaite soient préférés dans tous les cas. Dans les cas où nulle victoire n'est possible, nous avons défini le score comme:

score = 10*série2_j1 + 100*série3_j1 - 10*série2_j2 - 100*série3_j2

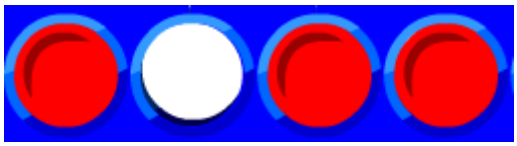
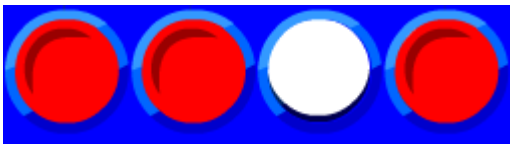
avec série2_j1 le nombre de séries de 2 jetons d'affilée horizontalement, verticalement et

diagonalement du joueur qui appelle la fonction d'évaluation heuristique, série3_j1 étant la même fonction mais pour les séries de 3 jetons d'affilée. Les fonctions série2_j2 et série3_j2 sont définies de la même manière mais représentent le nombre de séries respectivement de 2 et 3 jetons d'affilée mais pour l'adversaire du joueur appelant la fonction d'évaluation heuristique. En appliquant cette méthode, notre évaluation heuristique était plus précise mais nous voulions encore l'améliorer.

En effet, nous avons remarqué que la prise en compte des séries de 2 ou de 3 jetons d'affilée qui ne pouvaient pas donner suite à une série de 4 était inutile car elles ne pouvaient pas procurer la victoire. Nous avons alors défini les séries de 2 et 3 jetons d'affilée suivantes. Pour simplifier l'explication, nous ne prendrons en exemple que les séries horizontales, mais un raisonnement identique s'applique pour les séries verticales, diagonales montantes et diagonales descendantes. Nous définissons notre évaluation heuristique à l'aide d'exemples pour faciliter la compréhension. Pour les séries de 3 jetons:

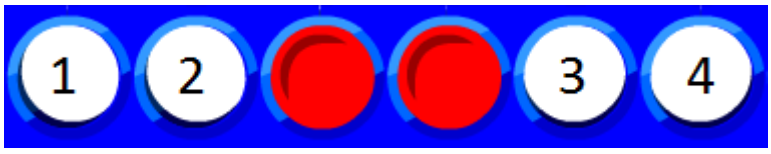


Si la case 1 ou la case 2 est libre, une série de 3 est attribuée, mais la les cases 1 et 2 sont libres, le score de deux séries de 3 est attribué.

 ou 

S'il y a 3 jetons de même couleur sur 4 cases d'affilée avec une case vide au milieu, un score de série de 3 est attribué.

Pour les séries de 2 jetons, le raisonnement est le même mais il y a plus de cas possibles:



Si les cases 1 et 2, ou 2 et 3, ou 3 et 4 sont vides, alors un score d'une série de 2 est attribué.

Si les cases 1 et 2, ou 2 et 3 sont vides, alors un score d'une série de 2 est attribué.

Si les 2 cases centrales sont vides, alors un score d'une série de 2 est attribué.

En définissant notre évaluation heuristique ainsi et en gardant la méthode de scoring qui donne un poids plus important aux séries avec le plus grand nombre de jetons (10000 pour les séries de 4, 100 pour celles de 3 et 10 pour celles de 2), l'intelligence artificielle semble jouer "intelligemment" pour des profondeurs de jeu respectables.

V- Partie SDL

SDL fut sans aucun doute la partie la plus difficile à mettre en œuvre. Ayant terminé l'élaboration du code, nous pensions qu'une représentation graphique, avec des pions de couleurs et un plateau de jeu convenable, renforcerait l'aspect ludique du jeu.

Les principes inhérents à SDL sont relativement intuitifs bien que l'écriture du code en lui-même relevait d'une difficulté supplémentaire. En effet, il fallait s'adapter aux nouvelles fonctions du *package* et en comprendre les usages, si bien qu'il nous a fallu un moment d'adaptation avant de pouvoir maîtriser les bases. Au terme de ce projet, nous pouvons nous vanter de maîtriser les aspects les plus importants, notamment la gestion des événements qui est simplifiée grâce aux types et fonctions inhérentes au *package*. Nous regrettons cependant que SDL manquât de fonctionnalités comme les formulaires ou les menus déroulants, car nous aurions pu ainsi les utiliser pour que l'utilisateur puisse faire le choix du niveau de difficulté, par exemple. Au lieu de cela, nous avons recouru à Paint (ou Paintbrush sur Mac) afin de modéliser les boutons sur l'interface graphique, ce qui est relativement « artisanal » et plutôt regrettable.

Nous avons donc importé SDL et défini des surfaces (***SDL_Surface***) en tant qu'attributs de classe afin de prendre en compte tous les *sprites* donc nous avions besoin. De même, nous avons défini des ***SDL_Rect*** et donné des coordonnées de pixels dans notre fonction *Reset.* L'objectif était pour nous de faciliter la gestion des événements, comme nous allons le voir par la suite.

Nous avons cinq fonctions qui renvoient directement aux utilisations du package *graphique* : ***OnDraw***, ***OnExecute***, ***OnInit***, ***OnRender*** et ***OnCleanup***.

OnInit est appelé au début du programme afin de lire les *sprites et* vérifier que ceux-ci ont bien été lus. Si ce n'est pas le cas, le programme s'arrête.

Nota Bene 1 : Attention si vous compilez le code de chez vous. En effet, les *sprites* sont appelés depuis un répertoire personnel, si bien que vous ne pourrez probablement pas retrouver le chemin que nous avons défini. N'ayant pas trouvé la fonction qui pourrait retourner automatiquement le chemin du fichier *.exe*, nous vous sommes gré de bien vouloir changer les chemins (*paths*) pour tous les *OnLoad* que nous aurions utilisé.

OnDraw est une fonction qui colle (« *blit* ») la nouvelle surface sur la surface précédente. Il s'agit d'un équivalent de *SDL_BlitSurface* mais dont les arguments sont plus « intuitifs ».

OnExecute constitue le cœur de la gestion des événements. Lorsque l'utilisateur clique sur la souris ou sur un bouton, le programme interprète l'événement et effectue les actions adéquates. Parfois, étant donné que nous faisons communiquer plusieurs pages, nous imbriquons les boucles « while » ce qui explique la longueur du code et la pénibilité de lecture de cette fonction en particulier.

OnRender gère l'affichage. Etant donné que nous avons travaillé sur le plateau de jeu grâce à un tableau, nous actualisons grâce à cette fonction le « rendu graphique », c'est à dire que si une case est jouée par le joueur « rouge », alors OnRender affichera un bouton rouge en lieu et place de la case concernée. A la fin, ***SDL_Flip*** nous actualise l'affichage.

OnCleanup gère la mémoire. En effet, lorsque nous changeons d'interface, nous ne voulons pas conserver inutilement les surfaces en mémoire, si bien que nous les nettoyons ; c'est ce que fait cette fonction.

L'on remarquera que l'utilisation du GUI (interface graphique) alourdit considérablement le code, étant donnée la nécessité de gérer les boucles « while » ainsi que la définition préalable des positions (.x et .y) des SDL_Rect. Par exemple, la fonction ***Reset*** de Start.cpp définit toutes les positions des boutons, ce qui peut être très confus pour les lecteurs car ils ne sauront pas d'où les chiffres proviennent. La méthode est, une nouvelle fois, très artisanale : nous avons simplement créé une image 640x480 et avons fait du repérage sur GIMP afin de définir les coordonnées des boutons. Ainsi, pour comprendre que le joueur a cliqué sur le bouton gauche de la souris, le code détecte si la souris se trouve dans les coordonnées afférentes à un bouton.

Nous avons enfin exploité la gestion temporelle qui est un apport intéressant de SDL. Ainsi, lorsqu'un joueur gagne la partie, la combinaison gagnante est affichée pendant 3 secondes sur l'écran, puis l'écran GameOver apparaît. Cela permet au joueur de revenir au menu principal ou de quitter définitivement le jeu.

Nota Bene 2 : avec le temps qu'il nous a fallu pour maîtriser SDL et mettre le tout en place, nous nous rendons compte, à l'heure où sont écrites ces lignes, que nous aurions gagné à définir des instructions plus précises pour le jeu, même si le tout reste intuitif. Le jeu se joue à la souris, nul besoin d'utiliser le clavier ; il vous suffit de cliquer là où vous souhaitez vous rendre ; durant le jeu vous pourrez cliquer sur n'importe quelle case et le coup sera joué.

Nota Bene 3 : lors du lancement du jeu, vous vous rendrez compte que l'écran d'accueil comporte 4 pions, dont l'une qui porte la mention « Niveau de Difficulté ». En cliquant dessus, il ne se passe rien ; il s'agit d'un défaut de notre part, puisqu'au moment de définir l'architecture du jeu, nous avions pensé qu'il serait intéressant de laisser au joueur le choix du niveau de difficulté avant de lancer le mode Arcade. Mais au fur et à mesure du développement, nous avons pris conscience qu'il était plus facilement gérable de sélectionner le niveau de difficulté après avoir cliqué sur le mode Arcade. Mais comme il était déjà trop tard pour *designer* un nouveau Menu principal, nous l'avons laissé comme tel. Nous vous prions de nous excuser.

Conclusion

Au terme de notre projet, nous ne saurions retenir un sentiment de soulagement et de satisfaction.

Nous ne regrettons pas d'avoir choisi ce projet qui s'avérait loin d'être trivial : passée la première étape du jeu sur la console où chacun jouait son tour, l'implémentation de l'intelligence artificielle nous a fait sentir toute la complexité des algorithmes récursifs de calcul. Plus le code s'allongeait, plus il était difficile parfois de trouver les erreurs et nous avons passé beaucoup de temps à retrouver les erreurs de calcul qui ont pu naître dans notre algorithme. Puis vint l'implémentation graphique qui fut une étape très formatrice, puisqu'elle a rajouté une dimension non négligeable de challenge pour des débutants, non familiers de C++ à l'origine. In fine, outre quelques défauts graphiques, nous sommes globalement satisfaits du rendu final.

Nous n'avions dès le départ pas la prétention de rendre un projet révolutionnaire, à l'instar de nombre de nos camarades dont les projets étaient certainement plus ambitieux que le nôtre. En lisant notre code, un lecteur avisé chevronné du C++ sera certainement consterné par les longueurs inutiles. Mais, débutants que nous fûmes et sommes toujours, nous reconnaissons que le C++ est un langage difficile, dont les complexités sont telles qu'il est ardu de tout bien maîtriser en l'espace d'un semestre. A cet égard, le lecteur de notre code se rendra certainement compte de l'absence de classes d'héritage alors que la multiplicité des types de jeux (contre Bot, ou contre Joueur 2) était justement propice à une telle architecture. Mais nous y avons renoncé pour la simple raison que nous avions peur de mal maîtriser ce sujet, qui constituait déjà pour nous la partie « difficile » du cours. Si bien que nous avons préféré nous en tenir aux structures de base, de sorte à rendre un code compilable et sans erreurs syntaxiques.

Quoiqu'il en soit, nous espérons que notre jeu plaira au plus grand nombre et que les futurs joueurs trouveront autant de plaisir à y jouer que nous, qui l'avons codé. Un plaisir véritable est né de cette expérience de développement, nouvelle pour nous, avec le rendu d'un projet qui nous semble abouti. nous en profitons ici pour adresser nos remerciements à M. Matthieu Durut, notre professeur, qui nous a fait profiter de son talent pédagogique pour comprendre de manière intuitive les tenants et aboutissants de ce « mur » qu'est le C++. Nous remercions également M. Gabriel Boya, qui nous a aiguillé pendant notre projet, nous sortant ainsi de certaines impasses difficiles.