UNIVERSITÉ
DE MONTPELLIER

Le jeu de Slither

Université de Montpellier - Licence 3 Informatique - TER

- Félix GUYARD - Lucas PICASARRI-ARRIETA - Boris MARCELLIN -

Encadrant
Monsieur Stéphane BESSY

Université de Montpellier, 2019
Département informatique

Remerciements

Nous tenons à remercier notre encadrant, Monsieur Stephane Bessy pour sa gentillesse et l'aide précieuse qu'il nous a apporté tout au long de notre projet. Ses conseils et son écoute nous ont permis de mieux comprendre les différents concepts algorithmiques et difficultés que nous avons pu rencontrer au cours de notre travail.

Table des matières

Chapitre 1

Introduction

1.1 Les jeux Combinatoires

La théorie des jeux combinatoires est une théorie mathématique qui étudie les jeux à deux joueurs comportant un concept de position, et où les joueurs jouent à tour de rôle un coup d'une façon définie par les règles, dans le but d'atteindre une certaine condition de victoire. La théorie des jeux combinatoires a pour objet les jeux à information complète où le hasard n'intervient pas, comme les échecs, les dames ou le jeu de go.
Les jeux initialement considérés par la théorie des jeux combinatoires possèdent les propriétés suivantes :

-Il est joué par deux joueurs qui jouent alternativement.
-Le jeu consiste en un certain nombre, généralement fini, de positions, et une position particulière est appelée position de départ.
-Les règles précisent clairement les coups qu'un joueur peut réaliser à partir d'une position donnée. Les positions accessibles par un joueur sont appelées ses options.
-Les deux joueurs connaissent parfaitement l'état du jeu, c'est-à-dire que le jeu est à information complète.
-Il n'y a aucune intervention du hasard.
-La partie se termine lorsqu'un joueur ne peut plus jouer, et les règles assurent que toute partie se termine en un nombre fini de coups (ce qui est appelé la condition de terminaison).
-Dans la convention normale de jeu, le joueur qui ne peut plus jouer est le perdant.
Si un jeu vérifie ces propriétés, le théorème de Sprague-Grundy (1935-1939) nous assure l'existence d'une stratégie gagnante pour l'un des deux joueurs.

1.2 Le jeu de Slither

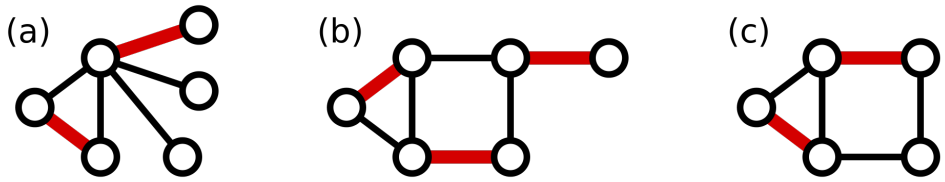
Le jeu de Slither est un jeu combinatoire ce qui signifie qu'il possède les proriétés précédentes. Le jeu se joue sur un plateau, modélisé par un graphe G, muni d'un pion posé sur un sommet de G. Tout au long de ce mémoire, Bob sera le premier joueur et Alice le second. Alice choisit la position de départ du pion, qui est un sommet de G. Les joueurs déplacent ensuite alternativement le pion selon les règles suivantes : à chaque tour de jeu le pion doit se déplacer sur un sommet adjacent à sa position actuelle et le pion ne peut pas se retrouver deux fois sur le même sommet. Autrement dit, le déplacement du pion depuis le commencement de la partie suit un chemin du graphe G. Le joueur ne pouvant plus déplacer le pion a perdu. Nous verrons que selon l'architecture du graphe, un des joueurs dispose d'une stratégie gagnante. L'objectif du TER est d'implémenter un espace de jeu offrant la possibilité d'affronter un autre joueur humain, mais aussi d'affronter l'ordinateur, celui-ci utilisant les stratégies gagnantes.

Chapitre 2

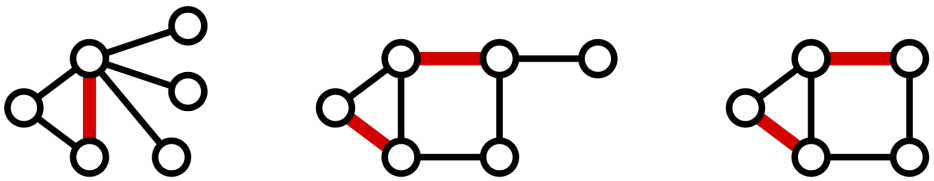
Définitions préalables

Soit M un couplage.
— Nous noterons le nombre d'arêtes du couplage M par $|M|$.
— Un sommet v est dit couvert par M s'il est incident à une arête de M.

Définition 1. *Un couplage maximum est un couplage contenant le plus grand nombre possible d'arêtes. Un graphe peut posséder plusieurs couplages maximum.*



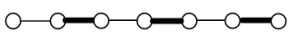
Définition 2. *Un couplage maximal est un couplage M du graphe tel que toute arête du graphe possède au moins une extrémité commune avec une arête de M. Ceci équivaut à dire dans l'ensemble des couplages du graphe, M est maximal au sens de l'inclusion, i.e. que pour toute arête a de A qui n'est pas dans M, $M \cup a$ n'est plus un couplage de G.*



Définition 3. *Un couplage est parfait s'il est incident à tous les sommets de V.*

Définition 4. *Un chemin dans le graphe est dit M-alternant si ses arêtes sont alternativement dans M et hors de M (donc une arête sur deux est dans M).*

Définition 5. *Un chemin M-augmentant est un chemin alternant dont les points de départ et d'arrivée sont deux sommets non couverts par M.*



Ce chemin est M-alternant.

Ce chemin est M-augmentant de 5 arêtes.

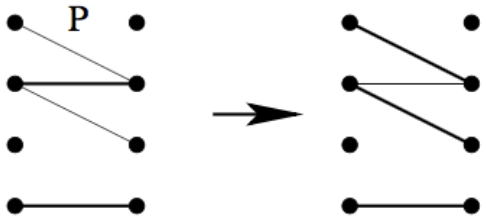
Définition 6. *Un graphe est dit biparti s'il existe une partition de son ensemble de sommets en deux sous-ensembles A et B telle que chaque arête ait une extrémité dans A et l'autre dans B .*

Théorème 1 (Berge 1957)**.** *Soit M un couplage dans un graphe G. M est maximum si et seulement si il n'existe pas de chemin M-augmentant.*

Démonstration 1. *(=>) On raisonne par contraposée : montrons alors que si le graphe possède un chemin M-augmentant, alors M n'est pas maximum.*
Soit P le chemin M-augmentant. Il suffit de considérer l'ensemble d'arêtes M' suivant :
$M' = (M \backslash P) \cup (P \backslash M)$.
Comme P est un chemin M-augmentant, aucune arête de P n'a une extrémité commune avec l'ensemble d'arêtes $(M \backslash P)$ (sinon, M ne serait pas un couplage). On en déduit que M' est un couplage. Comme P est un chemin M-augmentant, $P \backslash M$ possède une arête de plus que $P \cap M$. Donc $|M'| = |M| + 1$ et le nombre d'arêtes de M' est supérieur à celui de M. Ceci permet de conclure que M n'est pas maximum.

(<=) On raisonne de nouveau par contraposée : supposons que M ne soit pas maximum, et montrons qu'il existe un chemin M-augmentant.
Soit M' un couplage maximum.
Soit H le sous-graphe de G induit par les arêtes de $M \Delta M'$.
Les composantes connexes de H sont des cycles ou des chemins où les arêtes de M et M' alternent. Ainsi, pour chaque cycle, le nombre d'arêtes de M' est égal au nombre d'arêtes de M, sinon les arêtes de M et de M' n'alternent pas.
Puisque $|M'| > |M|$, il existe un chemin P pour le quel le nombre d'arêtes de M' est strictement supérieur au nombre d'arêtes de M. Alors P commence et termine par deux arêtes de M' et P est M-alternant, donc P est un chemin augmentant pour M.

Exemple 1. *Augmentation d'un couplage à partir d'un chemin augmentant P. Les arêtes en gras sont celles du couplage.*



Chapitre 3

La stratégie gagnante

3.1 Utilisation des couplages

Les couplages sont la clé des stratégies gagnantes du jeu de Slither. L'existence d'une stratégie gagnante pour l'un des joueurs dépend de la présence ou non d'un couplage parfait dans le graphe du jeu.
Si un graphe possède un couplage parfait, le second joueur possède une stratégie gagnante et ne peut pas perdre s'il la suit. Inversement, si le graphe ne possède pas de couplage parfait, le premier joueur possède une stratégie gagnante.
Dans tous les cas, pour adopter la meilleure stratégie, il convient au préalable de calculer un couplage maximum du graphe.

3.2 Stratégie avec un couplage parfait

3.2.1 Description

Le second joueur connaît un couplage parfait pour le graphe donné. Peu importe le choix du joueur adverse, il choisira un sommet en suivant le couplage parfait qu'il connaît, c'est-à-dire le sommet associé au sommet courant dans le couplage. Parce que le couplage est parfait, peu importe le choix du joueur adverse, il existera toujours un sommet associé dans le couplage.

3.2.2 Pourquoi cela fonctionne ?

Le premier joueur ne pourra jamais choisir un sommet dont tous les voisins ont déjà été vu. Si c'était le cas, le sommet en question ne serait pas couvert par le couplage, et donc le couplage en question ne serait pas parfait, ce qui contredit l'hypothèse de départ.

3.3 Stratégie sans couplage parfait

3.3.1 Description

Le premier joueur connaît un couplage maximum dans le graphe et commence en choisissant un sommet qui n'est pas couvert par le couplage.
Par la suite, il appliquera la stratégie expliquée précédemment : après que son adversaire ait choisi un sommet, il choisira le sommet associé dans le couplage qu'il a calculé.

3.3.2 Pourquoi cela fonctionne ?

Le second joueur ne peut pas choisir un sommet non couvert par le couplage maximum que le premier joueur a choisi. En effet, si c'était le cas, il suffit de retracer la liste d'arêtes jouées pour obtenir un chemin augmentant. Ce qui contredirait le fait que le couplage soit maximum d'après le théorème de Berge.

Chapitre 4

Algorithmes

Nous présenterons ici les algorithmes qui nous ont demandé le plus d'investissement. Nous nous contenterons donc de décrire les algorithmes permettant de créer les différents types de graphes, et nous présenterons en détail l'algorithme de recherche d'un couplage maximum et l'algorithme Force-based.

4.1 Génération des graphes

Le premier problème algorithmique que nous avons rencontré était celui de la génération de graphes. Comment créer un graphe aléatoire, et plus précisément, comment créer un graphe pour que le premier ou le deuxième joueur dispose d'une stratégie gagnante ?

4.1.1 Graphe aléatoire quelconque

Il s'agit ici de décrire comment nous avons choisi de créer des graphes quelconques, sans savoir à l'avance qui de Alice ou de Bob disposera d'une stratégie gagnante.
On commence par disposer aléatoirement sur la fenêtre un nombre fixe de sommets. Ce nombre est fixe afin de ne pas faire des parties qui dureraient trop longtemps ou trop peu de temps.
On créé ensuite des arêtes entre deux sommets tirés aléatoirement jusqu'à obtenir un graphe connexe.

4.1.2 Graphe gagnant pour Alice

Il s'agit ici de créer un graphe sur lequel Alice, le deuxième joueur, dispose d'une stratégie gagnante. Pour cela, le graphe doit disposer d'un couplage parfait.
La première étape est la même que précédemment : disposer aléatoirement sur la fenêtre les sommets du graphe. Le nombre de sommets doit être paire afin de permettre la création d'un couplage parfait.
Par la suite, on va partitionner l'ensemble V des sommets en deux sous-ensembles A et B de même taille. Dès lors, il suffit de créer $\frac{|V|}{2}$ arêtes en associant à chaque sommet de A un sommet de B différent pour créer un couplage parfait.
On créé ensuite comme précédemment des arêtes supplémentaires aléatoirement jusqu'à obtenir un graphe connexe.

4.1.3 Graphe gagnant pour Bob

Il s'agit cette fois de créer un graphe sur lequel Bob, le premier joueur, dispose d'une stratégie gagnante. Pour cela, le graphe ne doit pas disposer d'un couplage parfait.
On dispose comme précédemment les sommets sur la fenêtre. Ensuite on choisit un sous-ensemble A de l'ensemble des sommets V tel que $|A| = \lfloor \frac{n-1}{2} \rfloor$ où $n = |V|$. On créé ensuite des arêtes jusqu'à obtenir un graphe connexe mais de sorte que au moins une des deux extrémités de chaque arête soit dans A. Ainsi, puisque $|A| < \frac{n}{2}$ on ne disposera pas de couplage parfait.

4.2 Recherche d'un couplage maximum dans un graphe biparti

Lors de l'application de la stratégie gagnante, la difficulté repose sur la recherche d'un couplage maximum. Celle-ci se fait par la recherche de chemins augmentants. On ne considère ici que des graphes bipartis, l'algorithme s'étendant à l'ensemble des graphes étant sensiblement plus difficile à implémenter.
Dans ce qui suit, A et B désignent les deux parties du graphe biparti G, telles que $|A| \leq |B|$.

Algorithm 1 CouplageMax

Données: Graphe Biparti G

```
1: Couplage M ← ∅ ;
2: pour tous a_0 ∈ A non couvert par M faire
3:    T ← arbre(a_0)
4:    tant que Voisinage(A ∩ T) ≠ (B ∩ T) faire
5:       Choisir b ∈ Voisinage(A ∩ T)\(B ∩ T)
6:       Notons a' son voisin dans (A ∩ T)
7:       T ← T ∪ {b, a'b}
8:       si b est couvert par M alors
9:          Notons a son voisin dans M
10:         T ← T ∪ {a, ba}
11:      sinon
12:         // On a un chemin augmentant P de a_0 vers b dans l'arbre T
13:         M ← M XOR P ;
14:         Terminer tant que ;
15:      fin si
16:   fin tant que
17: fin pour
18: renvoyer M ;
```

4.2.1 Analyse Algorithmique

Terminaison

La boucle pour termine : un point qui est couvert par le couplage le restera (par définition du chemin augmentant).
Il reste à montrer que la boucle "Tant que" termine :
Notons déjà que l'on a toujours :

$$(B \cap T) \subseteq Voisinage(A \cap T) \subseteq B$$

car b est toujours choisi dans $Voisinage(A \cap T)$.
À une étape quelconque de la boucle, si b n'est pas couvert par M, alors la boucle "Tant que" termine directement.
Sinon, on ajoute à l'arbre T le sommet b, qui n'y était pas déjà. Donc la taille de $(B \cap T)$ augmente strictement. La taille de $Voisinage(A \cap T)$ étant majorée par la taille de B, la boucle Tant que terminera en au plus $|B|$ étapes.

Validité

On admet la validité de l'algorithme car la preuve de celle-ci est relativement complexe.

Complexité

La boucle pour s'effectue $|A|$ fois.
La boucle "Tant que" s'effectue au plus $|B|$ fois.

Choisir $b \in Voisinage(A \cap T) \backslash (B \cap T)$ et vérifier $Voisinage(A \cap T) \neq (B \cap T)$ s'effectue en au plus m opérations.
Finalement, la complexité de l'algorithme est $\mathcal{O}(|A|.|B|.m)$.

4.3 Algorithme Force-based

Cet algorithme n'a pas de lien avec le calcul des stratégies gagnantes. Pour un graphe donné, il vise à le rendre plus lisible en déplaçant les sommets pour réduire le nombre de croisements d'arêtes.
Il s'agit en fait d'assimiler les sommets du graphe à des particules de même charge (qui se repousseront donc) et les arêtes à des ressorts mécaniques.
En calculant la somme des forces appliquées à chaque sommet, on peut déplacer les sommets de manière à se rapprocher d'une situation stable.

Algorithm 2 Force-based

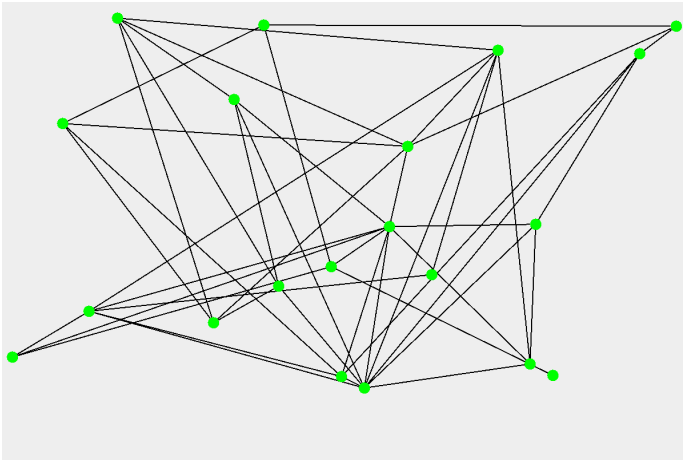
Données: G : Graphe, k (charge des particules), r (raideur des ressorts), n, m : entier, l_0 (longueur à vide des ressorts) : float.

Variable : dep : float[n][2] ;
//dep est le vecteur contenant les forces appliquées à chaque sommet
$dep \leftarrow 0$
Faire
$G.deplacer(dep)$ //Consiste à déplacer chaque sommet de G selon la composante de dep associée // sans qu'un sommet sorte du plateau de jeu ni ne superpose un autre sommet.
$dep \leftarrow 0$
pour i de 1 à n **faire**
  pour j de 1 à n **faire**
    si $dist > 0$ **alors**
      //Deux sommets se repoussent, comme deux particules de même charge
      $dist \leftarrow distance(points[i][j])$
      $dep[i][0] \leftarrow dep[i][0] + \frac{points[i].getX() - points[j].getX()}{dist} \times \frac{k}{dist^2}$
      $dep[i][1] \leftarrow dep[i][1] + \frac{points[i].getY() - points[j].getY()}{dist} \times \frac{k}{dist^2}$
    fin si
  fin pour
fin pour
pour i de 1 à m **faire**
  //Chaque arête agit comme un ressort entre les particules
  $A, B \leftarrow arêtes[i].getSommets()$ //A et B sont des sommets
  $i1, i2$ les indices de A et B dans les sommets de G
  si $dist > 0$ **alors**
    $dep[i1][0] \leftarrow dep[i1][0] + \frac{A.getX() - B.getX()}{dist} \times r(dist - l_0)$
    $dep[i1][1] \leftarrow dep[i1][1] + \frac{A[i].getY() - B[j].getY()}{dist} \times r(dist - l_0)$
    $dep[i2][0] \leftarrow dep[i2][0] + \frac{B.getX() - A.getX()}{dist} \times r(dist - l_0)$
    $dep[i2][1] \leftarrow dep[i2][1] + \frac{B.getY() - A.getY()}{dist} \times r(dist - l_0)$
  fin si
fin pour
Tant que dep contient une norme supérieur à normeMin

4.3.1 Application sur un exemple

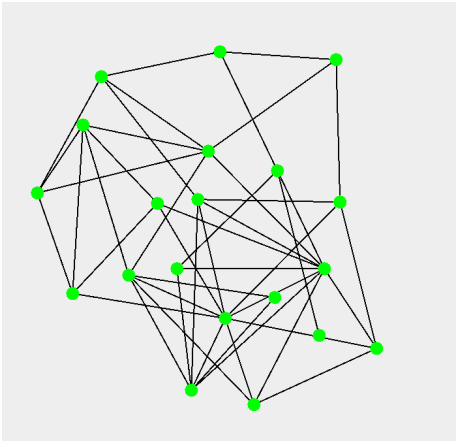
Appliquons l'algorithme précédent sur un exemple aléatoire.

Avant l'application du Force-based



La position de chaque sommet est générée aléatoirement, on obtient un graphe avec de nombreux croisements entre les arêtes et où les sommets voisins peuvent être très espacés.

Après l'application du Force-based



On obtient un graphe où le nombre de croisements est réduit, et où la distance entre deux sommets voisins est très proche de la distance moyenne. Le résultat est plutôt satisfaisant sur cet exemple.

Chapitre 5

Développement du jeu

5.1 Choix du langage Java

Le choix du langage étant libre, nous avons décidé d'utiliser Java en majeure partie pour son intégration de bibliothèques graphiques. Swing et Awt sont celles que nous avons utilisées.

5.2 Organisation et répartition des tâches

Afin de travailler de manière coordonnée nous avons mis en place plusieurs moyens de communication et outils de développement de projet.

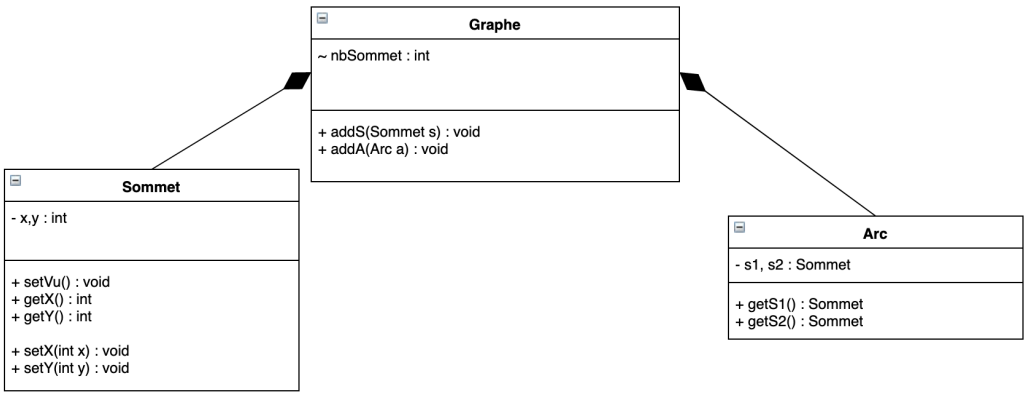
D'abord, nous avons mis en place un serveur de discussion sur la plate-forme Discord. Cela nous a permis de programmer le jeu à distance et, ainsi, discuter directement de nos problèmes de compréhension ou d'implémentation, et aussi, de prendre des décisions collectives facilement.

Ensuite, nous avons créé un serveur GitLab hébergé chez l'un d'entre nous afin de regrouper notre travail de manière organisée et structurée dans le temps.
Enfin, nous avons aussi créé un diagramme de Gantt (voir annexe) et basé notre organisation et notre répartition des tâches selon les méthodes agiles. Nous faisions le point ensemble sur l'avancée de chacun une fois par semaine, en plus des réunions effectuées avec l'encadrant.

5.3 Structure du programme

5.3.1 Structure de base

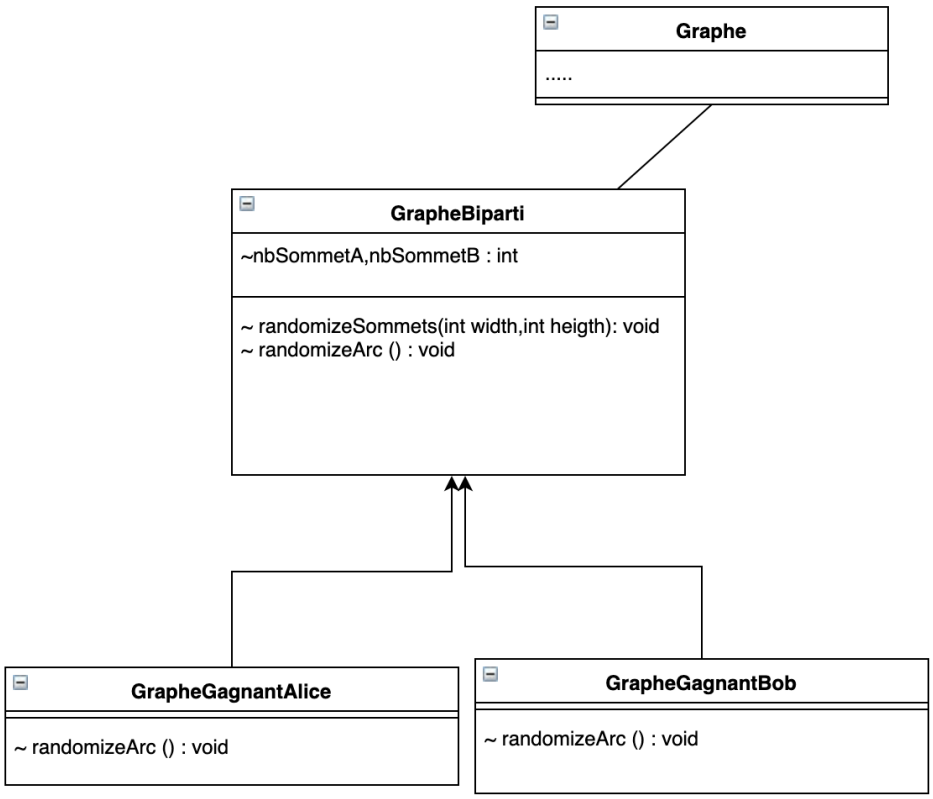
Une instance du jeu de Slither a besoin d'un graphe aléatoire en guise de "plateau de jeu". La première étape qui en découle est de créer une structure de graphe modulable qui servira de base pour toute partie jouée.
Un graphe est donc composé d'un ensemble de sommets et un ensemble d'arêtes.



On sait que le travail sur la stratégie gagnante ne s'effectuera que sur des graphes bipartis. On sait aussi que l'on va générer des graphes bipartis qui mettrons en évidence les deux stratégie gagnantes de Bob et de Alice. On intègre donc ces objets à notre jeu.

Dans un premier temps il faut connaître le sommet courant c'est-à-dire celui qui a été choisi par l'un des deux joueurs. Par conséquent, on veut aussi savoir quels sont les sommets encore accessibles depuis le sommet courant, pour déterminer si la partie est terminée ou non.

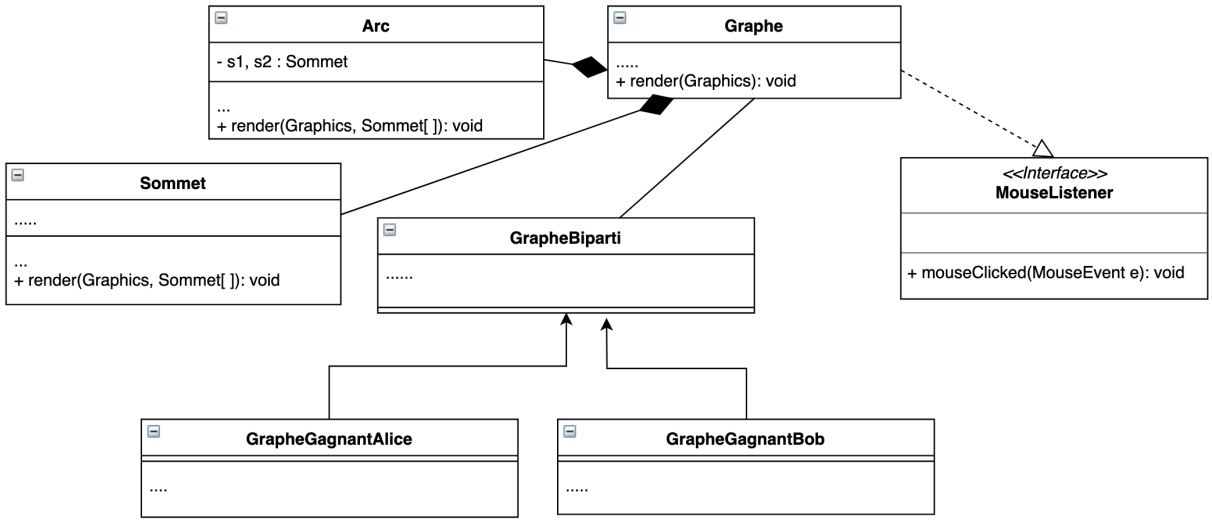
On veut dans un second temps générer des graphes aléatoires. Pour cela il faut générer des sommets et arêtes aléatoires en prenant soin de garder le graphe connexe. Il faut aussi prendre en compte que la génération aléatoire n'est pas la même si on choisit un graphe avec l'une des deux stratégies gagnantes.



Pour finir, il nous a été nécessaire d'introduire de nouvelles classes afin de pouvoir implémenter les algorithmes majeurs vus dans le chapitre précédent. Les notions d'arbres et de couplages, ont servi à l'implémentation de l'algorithme de couplage maximal, et sont utilisées pour implémenter la génération de graphes aléatoires pour Bob et Alice, ainsi qu'au développement de l'IA effectué plus tard. Il a aussi été nécessaire d'implémenter l'algorithme Force-based, ainsi que toutes les fonctions dont il dépend (voir annexe : UML).

5.3.2 Structure graphique

Le but est de faire afficher les graphes que l'on fabrique et de rendre leurs sommets "cliquables". Pour cela, on utilise les bibliothèques Swing et Awt. En résumé chaque objet possède une méthode de rendu, qui affichera ses différents composants graphiques (sommets, arêtes, couleurs, textes...). Chaque graphe implémente une interface de gestion de clique (MouseListener). À chaque clique effectué par un joueur, on vérifiera si un sommet a été cliqué. Si tel est le cas, on décide de l'action qui en découle.

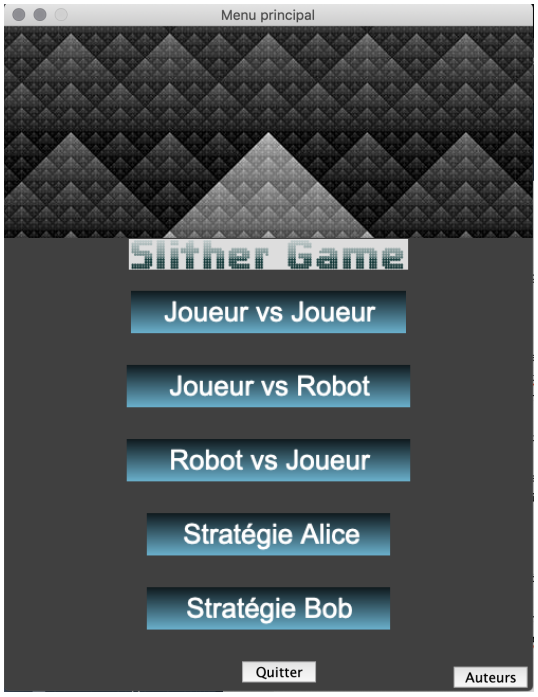


On verra plus tard que l'on utilise ces mêmes méthodes ainsi que différents attributs provenant de ces bibliothèques pour les actions du menu principal.

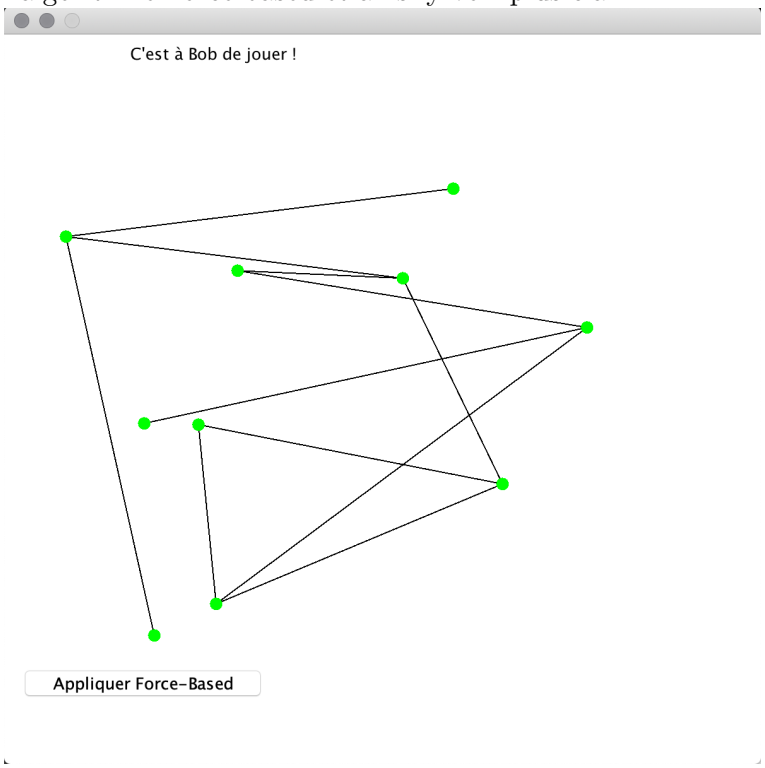
5.3.3 Structure centrale

Une partie (Game) est composée de deux joueurs qui peuvent être une IA ou un humain, et également d'un graphe. On spécialise une partie selon le type de graphe généré (partie gagnante pour Alice, partie standard, partie contre une IA). Toute partie peut être lancée à partir du menu principal (Launcher), où l'on peut choisir son type. Toutes les méthodes utilisées ont pour but de gérer l'affichage des composants graphiques et le déroulement de la partie (tours de jeux, mise à jour de l'affichage...).
(Voir UML en annexe pour toute la structure).

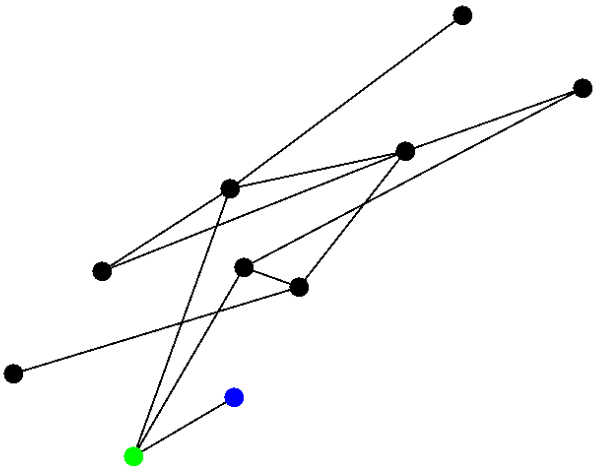
5.4 Démonstration

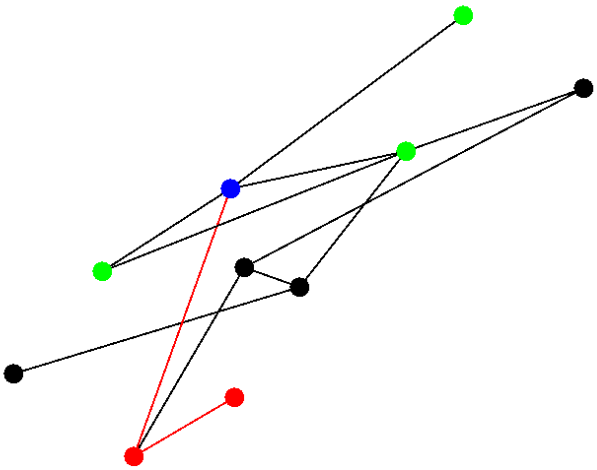


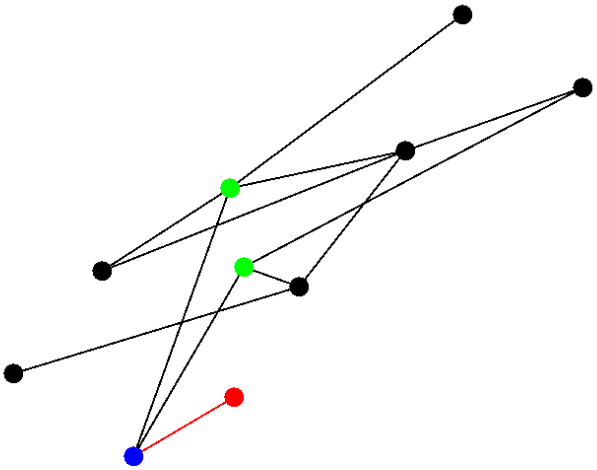
Lorsqu'on lance une partie contre un humain par exemple, on obtient une instance de Game standard avec un graphe aléatoire biparti. Si celui-ci possède trop de croisements, on peut appliquer l'algorithme Force-based et ainsi y voir plus clair...

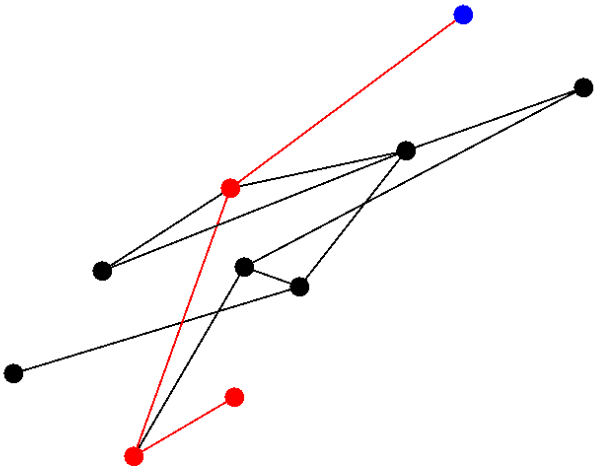


À son tour de jeu, un joueur clique sur un sommet accessible du graphe. La partie s'arrête lorsqu'aucun sommet n'est accessible. Un sommet vert indique un sommet accessible, le sommet bleu indique celui joué par le joueur précédent. Le chemin parcouru est tracé en rouge.

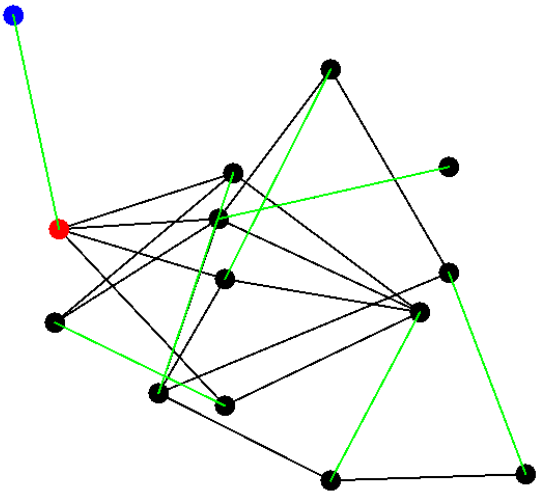








Dans le cas d'une partie avec une IA, le principe est le même sauf qu'un tour est joué par l'IA au lieu d'un humain. En fin de partie, on affiche le couplage maximum utilisé par l'IA en vert.



5.5 Problèmes rencontrés

Tout au long de ce projet, nous avons rencontré différents problèmes au niveau de l'implémentation en Java de nos algorithmes. Les problèmes rencontrés provenaient essentiellement de la programmation et non de l'algorithmique. L'utilisation des bibliothèques Swing et Awt qui nous étaient inconnues nous ont posé quelques difficultés en début de projet car elles utilisent des notions que nous ne connaissions pas en Java, comme la gestion d'évènements.

Chapitre 6

Conclusions

6.1 État final du projet

Notre application permet donc, en cette fin de projet, d'afficher un graphe aléatoire biparti, un graphe où le premier joueur dispose d'une stratégie gagnante, et un où le second joueur en dispose d'une. On peut également choisir de jouer contre un autre joueur, ou encore de jouer contre l'IA qui choisit la meilleure stratégie en fonction du graphe généré. On peut choisir de commencer ou de laisser l'IA commencer. Toutes ces options sont disponibles à partir du menu principal.
On peut également choisir d'utiliser le force-based que nous avons implémenté pour essayer de rendre le graphe plus lisible avec des points mieux répartis sur la fenêtre et moins de croisements entre les arêtes bien que le résultat ne soit pas toujours à la hauteur de nos attentes.

6.2 Améliorations possible

Bien que notre programme réponde au sujet donné par notre encadrant, il n'en reste pas moins améliorable sur plusieurs aspects que nous allons aborder ici :

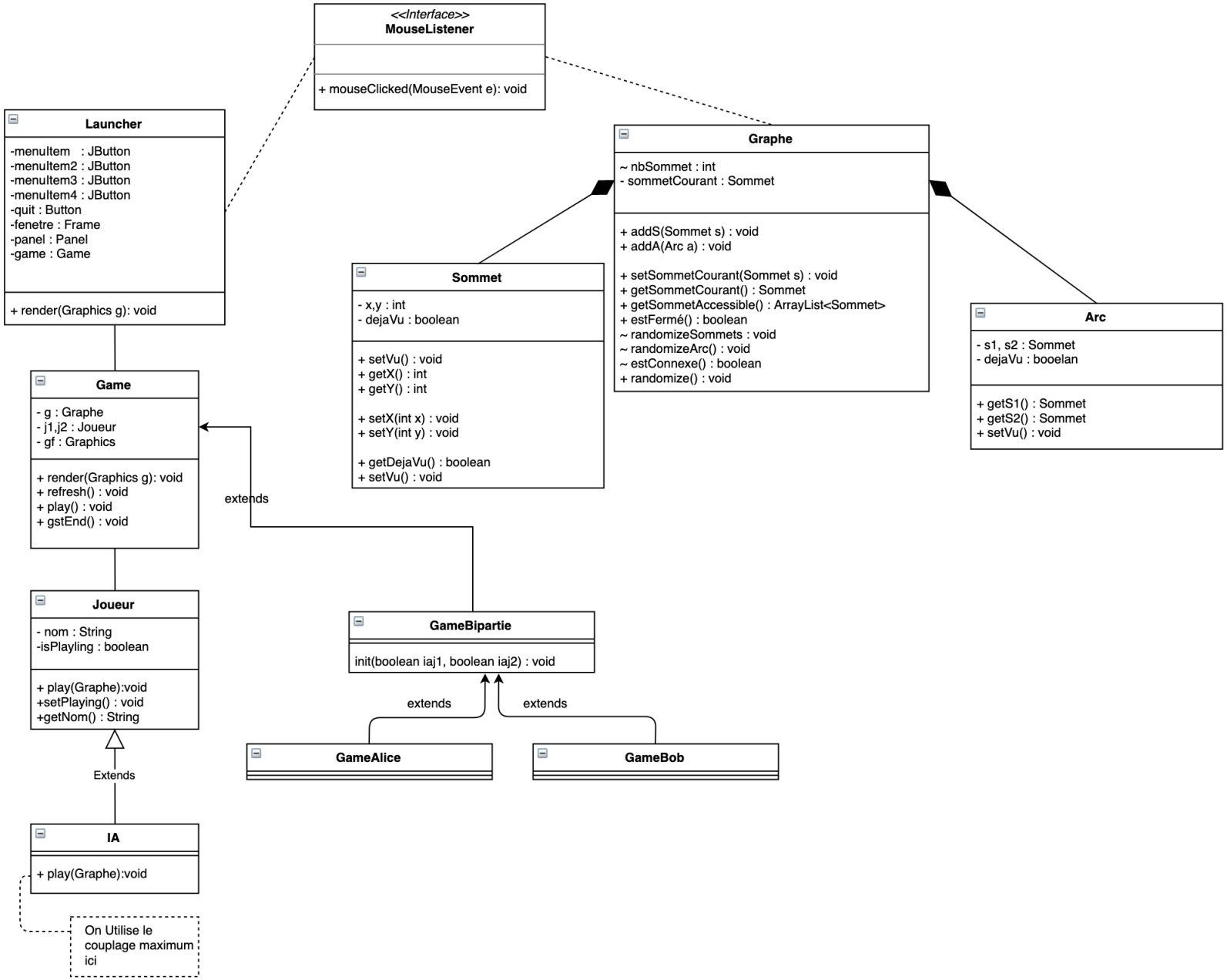
Tout d'abord la possibilité de choisir le nombre de sommets pour une partie, nous permettant ainsi de traiter des cas particuliers ou tout simplement d'agir sur la durée d'une partie. Actuellement, le programme utilise un nombre fixe de sommets qu'aura notre graphe de jeu, modifiable dans le code.
Nous pourrions également mettre un système permettant de prendre en charge des fichiers de configuration et permettre aux joueurs de jouer sur leurs propres plateaux de jeu.
Un ajout intéressant mais plutôt difficile à mettre en place aurait été d'écrire l'algorithme de calcul d'un couplage maximum sur tout type de graphe, et pas seulement sur les graphes bipartis. Cependant, la complexité d'écriture d'un tel algorithme nous a poussé à conserver notre algorithme actuel.
Pour finir, nous pourrions améliorer l'esthétique de l'application et ajouter quelques fonctionnalités au menu, comme des sous-menus ou encore, comme dit plus haut, un menu proposant la gestion du nombre de sommets.

6.3 Compétences acquises

Il convient de mentionner que ce projet nous a demandé beaucoup d'investissement et nous a permis de consolider nos connaissances en algorithmique des graphes et en programmation objet, et ainsi d'appliquer ce que nous avons étudié au sein des unités d'enseignement HLIN501 et HLIN505. Nous avons également pu mettre en place les concepts vu en conduite de projet (HLIN 408) et ainsi mener un projet ensemble et efficacement.

6.4 Remarques personnelles

Sachant que traiter ce sujet était notre premier choix pour cette année, nous avons été très intéressés par le projet et avons exprimé une motivation particulière pour cette UE. Enfin, étant un groupe de trois étudiants, et nous connaissant chacun les uns les autres depuis longtemps, nous avons travaillé dans de très bonnes conditions, et avec une très bonne entente et ambiance.


«Interface»
MouseListener
+ mouseClicked(MouseEvent e): void
Launcher
-menuItem : JButton
-menuItem2 : JButton
-menuItem3 : JButton
-menuItem4 : JButton
-quit : Button
-fenetre : Frame
-panel : Panel
-game : Game
+ render(Graphics g): void
Graphe
~ nbSommet : int
- sommetCourant : Sommet
+ addS(Sommet s) : void
+ addA(Arc a) : void
+ setSommetCourant(Sommet s) : void
+ getSommetCourant() : Sommet
+ getSommetAccessible() : ArrayList<Sommet>
+ estFermé() : boolean
~ randomizeSommets : void
~ randomizeArc() : void
~ estConnexe() : boolean
+ randomize() : void
Sommet
- x,y : int
- dejaVu : boolean
+ setVu() : void
+ getX() : int
+ getY() : int
+ setX(int x) : void
+ setY(int y) : void
+ getDejaVu() : boolean
+ setVu() : void
Arc
- s1, s2 : Sommet
- dejaVu : booelan
+ getS1() : Sommet
+ getS2() : Sommet
+ setVu() : void
Game
- g : Graphe
- j1,j2 : Joueur
- gf : Graphics
+ render(Graphics g): void
+ refresh() : void
+ play() : void
+ gstEnd() : void
extends
Joueur
- nom : String
-isPlayling : boolean
+ play(Graphe):void
+setPlaying() : void
+getNom() : String
Extends
GameBipartie
init(boolean iaj1, boolean iaj2) : void
extends
extends
GameAlice
GameBob
IA
+ play(Graphe):void
On Utilise le couplage maximum ici