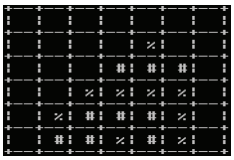
PROJET C++ DE SECONDE ANNÉE - 2013/2014

JEU DE PUISSANCE 4

CLAUDE PETIT ET MATTHIEU PLUNTZ, ENCADRÉS PAR JULIEN CAZORLA



Introduction et rapide historique

Le jeu de puissance 4 (« connect 4 ») a été édité en 1974 par Milton Bradley et commercialisé par Hasbro. Il s'agit d'un jeu à deux joueurs, à somme nulle et à information parfaite. La grille verticale du jeu standard possède 7 colonnes et 6 lignes. Chaque joueur dispose de 21 pions (rouge ou jaune) et joue à tour de rôle, en laissant tomber un pion dans une colonne. Le pion occupe alors la dernière case libre de la colonne. Le premier joueur qui aligne de façon contigüe 4 de ses pions horizontalement, verticalement ou en diagonale, gagne la partie. Si aucun alignement de 4 pions n'apparaît au 42^{e} coup, la partie est nulle. En 1988, Victor ALLIS a démontré que le jeu était à stratégie gagnante: le joueur qui débute gagne s'il suit la bonne stratégie.

Le nombre de coups possibles est inférieur à 42, et le nombre de parties possibles est donc au plus 3^{42}, puisque chaque case peut se trouver dans trois états (vide, rouge ou jaune). En fait, un certain nombre d'états ne sont pas atteignables lors d'une partie régulière et l'on donne comme ordre de grandeur du nombre de parties possibles environ 7×10^{13}. Nous verrons plus loin que l'on peut coder une partie sur moins de 128 bits. Pour répertorier l'ensemble des parties possibles, il faut donc une quantité de mémoire de l'ordre de 4 teraoctets. Il y a dix ans, il était impossible à un ordinateur domestique de stocker une telle masse de données; aujourd'hui, quatre disques durs suffisent. Pour des raisons financières, mais surtout par soucis d'élégance, nous n'implémenterons pas l'algorithme de force brute qui résoudrait le jeu par cette méthode.

Le rapport s'articule comme suit: la 1re partie présente rapidement le module métier du programme et expose les choix et le contenu des classes. La 2^{e} partie présente l'implémentation graphique, effectuée avec la bibliothèque OpenGL. La 3^{e} partie présente le moteur d'intelligence artificielle. La 4^{e} et dernière partie présente les tests effectués, les résultats obtenus et compare les différents algorithmes de jeu. Enfin, une courte conclusion expose les perspectives d'amélioration du programme.

1 Structure du programme et module métier.

1.1 Les classes.

Le programme se découpe naturellement en trois modules: un module dédié à l'intelligence artificielle, un module dédié à l'interface graphique et un module métier que nous présentons dans ce paragraphe. On pourra se reporter au diagramme des classes donné en annexe du rapport (nous ne détaillerons pas, sauf cas particulier, les constructeurs, destructeur, accesseurs et mutateurs; le code source est convenablement commenté et permet de comprendre le rôle de chaque méthode).

Lors d'une partie, nous souhaitons pouvoir jouer un pion, éventuellement annuler le dernier coup joué, sauvegarder ou charger une partie, rejouer une partie au coup par coup, recommencer une partie en choisissant deux joueurs humains, un humain contre l'ordinateur ou l'ordinateur contre lui-même. Ce qui précède et la description physique du jeu nous amène donc à considérer les classes suivantes:

• Une classe `Pion`, carctérisée par une `couleur`, un numéro `num` et dotée d'un champ de classe qui sert de compteur à l'ensemble des pions.

• Une classe `Case` qui peut être vide ou contenir un pion. Son état est donné une chaîne de caractères (vide, jaune ou rouge) et elle possède un attribut `pion`.

• Une classe `Colonne` dont les attributs sont le nombre de pions contenus dans la colonne, le numéro de la colonne et un tableau de cases de taille HAUTEUR décrivant le contenu de chaque case de la colonne. Il est possible de mettre ou d'enlever un pion dans la colonne, d'obtenir le contenu d'une case à hauteur donnée et de savoir si la colonne est pleine.

• Une classe `EtatCourant` formée d'un tableau de colonnes de taille LARGEUR, d'un attribut `gagne` indiquant si un alignement de 4 pions est présent dans la grille, d'un tableau répertoriant les 69 positions gagnantes du jeu standard et d'un tableau de deux pointeurs sur des entiers de 64 bits, appelé `etat`, qui résume de façon condensée l'état de la grille de jeu lors d'un tour donné; nous reviendrons sur ce dernier attribut plus loin. La classe est responsable de l'évolution de l'état de la partie, elle gère donc l'ajout ou la suppression d'un pion durant un tour de jeu (et la mise à jour des attributs

correspondants) et la surveillance de la fin de partie.

• Une classe `Partie` qui gère les deux joueurs, leur tour de jeu, le lien entre le coup d'un joueur et son effet sur la grille de jeu courante, la sauvegarde et le chargement d'une partie, l'historique et le nombre des coups joués depuis le début de la partie. Elle possède (entre autres) un attribut `situation` qui est de type `EtatCourant`.

• Une classe `Fenetre` qui gère tout ce qui concerne l'affichage et la machine à états OpenGL.

• Une classe `Joueur` dotée d'un numéro, d'un attribut `nom`, d'un attribut entier `coup` qui représente le numéro de la colonne dans laquelle va jouer le joueur, d'une couleur et d'une liste de pions. Cette classe possède une méthode virtuelle `jouer`.

• Deux classes `Humain` et `IA` qui héritent de la classe joueur et implémentent chacune une méthode `jouer` différente.

D'autres classes seront présentées dans le paragraphe dédié à l'intelligence artificielle.

1.2 Nos choix et la philosophie du programme.

Nous avons souhaité, dans ce programme, ne pas avoir à tester sans arrêt les alignements horizontaux, verticaux, diagonaux ou antidiagonaux et multiplier ainsi les boucles. De plus, le moteur d'intelligence artificielle doit parcourir de façon efficace l'arbre des positions possibles et il est donc vital d'optimiser la taille de la représentation d'une partie. La méthode la plus courante (de la littérature) pour représenter de façon concise la grille de jeu est d'utiliser une grille binaire. Nous présenterons cet objet dans le paragraphe 3; notons simplement que son utilisation nous permet d'effectuer dans la partie métier, sans aucune boucle, l'ajout ou la suppression d'un pion, ou encore le test de fin de partie.

La possibilité d'annuler un coup impose le fait de pouvoir enlever un pion d'une case pour le remettre dans la liste des pions d'un joueur. L'attribut `pion` d'une case devrait donc pouvoir être vide. Cela nous conduirait à définir un pion vide et nous n'avons pas souhaité implémenter cette solution. À la place, nous modifions seulement l'état de la case lorsque le pion est retiré, sans toucher à l'attribut `pion`. Aucune fonction ne peut alors plus y accéder, tant que l'attribut `etat` reste vide. Le pion initialement dans la case, a été remis dans le vecteur des pions du joueur correspondant.

Il existe des façons clairement plus efficaces de coder un puissance 4, mais nous avons souhaité jouer le jeu de la programmation orientée objet. C'est donc cet aspect qui nous a guidé dans la conception de la partie métier, tandis que le soucis d'optimisation nous a guidé dans la partie intelligence artificielle.

2 Interface graphique.

2.1 Pourquoi OpenGL ?

Par curiosité, puisqu'il s'agit sans doute de l'interface graphique la plus connue et la plus utilisée en informatique (la plupart des cartes graphiques implémentent de façon matérielle les primitives openGL et la plupart des jeux vidéos possèdent un moteur graphique dédié openGL). Mais cette bibliothèque a la réputation (justifiée) d'être peu pratique et contraignante, s'intègre mal à la programmation orientée objet (POO par la suite) et est maintenant (très) ancienne (elle est apparue en 1992, dans les stations graphiques Silicon Graphics indigo, O_2 ou octane).

Si le projet était à refaire, nous choisirions sans doute SDL. Le temps que nous avons passé pour réussir à ouvrir une simple fenêtre et y afficher quelque chose se compte en semaines. Cela ne se fait pas de se plaindre dans un rapport, mais c'était tellement pénible qu'il nous semble légitime d'en parler.

2.2 Le fonctionnement d'OpenGL et de GLUT.

Les primitives openGL s'exécutent via une machine à états, de façon séquentielle (c'est en fait un « shell ») et toute l'interface avec le système d'exploitation windows se fait par l'intermédiaire d'une autre bibliothèque appelée GLUT (pour « GL utilities »). GLUT gère l'ouverture et la gestion de la fenêtre windows, la gestion des menus, les évènements liés au clavier et à la souris, tandis que le moteur openGL s'occupe uniquement de l'affichage des objets graphiques.

L'initialisation se fait à partir de fonctions GLUT qui déterminent la taille de la fenêtre, la couleur de fond, le mode d'affichage, etc. Une fois tous ces paramètres fixés, la machine à états est lancée par l'instruction `glutMainLoop()`. À partir de ce moment, toute interaction avec le programme est gérée par l'intermédiaire de fonctions « callback ». Ce sont des pointeurs sur fonction qui admettent en argument une fonction qui sera exécutée lorsque le clavier est utilisé, lorsqu'un clic a lieu depuis la souris, ou bien encore en tâche de fond. La fonction en paramètre doit avoir un prototype bien précis qu'il n'est pas possible de modifier. En particulier, ces fonctions ne peuvent pas être les méthodes d'une classe.

La totalité de la gestion graphique est effectuée dans la classe `fenetre` et les contraintes que nous venons d'évoquer nous ont obligé à utiliser, dans cette seule classe, des variables globales. Chaque fonction est correctement commentée et prévient de leur utilisation.

3 Le moteur d'intelligence artificielle.

3.1 Les structures de données utilisées.

On peut représenter une partie à l'aide de deux entiers de 64 bits (on pourrait même faire mieux) de la façon suivante: chacune des cases de la grille de jeu standard est numérotée selon le schéma ci-dessous.

.	.	.	.	.	.	.
5	12	19	26	33	40	47
4	11	18	25	32	39	46
3	10	17	24	31	38	45
2	9	16	23	30	37	44
1	8	15	22	29	36	43
0	7	14	21	28	35	42

Les positions 6, 13, ..., 48 servent à éviter les effets de bord lorsque des tests d'alignement sont effectués. Chaque case numérotée peut contenir un pion rouge, jaune ou être vide. Pour coder trois états, il faut deux bits. Mais deux bits suffisent également à coder quatre états. On peut donc considérer deux grilles binaires; la première contiendra les positions des pions rouges (la case i vaudra 1 si elle contient un pion rouge et 0 si elle est vide) et la seconde grille contiendra les positions des pions jaunes. Chacune de ces grilles peut aussi être considérée comme un entier de 49 bits dont le développement binaire donne la position des pions d'un joueur (les positions non utilisées sont fixées à zéro). Ainsi, deux entiers de 64 bits suffisent à coder une position; c'est le rôle de l'attribut `etat` de la classe `EtatCourant` (tableau de deux cases, chaque case pointant vers un entier de 64 bits représentant respectivement l'état de la partie du joueur 0 et celle du joueur 1).

Avec cette représentation, l'ajout ou le retrait d'un pion s'effectue avec une unique opération combinant un décalage binaire et un ou exclusif. Les tests d'aligenements de 3 ou 4 pions se font également sans aucune boucle par des opérations binaires.

Initialement, nous avions créé un attribut contenant toutes les positions gagnantes du jeu. Il en existe 69 pour une taille de jeu standard, mais l'on peut facilement calculer par récurrence le nombre de positions gagnantes quelles que soient HAUTEUR et LARGEUR. Ce tableau était rempli en début d'exécution de programme par lecture dans un fichier texte qui contenait les entiers représentant les alignements gagnants de quatre cases. Le test de fin de partie se faisait en comparant chaque case à la position du joueur, pour une complexité correspondant au parcourt des 69 cases. Nous avons finalement trouvé une méthode plus rapide ne nécessitant aucune boucle (cf. références en fin de rapport). Le tableau de positions gagnantes est toujours présent dans le code source sous forme de commentaires, en cas de besoin.

Les instances de la classe `IA` gèrent le jeu d'une IA contre un joueur humain ou contre une autre IA. La méthode `jouer` renvoie alors directement la colonne où le pion doit être déposé, au lieu de la valeur -1 par défaut restituée par un joueur humain. Cette valeur par défaut provoque l'attente d'un clic de souris déclenchant le coup humain, tandis que le coup d'une IA est affiché directement.

La classe `IA` possède un attribut de type (enum) `ALGO` qui caractérise l'algorithme utilisé pour jouer. Nous avons implémenté un algorithme naïf qui joue au hasard et un algorithme négamax, mais on pourra éventuellement implémenter des algorithmes de Monte Carlo ou bien l'algorithme de Victor ALLIS. Ce dernier n'utilise pas de parcours d'arbre mais des règles de jeu à valider.

La classe `AlphaBeta` implémente un algorithme de parcours d'arbre de type minimax, ainsi que sa fonction d'évaluation. Elle possède comme attribut le coup à effectuer et comme méthodes une fonction récursive `parcours` effectuant l'exploration de l'arbre et une fonction `eval` évaluant chaque position. D'autres méthodes sont appelées par `eval` pour tester les alignements de pions et les menaces potentielles.

3.2 L'algorithme négamax avec élagage $\alpha - \beta$.

Nous rappelons brièvement que l'algorithme minimax évalue toutes les positions possibles dans l'arbre de jeu, par l'intermédiaire d'une fonction d'évaluation associant un score à chaque position. Le coup choisi par le joueur est celui qui maximise, à un tour donné, la fonction d'évaluation. On suppose que l'adversaire joue au mieux, c'est à dire qu'il minimise cette fonction lorsqu'il joue. On dispose donc d'un algorithme récursif qui alterne phases de minimisation et de maximisation.

Si l'on symétrise le score par rapport à 0, le processus devient identique pour les deux joueurs et le codage s'effectue avec une unique fonction. Il s'agit de l'algorithme négamax. D'un point de vue complexité, il est identique au minimax. On peut par ailleurs optimiser le nombre d'opérations en se donnant un intervalle $[\alpha, \beta]$ dans lequel on restreint la recherche du score (on parle d'élagage): les scores calculés au fur et à mesure du parcours servent de minorant et de majorant pour les scores futurs. Si un nœud min a un score inférieur au max du nœud supérieur, il est inutile de poursuivre l'exploration (on parle de coupure α). De façon symétrique, si un nœud max a une valeur α supérieure à celle du min du nœud supérieur, on ne poursuit pas l'exploration (on parle de coupure β). Le gain en termes de nombre d'opérations peut être très important; il le sera d'autant plus que les meilleurs coups sont testés en premier. Lorsque minimax trouve un coup en n nœuds, $\alpha - \beta$ le trouve en $1 + 2\sqrt{n}$ nœuds, sous réserve d'un ordre convenable des nœuds. Un ordonanncement dans les coups potentiels peut alors

permettre un gain supplémentaire.

Nous disposons donc de deux techniques pour optimiser le parcours de l'arbre: une optimisation structurelle dans la forme de l'arbre, son parcours ou dans la classification des nœuds visités, ou bien une optimisation dans la qualité de la fonction d'évaluation.

Nous avons implémenté directement un algorithme négamax avec élagage $\alpha - \beta$, sans passer par un algorithme minimax, et avons ensuite cherché à optimiser la fonction d'évaluation.

3.3 La fonction d'évaluation d'une position.

La première fonction d'évaluation est très simple: elle affecte comme poids à chaque position du vecteur état d'un joueur, le nombre de positions gagnantes auxquelles il participe. Cela revient donc à effectuer un produit scalaire entre le vecteur et le tableau suivant:

3	4	5	7	5	4	3
4	6	8	10	8	6	4
5	8	11	13	11	8	5
5	8	11	13	11	8	5
4	6	8	10	8	6	4
3	4	5	7	5	4	3

Les positions centrales de la grille sont les plus favorisées, ce qui est conforme à l'intuition. Sous profondeur 1, l'algorithme $\alpha - \beta$ va donc jouer plutôt au milieu et favoriser la hauteur. La fonction donne des résultats acceptables et gagne 90% du temps contre l'algorithme qui joue au hasard, mais elle est pratiquement aveugle aux menaces immédiates et a tendance à s'acharner sur le milieu de la grille alors même que deux alignements adverses de trois pions sont présents sur les côtés.

Affecter un poids à chaque pion ne suffit donc pas pour jouer convenablement. Il faut également anticiper les alignements de 4 pions qui provoquent la fin de partie immédiate, les alignements de 3 pions possédant une case libre contigüe (ce sont des menaces auxquelles le joueur doit répondre immédiatement) et les alignements de deux points possédant deux cases libres contigües. Tous ces tests sont effectués au sein de la fonction d'évaluation, par des opérations binaires sur l'entier qui représente l'état du joueur.

À l'aide de ces améliorations, le jeu de l'IA devient meilleur, mais nous observons toujours des comportements aberrants, probablement dus aux mauvais réglages dans les poids affectés aux alignements.

4 Conclusion.

Le jeu tourne, fonctionne parfaitement entre deux joueurs humains et donne des résultats acceptables lorsqu'il met en jeu une intelligence artificielle. Nous aurions souhaité mettre en place d'autres types d'optimisations découverts dans la littérature: mise en place d'une bibliothèque d'ouverture, implémentation de tables de transposition qui permettent de détecter des positions déjà rencontrées, etc.

La bibliothèque openGL est essentiellement conçue pour de la 3D. On peut facilement améliorer l'aspect graphique du jeu et, pourquoi pas, utiliser des commandes de spécularité, de plaçage de textures, etc.

Enfin, nous aurions souhaité implémenter l'algorithme à stratégie gagnante de Victor ALLIS, mais le temps nous a manqué.

Références.

- Tristan CAZENAVE, « des optimisations de l'alpha-béta », *colloque de Berder*, 2000.

- Victor ALLIS, « A Knowledge-based Approach of Connect-Four. The Game is Solved: White Wins », *Vrije Universiteit Amsterdam*, 1988.

- Jonathan SCHAEFFER, « heuristic search », `http://webdocs.cs.ualberta.ca/~jonathan/PREVIOUS/Courses/657/`

- John TROMP, « The Fhourstones Benchmark. », `http://homepages.cwi.nl/~tromp/c4/fhour.html`

Annexe A: exécuter le programme.

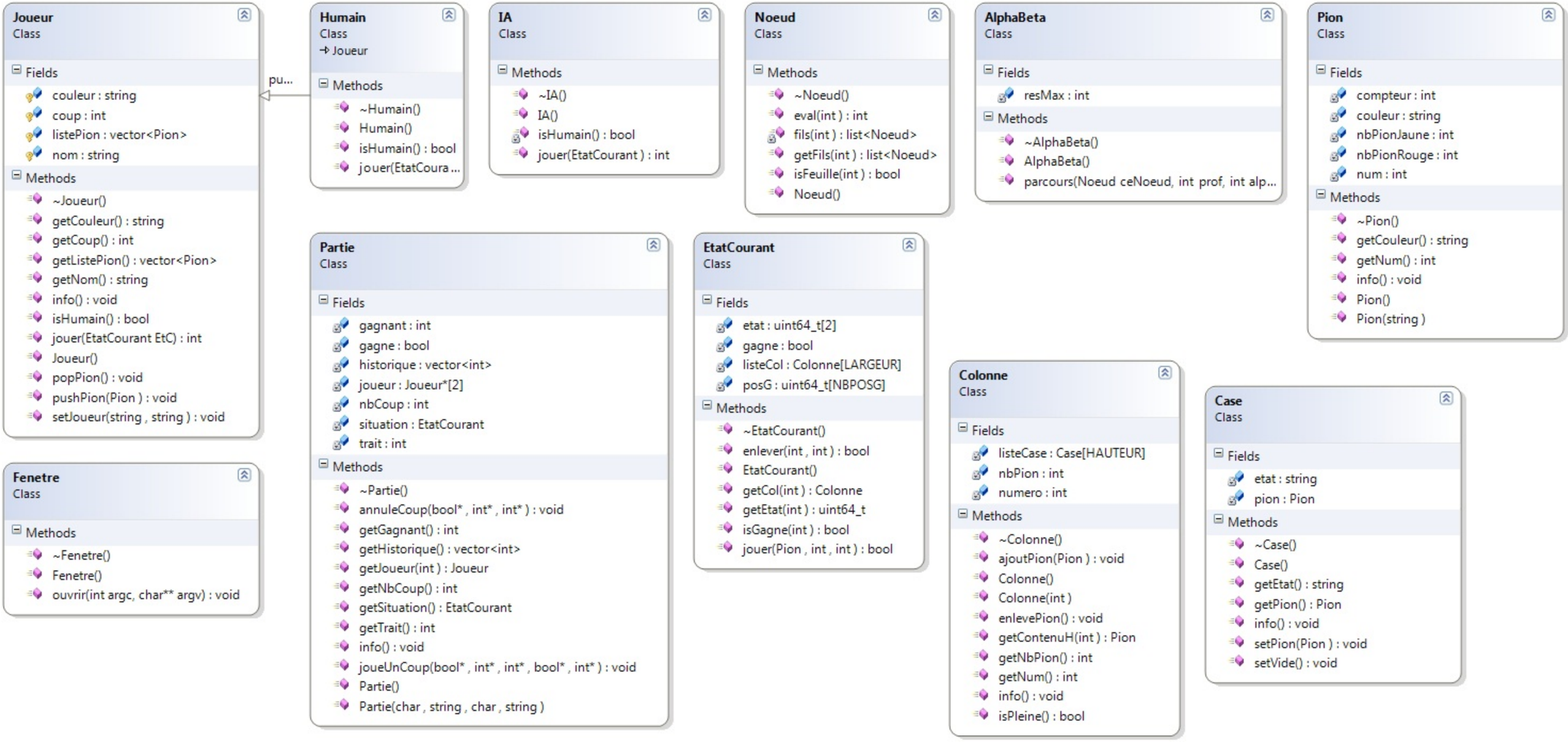
Le projet est disponible sur `https://github.com/cpetit`. Un dossier documentation comprend le rapport, des copies d'écran, ainsi qu'un fichier compressé avec l'exécutable et la librairie glut.dll. En principe l'exécutable fonctionne avec cette seule librairie (il a été testé sur plusieurs machines).

Le projet sera également déposé en intranet via xchangefile en respectant la même architecture.

Les dll. nécessaires à la compilation (i.e. les bibliothèques openGL et GLUT) sont joints au projet.

Enfin, en cas de problème, nous sommes disponibles pour effectuer une démonstration.

Annexe B: diagramme des classes.


Joueur
Class
Fields
couleur : string
coup : int
listePion : vector<Pion>
nom : string
Methods
~Joueur()
getCouleur() : string
getCoup() : int
getListePion() : vector<Pion>
getNom() : string
info() : void
isHumain() : bool
jouer(EtatCourant EtC) : int
Joueur()
popPion() : void
pushPion(Pion) : void
setJoueur(string , string) : void
pu...
Humain
Class
Joueur
Methods
~Humain()
Humain()
isHumain() : bool
jouer(EtatCoura ...
IA
Class
Methods
~IA()
IA()
isHumain() : bool
jouer(EtatCourant) : int
Noeud
Class
Methods
~Noeud()
eval(int) : int
fils(int) : list<Noeud>
getFils(int) : list<Noeud>
isFeuille(int) : bool
Noeud()
AlphaBeta
Class
Fields
resMax : int
Methods
~AlphaBeta()
AlphaBeta()
parcours(Noeud ceNoeud, int prof, int alp...
Pion
Class
Fields
compteur : int
couleur : string
nbPionJaune : int
nbPionRouge : int
num : int
Methods
~Pion()
getCouleur() : string
getNum() : int
info() : void
Pion()
Pion(string)
Partie
Class
Fields
gagnant : int
gagne : bool
historique : vector<int>
joueur : Joueur*[2]
nbCoup : int
situation : EtatCourant
trait : int
Methods
~Partie()
annuleCoup(bool* , int* , int*) : void
getGagnant() : int
getHistorique() : vector<int>
getJoueur(int) : Joueur
getNbCoup() : int
getSituation() : EtatCourant
getTrait() : int
info() : void
joueUnCoup(bool* , int* , int* , bool* , int*) : void
Partie()
Partie(char , string , char , string)
EtatCourant
Class
Fields
etat : uint64_t[2]
gagne : bool
listeCol : Colonne[LARGEUR]
posG : uint64_t[NBPOSG]
Methods
~EtatCourant()
enlever(int , int) : bool
EtatCourant()
getCol(int) : Colonne
getEtat(int) : uint64_t
isGagne(int) : bool
jouer(Pion , int , int) : bool
Colonne
Class
Fields
listeCase : Case[HAUTEUR]
nbPion : int
numero : int
Methods
~Colonne()
ajoutPion(Pion) : void
Colonne()
Colonne(int)
enlevePion() : void
getContenuH(int) : Pion
getNbPion() : int
getNum() : int
info() : void
isPleine() : bool
Case
Class
Fields
etat : string
pion : Pion
Methods
~Case()
Case()
getEtat() : string
getPion() : Pion
info() : void
setPion(Pion) : void
setVide() : void
Fenetre
Class
Methods
~Fenetre()
Fenetre()
ouvrir(int argc, char** argv) : void