Le jeu de Shannon
Projet de programmation 2

Auteurs

Martina Agüera Sanchez
Samuel Casas Dray
Julien Lahoz
Adrien Martinelli

Encadrant

Stéphane Bessy

12 mai 2024

Table des matières

1 Présentation du jeu de Shannon

Dans le cadre de notre projet de licence, nous avons conçu en équipe un programme dédié au jeu de Shannon. Notre objectif était d'explorer les différentes facettes algorithmiques du jeu en étudiant l'aspect théorique lié aux graphes et en implémentant les algorithmes liés, le tout assorti d'une interface graphique et de fonctionnalités visant à rendre l'expérience utilisateur plus plaisante.

Le jeu de commutation de Shannon est un jeu de connexion pour deux joueurs, inventé par le mathématicien et ingénieur électricien américain Claude Shannon en 1951 [1]. Le jeu oppose sur un graphe connexe deux joueurs nommés SHORT et CUT qui jouent à tour de rôle. Le but de CUT est de rendre le graphe non connexe en sectionnant des arêtes, celui de SHORT est de l'en empêcher en sécurisant des arêtes. SHORT gagne s'il sécurise un arbre couvrant du graphe. CUT commence toujours à jouer.

Voici un exemple simple :

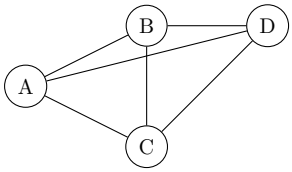


FIGURE 1 – Graphe initial.

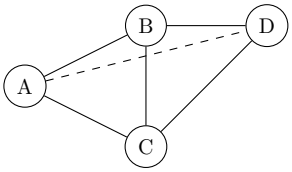


FIGURE 2 – CUT coupe (A, D).

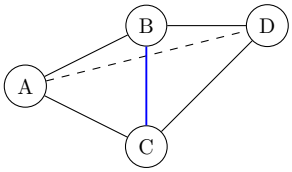


FIGURE 3 – SHORT sécurise (B, C).

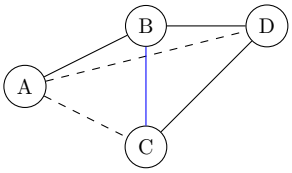


FIGURE 4 – CUT coupe (A, C).

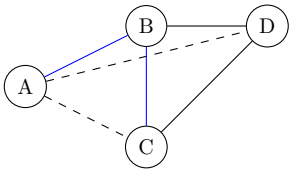


FIGURE 5 – SHORT sécurise (A, B) pour empêcher CUT de gagner.

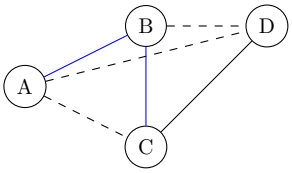


FIGURE 6 – CUT coupe (B, D).

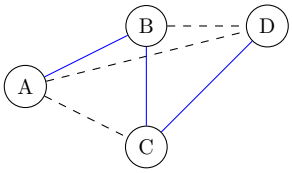


FIGURE 7 – SHORT sécurise (C, D).

À la dernière figure, nous constatons que SHORT a sécurisé un arbre couvrant du graphe. En effet, le chemin formé par les arêtes bleues est connexe et passe par tous les sommets. SHORT a donc remporté la partie.

Le jeu de Shannon est un jeu tour à tour, fini, à deux joueurs, à information parfaite, sans hasard et sans match nul, il vérifie donc les hypothèses du théorème de Zermelo [8], l'un des deux joueurs possède une stratégie gagnante qui sera détaillée dans la section 4 qui y est consacrée.

2 Développements Logiciel : Conception, Modélisation, Implémentation

2.1 Méthode et Organisation du développement

Dans un premier temps, toute l'équipe du projet s'est réunie avec le professeur, notre tuteur de projet, pour faire un premier point et choisir vers quelle direction notre groupe devait aller. Ensuite, toutes les deux semaines, le mardi, nous avions un nouveau rendez-vous pour discuter de l'avancement du projet, décider des nouvelles directives ainsi que pour poser nos questions si nous en avions.

Suite à la première réunion, nous nous sommes tous retrouvés pour décider de nos méthodes de travail. Nous avons commencé par énumérer les tâches à faire. Pour cela nous avons utilisé le site Jira qui est un outil de gestion de projet et de suivi des tâches.

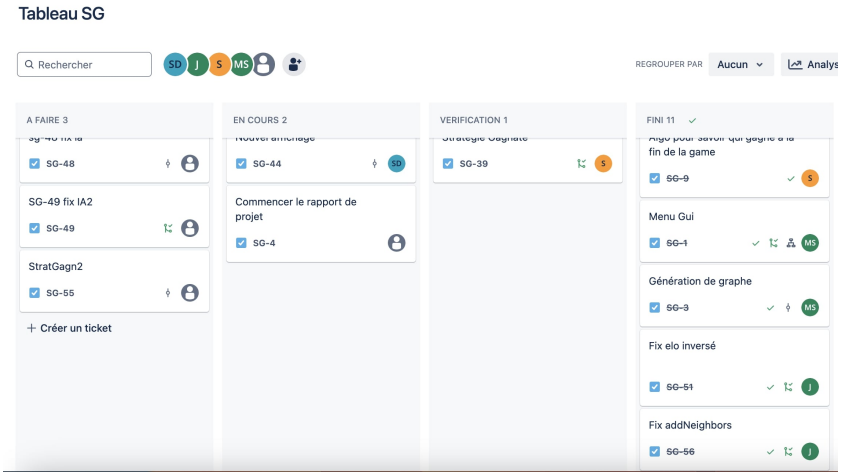


FIGURE 8 – Tâches de Jira.

Aussi, nous avons décidé d'utiliser GitHub (voir figure 9) pour partager entre nous le code

source du projet. Grâce à cela, nous avons pu développer plusieurs versions du projet et permettre à plusieurs personnes de travailler sur des tâches qui leur sont attribuées sans se perturber mutuellement.

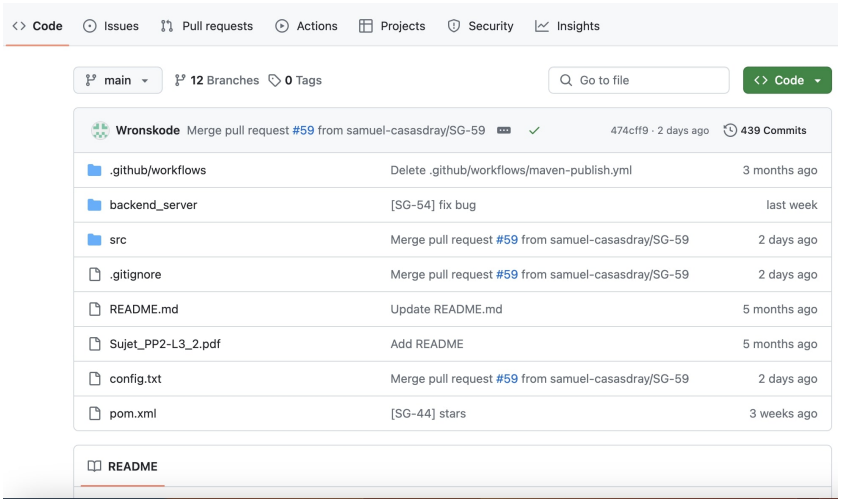


FIGURE 9 – Projet Github.

Tout au long du projet, le rapport était accessible, ce qui nous a permis d'avancer graduellement sur les tâches individuelles de chacun. Naturellement, chaque membre du groupe a progressé simultanément sur le rapport et le projet tout au long du développement.

2.2 Développements logiciel réalisés

Le code du programme est presque intégralement écrit en Java [1] et nous avons utilisé la bibliothèque JavaFX pour l'aspect graphique. Nous avons opté pour ce langage, car il offre une souplesse permettant de concevoir à la fois des algorithmes de traitement et de création de graphes, tout en intégrant une interface graphique. De plus, chaque membre du groupe se sentait à l'aise avec ce langage, ce qui nous donnait confiance dans notre capacité à réaliser correctement ce projet.

Le jeu est jouable en ligne et le serveur est codé en Rust. Pour le serveur, nous avons opté pour l'utilisation d'axum, un framework web performant et open source.

Nous proposons dans notre jeu de nombreuses modalités.

1. Jouer contre l'IA

   Nous proposons quatre niveaux de difficulté pour jouer contre l'IA (expliqués dans la section 3.3 concernant les différentes IA) :
   - Niveau facile
   - Niveau moyen
   - Niveau difficile
   - Stratégie gagnante

2. Jouer en ligne

   Dans cette modalité, nous pouvons jouer en ligne contre un autre joueur, en tant qu'utilisateur invité (sans compte) ou alors en tant qu'utilisateur connecté.

1. En version 21.

3. Jouer en local sur un même écran.
4. Voir deux IA jouer entre elles

   Cela n'a pas une grande utilité pour un utilisateur, mais nous voulions laisser la possibilité d'observer nos différentes IA et pouvoir les comparer.
5. Mode histoire

   Le mode histoire raconte l'histoire intergalactique de notre jeu et explique comment jouer en tant que SHORT ou CUT.

Sur chacune de ces modalités (sauf IA vs IA), le joueur peut choisir son camp (CUT ou SHORT) et le nombre de sommets du graphe (compris entre 5 et 40, et par défaut à 20 pour pouvoir garantir une bonne visibilité du graphe, le bon fonctionnement de l'IA et l'intérêt de la partie).

Nous avons également mis à disposition, des options de paramétrages qui offrent à l'utilisateur la possibilité de définir le volume de la musique et des effets sonores, de régler le nombre d'étoiles en mouvement en arrière-plan, et de choisir entre un graphe avec des planètes comme sommets ou des cercles. Dans le cas des cercles, ces derniers sont colorés avec un maximum de six couleurs, de sorte que deux sommets adjacents soient colorés différemment. Cet algorithme de coloration à six couleurs, appelé 6-coloration, sera expliqué dans la section 3.

Enfin, nous avons créé une page de statistiques qui indique le nombre de parties réalisées depuis l'implémentation de cette fonctionnalité, le nombre de parties remportées par CUT, le nombre de parties remportées par SHORT et le nombre de parties jouées en ligne.

2.3 Interface graphique

L'entièreté de l'interface graphique a été réalisée avec la bibliothèque JavaFX. Ce fut un défi pour nous, car nous ne l'avions jamais utilisée auparavant. L'interface de notre projet a donc évolué en même temps que nos compétences dans JavaFX. Nous avons d'abord codé une version qui nous permettait de visualiser simplement le plateau de jeu puis, alors que nous arrivions à la fin du projet, lorsque la partie algorithmique était presque finie, nous avons décidé de la faire évoluer.

L'interface graphique finale présente une thématique spatiale, ce qui rend le jeu plus attirant pour l'utilisateur. En effet, au lieu de présenter un jeu avec un graphe sur lequel il faut couper ou sécuriser des arêtes, nous proposons une galaxie dans laquelle les planètes sont reliées entre elles, le but du joueur CUT est donc de les séparer alors que SHORT veut créer un réseau de connexion galactique. Pour développer cet univers, nous avons remplacé les simples sommets par des planètes, choisi une police futuriste et nous avons rajouté des sons qui représentent le fait de couper ou sécuriser une arête.

Pour un résultat encore plus convaincant, nous avons conçu une ambiance sonore pour notre jeu ainsi qu'un fond stellaire en mouvement évoquant un voyage dans l'espace.

Le fond contient des étoiles en mouvement afin d'évoquer le voyage interstellaire. Pour faire l'animation, nous avons utilisé un peu de géométrie dans l'espace et le calcul final est optimisé pour ne pas consommer beaucoup de ressources. Nous avons toutefois laissé la possibilité à l'utilisateur d'ajuster le nombre d'étoiles grâce à un curseur.

La musique a été entièrement composée à la main, note après note, grâce à un logiciel de création musicale. Quant aux effets sonores, ils ont été créés en combinant des boucles trouvées sur Internet.

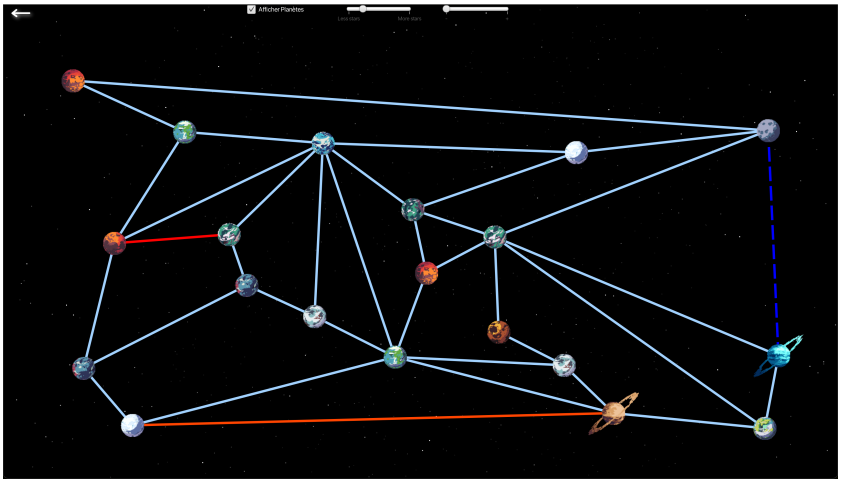


FIGURE 10 – Image d'une partie en cours entre deux IA.

2.4 Modélisation

Nous avons cinq packages : un relative au jeu, un à la création du graphe, un à l'interface graphique, un à l'intelligence artificielle et enfin un au serveur. Chaque type de classe est regroupé dans un dossier correspondant à sa catégorie.

1. Game : Nous avons une classe Game qui s'occupe de gérer les parties du début à la fin. Elle supervise les tours dans toutes les modalités possibles et détermine les conditions de fin de partie en identifiant le vainqueur. Pour réaliser cela, elle utilise les trois énumérations du dossier : Level, Turn et TypeJeu.
2. Graph : Cette classe gère la création et la manipulation de graphes planaires. Elle comprend des méthodes pour évaluer la connexité du graphe, calculer les arbres couvrants, et déterminer la stratégie gagnante du jeu de Shannon. Le dossier comprend aussi la classe Vertex qui représente un sommet dans un graphe. Chaque sommet est caractérisé par des coordonnées (x, y) et une couleur.
3. Gui : Ce dossier contient deux interfaces (HandleClick et JoinCreateField), quatre classes (Etoile, Gui, GuiScene et UtilsGui) et une énumération (ButtonClickType).

   Nos deux interfaces sont fonctionnelles, c'est-à-dire qu'elles ne contiennent, qu'une méthode abstraite. HandleClick sert à gérer les clics et JoinCreateField à gérer les champs de texte Join et Create du mode de jeu en ligne.

   La classe Etoile s'occupe du fond stellaire dynamique. Elle contient des méthodes de création d'étoiles avec des coordonnées aléatoires et avec leur couleur qui change selon leur position (ce qui crée un effet dynamique). Ensuite, la classe Gui est la plus importante de notre interface graphique, en effet, il s'agit d'un fichier assez volumineux qui contient les principales fonctionnalités de GUI. Elle s'occupe d'initialiser l'interface JavaFX, de gérer les clics, de créer des parties en faisant appel aux autres classes, d'afficher le graphique créé par Graph, de redimensionner l'affichage en cas de changement de taille de la fenêtre, de rejoindre des parties en ligne et de gérer les ressources telles que la police, les images et les sons. La classe suivante est GuiScene, on pourrait dire qu'elle représente l'épine dorsale de l'interface graphique. En effet, elle crée toutes les scènes (accueil, choix de niveaux, choix de nombre de sommets...) avec tout ce qu'elles contiennent (texte, boutons,

sliders...) et gère les liens et transitions possibles entre ces scènes. De plus, cette classe s'occupe de l'arrière-plan et de ses threads. Pour finir avec les classes, nous avons UtilsGui qui représente une collection de méthodes utilitaire. Comme la définition des différentes polices, des méthodes pour créer différents styles de texte et de bouton, des gestions d'évènements comme le survol d'un bouton, l'appui de la touche Entrée...

Enfin, l'énumération contient toutes les transitions de scène possibles. Par exemple pour définir ce qui se passe lorsqu'on clique la flèche de retour pendant le jeu.

4. IA : Ce dossier contient une classe abstraite InterfaceIA qui offre une infrastructure générique pour la création de nos différentes stratégies d'IA. Ses attributs, tels que game, plays et graph, fournissent un accès à l'état du jeu et à sa structure sous-jacente. De plus, la variable depth permet de contrôler la profondeur de recherche de l'IA, ce qui permet ensuite de régler le niveau. Elle contient aussi les méthodes abstraites pour faire jouer SHORT ou CUT. Nous avons trois classes qui étendent cette classe abstraite, il s'agit de BasicIA, Minimax et de WinnerStrat. Elles mettent en place quatre niveaux de difficulté d'IA différents expliqués dans la section 3.3 concernant les différentes IA.
5. Server : les classes contenues dans ce dossier seront expliquées dans la partie 6.1 expliquant le mode en ligne.

En dehors de ces dossiers, on retrouve une interface fonctionnelle CallBack qui contient une méthode call() qui ne prend aucun argument et ne renvoie rien. Elle est conçue pour être utilisée pour passer en paramètre de fonction, une fonction et donc exécuter une action donnée lorsqu'elle est invoquée. Elle est utilisée dans Server. On retrouve aussi la classe Main, la classe principale de notre application. Elle importe Gui et Application (nécessaire pour que JavaFX lance une application) et ensuite dans la méthode main, elle lance l'application et appelle System.exit(0) pour quitter proprement l'application lorsque l'interface graphique est fermée.

Le diagramme de classes est disponible en annexe, figure 15.

2.5 Statistiques

Notre projet comporte 22 classes Java (en prenant en compte les interfaces, les énumérations et les interfaces fonctionnelles), ce qui fait d'après le plugin « statistics » d'IntelliJ, un total de 4 118 lignes de code Java, et 3 581 lignes de code source (en enlevant les lignes vides ainsi que les commentaires). À cela, nous pouvons ajouter 577 lignes de code Rust.

3 Algorithmes et Structures de Données

3.1 Le graphe

Pour mettre en œuvre le jeu de Shannon, nous avons dû concevoir une classe graphe. Après plusieurs tentatives et corrections, nous avons finalement opté pour l'utilisation d'une table de hachage servant de liste d'adjacence. Celle-ci utilise des objets *Vertex* dont la classe est décrite dans la section 2 comme clés et associe à chacun une liste de sommets représentant ses voisins. Le graphe n'est ni pondéré ni orienté. Pour simplifier l'implémentation de certains algorithmes, nous avons également un HashSet contenant les arêtes du graphe.

Nous avons deux constructeurs :

— Nous pouvons construire le graphe avec la liste des arêtes, il faut cependant faire attention à ajouter les sommets isolés *a posteriori*.

— Il est également possible de construire le graphe avec une liste de sommets et une liste d'adjacence.

3.2 Les principaux algorithmes

Plusieurs algorithmes ont été implémentés dans ce projet :
— estConnexe, pour savoir si un graphe donné est connexe.
— Un algorithme de création de graphes planaires.
— Un algorithme de 6-coloration de graphe planaire basé sur le fait que tout graphe planaire admet au moins un sommet de degré au plus 5 (la preuve de ce fait est en annexe 8.2).
— Les algorithmes qui permettent de détecter si SHORT ou CUT a gagné, ces deux algorithmes prennent en entrée une « Game » et renvoient un booléen selon si CUT ou SHORT a gagné.
— Un algorithme permettant d'obtenir un arbre couvrant.

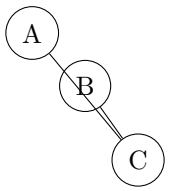
3.2.1 estConnexe

L'algorithme permettant de savoir si un graphe est connexe est très simple, il s'agit d'un parcours en profondeur itératif, dans lequel les sommets visités sont marqués. À la fin nous vérifions si le nombre de sommets marqués est égal au nombre de sommets du graphe.

3.2.2 Génération de graphes planaires

Cet algorithme prend en entrée un nombre de sommets n, une distance minimale d (en termes de pixels) entre chaque sommet du graphe et un rayon r, un degré minimal m, un degré maximal M ainsi qu'une probabilité p.

La distance d permet de créer des graphes visuellement agréables, et le rayon permet d'éviter que certaines arêtes intersectent un nœud du graphe comme illustré ci-dessous :



Le graphe renvoyé est un graphe planaire (connexe) dont tous les sommets sont de degré au moins m et au plus M, et tel que pour tout couple de sommets s_1, s_2, la distance euclidienne entre s_1 et s_2 soit d'au moins d. La probabilité p est utilisée pour ajuster le nombre d'arêtes du graphe, afin d'atteindre un équilibre où environ 50 % des graphes favorisent la stratégie de cut et 50 % favorisent celle de short.

La première phase de l'algorithme consiste à placer les sommets aléatoirement en respectant la distance minimale entre chaque sommet. Ce n'est pas toujours possible, car cela dépend de la taille de l'écran, du nombre de sommets à placer, et de la distance d. Pour éviter tout problème, cette boucle est un « tant que » qui termine après un nombre conséquent d'itérations.

La deuxième partie de l'algorithme consiste à parcourir tous les sommets, et de relier chaque sommet au maximum de ses voisins (on vérifie notamment que les arêtes ne s'intersectent pas, et on vérifie aussi l'intersection entre les arêtes à placer et les cercles représentant les sommets).

Nous obtenons à partir d'ici un graphe planaire ayant le maximum d'arêtes possible. Nous faisons donc une dernière itération sur les arêtes pour retirer certaines arêtes selon la probabilité

p. La raison de placer cette boucle après le placement des arêtes est que l'on peut vérifier le degré des sommets auxquels nous retirons des arêtes pour maintenir un degré minimal de m.

Si le graphe obtenu n'est pas connexe, ou que son degré minimum est plus petit que m, ou que son degré maximum est plus grand que M, nous recommençons l'algorithme entièrement dans un « tant que » qui vérifie à chaque itération si le temps depuis la première itération est supérieur à 2 secondes. Si c'est le cas, l'algorithme n'a pas réussi à trouver un tel graphe et un message d'erreur est renvoyé à l'utilisateur lui suggérant de réduire le nombre de sommets.

3.2.3 6-coloration

L'algorithme proposé ici est récursif : si le graphe G a 6 sommets ou moins alors nous colorons chaque sommet avec une couleur différente, sinon nous trouvons un sommet v de degré au plus 5 (l'existence est garantie, car G est planaire), et nous faisons un appel récursif sur $G' = G \setminus \{v\}$, enfin nous donnons à v une couleur qui n'est pas prise par un de ses voisins.

Nous proposons ici une petite preuve par induction sur le nombre de sommets n de G.

Démonstration. Soit G un graphe planaire connexe.

Cas de base Si $n \leq 6$ nous attribuons une couleur différente à chaque sommet de G

Induction Soit v un sommet de G de degré au plus 5, soit $G' = G \setminus \{v\}$, comme G' est planaire et connexe, nous pouvons supposer que l'algorithme décrit ci-dessus renvoie une 6-coloration.

Comme $\deg(v) \leq 5$, il existe (au moins) une couleur parmi les 6 couleurs qui n'est pas utilisée dans le voisinage de v. On colorie v avec cette couleur et G est bien 6-colorié. □

Une illustration de cette 6-coloration est donnée en annexe, figure 17.

3.2.4 cutWon et shortWon

Ces deux algorithmes sont très similaires, dans le cas de la détection de cut il suffit de savoir si le graphe de départ sans les arêtes sectionnées est connexe ou non. Si ce graphe n'est pas connexe alors cut a gagné. Dans le cas de shortWon nous construisons le graphe des arêtes sécurisées, et nous vérifions s'il a le même nombre de sommets que le graphe de départ, et s'il est connexe. Si ce graphe n'a pas le même nombre de sommets que le graphe de départ, cela veut dire que certains sommets n'ont pas d'arêtes incidentes sécurisées, donc short ne peut pas avoir gagné, en revanche cette condition est nécessaire, mais pas suffisante, il faut aussi que ce graphe soit connexe.

3.2.5 Obtenir un arbre couvrant

Le premier algorithme implémenté pour obtenir un arbre couvrant du graphe, est l'algorithme de Kruskal [4] puisqu'il a été vu en cours d'algorithmique 3 et 4, et qui permet de déterminer un « Minimum Spanning Tree », ou, en français, un arbre couvrant de poids minimal. Mais étant donné que notre graphe n'est pas pondéré, un parcours en largeur permet également de construire un arbre couvrant. Si l'on note E l'ensemble des arêtes, et V, l'ensemble des sommets du graphe, alors l'algorithme de Kruskal a une complexité dans le pire des cas en $O(|E| \log |V|)$ tandis que la complexité du parcours en largeur est en $O(|V| + |E|)$. C'est donc le parcours en largeur que nous avons retenu pour obtenir un arbre couvrant.

3.3 Intelligence artificielle

La première version de l'IA jouait aléatoirement et, si elle le pouvait, effectuait le coup gagnant. C'est toujours cette IA contre laquelle joue le joueur en mode facile. Pour les niveaux de difficulté moyen et difficile, le premier algorithme implémenté fut un algorithme de type minimax, plus tard amélioré avec un élagage alpha-bêta. Le niveau de difficulté dépend ici du nombre de coups qu'évaluera la fonction. Le dernier niveau de difficulté utilise la stratégie gagnante sur un graphe qui lui est favorable.

3.3.1 Minimax

Algorithme L'algorithme Minimax est une approche fondamentale utilisée dans les jeux à deux joueurs à somme nulle, tels que les échecs ou le jeu de Shannon. Son objectif est de déterminer le meilleur coup à jouer pour un joueur donné, en maximisant son gain potentiel tout en minimisant les pertes potentielles. L'algorithme explore de manière récursive les différentes possibilités de coups jusqu'à une certaine profondeur dans l'arbre de jeu, alternant entre les joueurs pour évaluer les positions. Ici, une partie de la fonction d'évaluation utilisée constate, à une position donnée, lequel des deux joueurs est gagnant, attribuant un bonus de points si l'IA gagne et un malus si elle perd. Ce bonus de points est pondéré par la profondeur de l'évaluation. En effet, si l'IA constate que son prochain coup est gagnant, elle préférera celui-ci à un coup qui gagnera plus tardivement. Après avoir testé plusieurs fonctions d'évaluation, c'est celle-ci qui est ressortie avec les meilleurs résultats. En effet, comme nous jouons avec des graphes planaires, il y a toujours un nœud avec suffisamment peu d'arêtes tel que l'IA trouve rapidement un point d'attaque.

D'autres fonctions d'évaluation ont été testées comme par exemple une fonction qui calcule le degré minimal des sommets du graphe en comptabilisant comme un seul sommet un ensemble relié par SHORT.

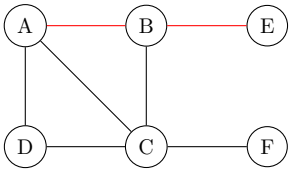


FIGURE 11 –

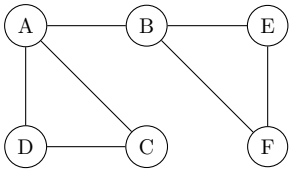


FIGURE 12 –

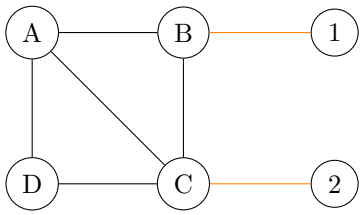
Par exemple dans la figure de gauche, F est de degré 1 car relié uniquement à C et on considérera $\{A, B, E\}$ comme un seul sommet de degré 3. Le problème de cette IA était que seul un parcours du graphe (plus coûteux) peut détecter une situation gagnante pour CUT comme celle de la figure de droite. La fonction minimisant les degrés des sommets n'essaiera donc jamais de séparer le graphe en deux parties comprenant plusieurs sommets non connectés par SHORT.

L'IA implémentée utilise tout de même cette évaluation sur les sommets au départ, au cas où la profondeur ne soit pas assez élevée pour trouver une situation gagnante pour CUT ou SHORT.

Critiques Bien que son concept soit simple, Minimax peut devenir coûteux en termes de calculs, en particulier pour les jeux avec un grand espace d'états, d'où l'utilisation d'optimisations telles que l'élagage alpha-bêta pour réduire le nombre de nœuds explorés.

Aussi, on constate que dans certains cas, l'algorithme ne suit pas une logique « humaine », c'est-à-dire qu'il ne prend pas en compte l'erreur adverse. Par exemple, dans le cas où l'adversaire a deux coups gagnants, peu importe ce que jouera l'IA, elle ne tentera pas d'échapper à l'un des

deux. En ne considérant pas que son adversaire pourrait omettre le coup gagnant, elle jouera au hasard, persuadée que son adversaire gagnera la partie au prochain coup.



Supposons que l'IA joue SHORT et que ça soit son tour. Sachant que même si elle sécurise (B,1) ou (C,2), son adversaire aura toujours la possibilité de gagner la partie au coup suivant, elle décidera de jouer aléatoirement, sachant qu'elle sera perdante au prochain coup quoi qu'il arrive. Un humain tenterait de sécuriser l'une des deux arêtes, espérant une faute de son adversaire au tour suivant.

Plus tard, l'algorithme minimax a été amélioré grâce à l'élagage Alpha-Bêta, apportant plus de rapidité et sans perte d'informations. L'objectif est de réduire le nombre de positions évaluées par minimax, l'idée est donc sensiblement la même à l'exception que l'élagage Alpha-Bêta n'explorera pas certaines branches s'il comprend que l'information qu'il y trouvera sera écrasée par celle qu'il possède déjà.

4 Stratégie gagnante

Le jeu de Shannon est un jeu résolu : il y a toujours un gagnant, celui-ci peut être défini à l'avance si on suppose que les deux joueurs jouent parfaitement. Le graphe sur lequel les joueurs vont s'affronter détermine quel joueur possède une stratégie gagnante. Pour cela nous avons besoin du théorème suivant :

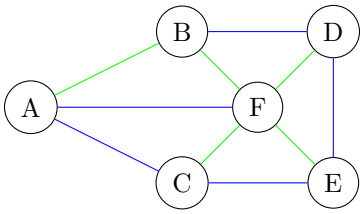
Théorème (tree-packing theorem). *Soit $\mathcal{G}$ un graphe, alors soit $\mathcal{G}$ possède 2 arbres couvrants à supports d'arêtes disjoints, soit il existe une partition des sommets de $\mathcal{G}$ en p ensembles reliés par moins de $2 \cdot |P| - 2$ arêtes.*

Si $\mathcal{G}$ possède deux arbres couvrants à support d'arêtes disjointes, alors SHORT gagnera s'il joue parfaitement. Dans le cas contraire, c'est-à-dire si le graphe ne possède pas deux arbres couvrants à support d'arêtes disjointes, alors CUT gagnera s'il joue parfaitement. [5]

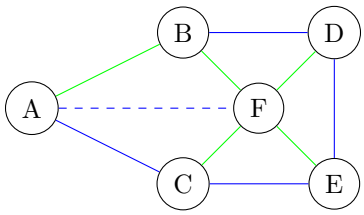
4.1 Jouer pour gagner

SHORT Nous détaillons ici la stratégie gagnante pour SHORT en l'illustrant par un exemple.

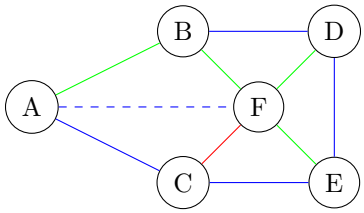
Voici un graphe $\mathcal{G}$ pour lequel SHORT peut appliquer la stratégie gagnante.



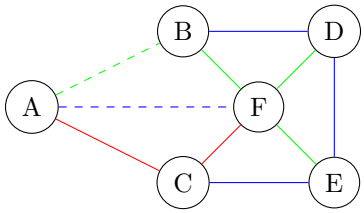
Comme nous pouvons le constater, nous avons deux arbres couvrants sur G, l'un en bleu et l'autre en vert. Chaque fois que CUT coupera une arête de l'un des deux arbres, ce dernier ne sera plus connexe. Soit $\mathcal{S}$ l'ensemble des arêtes sécurisées et C l'ensemble des arêtes coupées. Supposons que les deux arbres soient nommés $\mathcal{A}$ et $\mathcal{B}$ et que CUT coupe une arête de l'arbre $\mathcal{A}$. Si SHORT veut gagner, il devra sécuriser une arête de l'arbre $\mathcal{B}$ de telle sorte que l'ensemble des arêtes sécurisées ajouté à $\mathcal{A}$ reste connexe. Son objectif étant que $\mathcal{A}$+$\mathcal{S}$ et $\mathcal{B}$+$\mathcal{S}$ reste toujours connexe. Voici un exemple plus parlant :



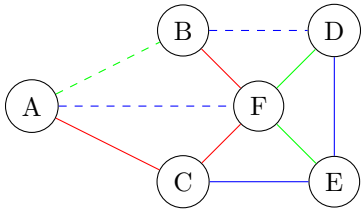
Ici bleu n'est visiblement plus connexe, nous voulons donc rendre bleu + $\mathcal{S}$ connexe donc SHORT devrai sécuriser (C,F)



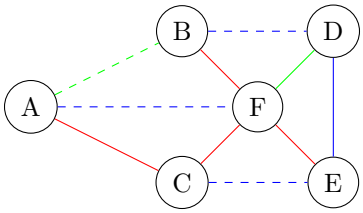
Ici nous avons donc bien bleu + $\mathcal{S}$ connexe ainsi que vert + $\mathcal{S}$ connexe. En continuant, on aurait donc par exemple :



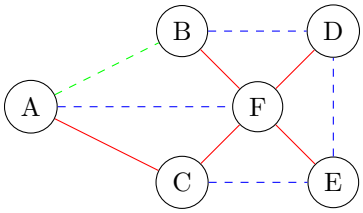
CUT à joué (A,B) et SHORT joue (A,C) pour rendre vert + $\mathcal{S}$ connexe à nouveau



CUT à joué (B,D) et SHORT joue (B,F) pour rendre bleu + $\mathcal{S}$ connexe à nouveau



CUT à joué (C,E) et SHORT joue (F,E) pour rendre bleu + $\mathcal{S}$ connexe à nouveau

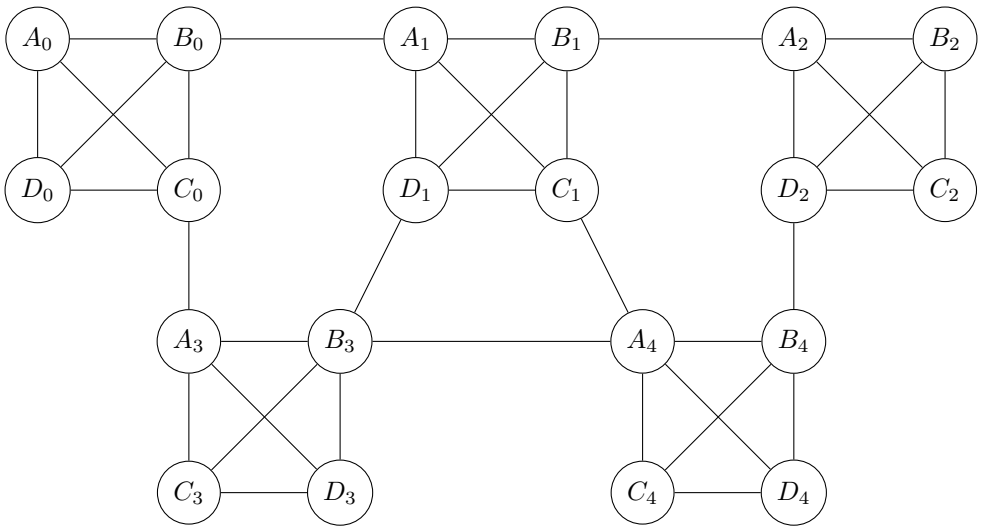


CUT à joué (D,E) et SHORT joue (D,F) pour rendre bleu + $\mathcal{S}$ connexe à nouveau

Ainsi, en ce protégeant à chaque coup, SHORT à réussi à sécuriser un sous arbre couvrant de $\mathcal{G}$.

CUT Nous détaillons ici la stratégie gagnante pour CUT en l'illustrant par un exemple.

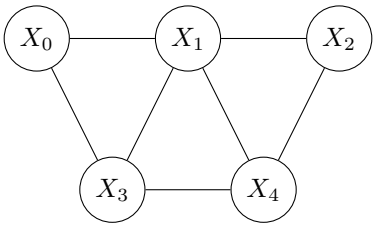
Soit $\mathcal{G}$ un graphe tel que CUT puisse appliquer la stratégie gagnante. Il existe un théorème dont la preuve ne sera pas faite ici qui nous certifie que si $\mathcal{G}$ ne possède pas deux arbres couvrant à support d'arêtes disjointes alors il existe X_0,X_1,...,X_{p-1} une partition des sommets de $\mathcal{G}$ telle que le nombre d'arêtes reliant les X_i est inférieur ou égal à $2p-3$. Voici un tel graphe $\mathcal{G}$:



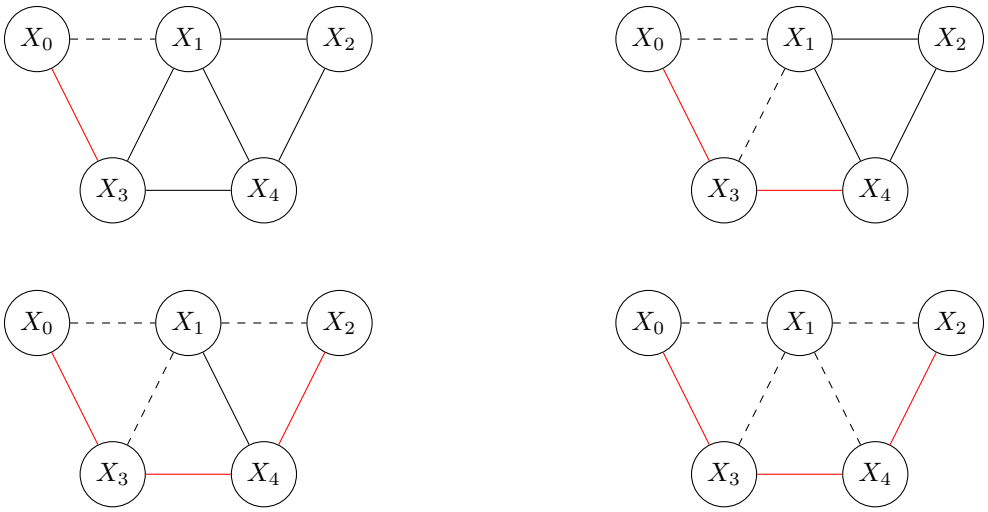
Ce graphe est volontairement d'apparence très complexe, mais nous allons voir que les choses vont très vite se simplifier pour CUT. Nous pouvons constater que ce graphe contient 5 cliques, mais que ces dernières sont mal reliées entre elles. Choisissons donc la partition suivante :

$X_0 = \{A_0, B_0, C_0, D_0\}$

$X_1 = \{A_1, B_1, C_1, D_1\}$
$X_2 = \{A_2, B_2, C_2, D_2\}$
$X_3 = \{A_3, B_3, C_3, D_3\}$
$X_4 = \{A_4, B_4, C_4, D_4\}$
Le graphe G peut donc être vu comme ceci :



Nous constatons donc que nous avons 5 parties reliées par 7 arêtes, ce qui est bien inférieur ou égal à $2p-3 = 2\times5-3 = 7$. CUT peut donc laisser tomber toutes les arêtes internes à chaque partie et se concentrer sur celles extérieures. Ainsi, si CUT se met à couper ces arêtes, même si SHORT tente de l'en empêcher, il ne pourra pas. Voici donc un exemple où nous supposons que SHORT joue pour le mieux (jouer à l'intérieur d'une partie ne ferait que donner de l'avance à CUT).



Inévitablement, SHORT a perdu. Bien sûr la difficulté de cette stratégie est de trouver une partition qui fonctionnera pour CUT.

Trouver une solution Nous venons donc d'expliquer quelle stratégie doit adopter SHORT ou CUT en fonction du graphe afin de gagner. Cependant, pour qu'ils puissent mettre en œuvre leurs stratégies, il faut un algorithme capable de déterminer si un graphe possède deux sous-arbres couvrants avec un support d'arêtes disjoint. Dans ce cas-là, il doit être capable de les retrouver ; sinon, il doit renvoyer la partition permettant à CUT de gagner. Un tel algorithme existe et fait l'objet de la partie suivante.

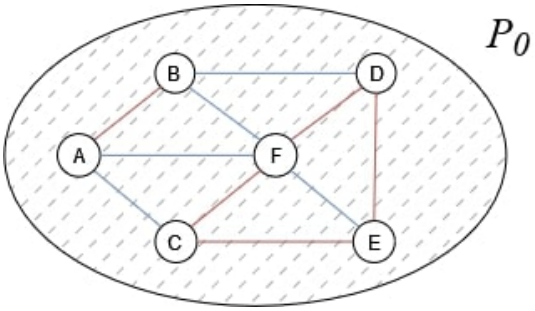
4.2 Algorithme de stratégie gagnante

Ce que nous appelons ici *l'algorithme de la stratégie gagnante* est l'algorithme qui permet de trouver, s'ils existent, deux arbres couvrants à supports d'arêtes disjoints et, sinon, une partition des sommets en p ensembles reliés par strictement moins de $2p-2$ arêtes. L'algorithme est assez complexe et a été conçu avec l'aide du papier *A short proof of the tree-packing theorem* de Tomáš Kaiser publié en 2012 [3] et avec la guidance de l'enseignant encadrant.

4.3 Explication de l'algorithme

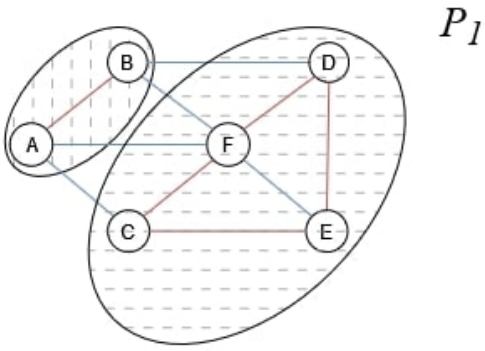
L'objectif est de partir d'une partition des sommets du graphe d'origine et de raffiner cette partition afin d'obtenir le résultat souhaité. Pour mieux comprendre, nous suivrons un exemple sur un graphe $\mathcal{G}$.

D'abord, on choisit un arbre couvrant T_1 de $\mathcal{G}$, cela se trouve facilement avec un parcours en largeur comme expliqué dans la section 3.2.5. Ensuite on créé T_2 le graphe qui contient tous les sommets de $\mathcal{G}$ ainsi que ses arêtes qui ne sont pas dans T_1. Évidemment si à cette étape on constate que T_2 est connexe alors l'algorithme finit, car T_2 contient forcément un arbre couvrant. Supposons que ce n'est pas le cas. Dans notre exemple, on a donc T_1 en bleu et T_2 en rouge.

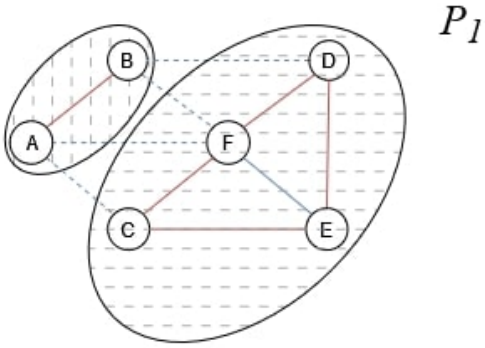


On créé donc notre première partition : on choisit simplement tous les sommets : $P_0 = \{\{A, B, C, D, E, F\}\}$

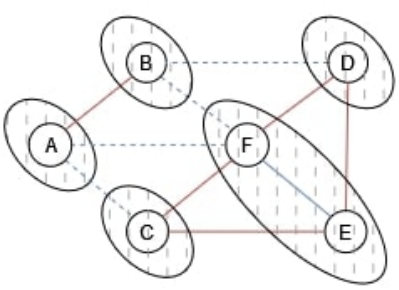
[boucle] On se demande maintenant si l'un des graphes T_1 ou T_2 n'est pas connexe dans P_0. Ici T_2 ne l'est pas, car A est isolé. On décide donc de s'intéresser au graphe qui n'est pas connexe. On raffine ainsi la partition en fonction des composantes connexes de T_2. Dans notre exemple on obtient donc $P_1 = \{\{A, B\}, \{C, D, E, F\}\}$.



On constate que les arêtes $(A, C), (A, F), (B, F)$ et (B, D) se retrouvent entre les deux partitions. On modifie maintenant T_1 et T_2 en leur retirant les arêtes entre les différentes parties de la partition. Ici, T_2 n'est donc pas modifié, mais T_1 perd les arêtes $(A, C), (A, F), (B, F)$ et (B, D). On veut également se souvenir de l'étape à laquelle on a retiré chaque arête, on obtient donc ce qu'on nommera le « level » d'une arête (par défaut ∞). On obtient donc le tableau suivant :
$L = \{(A, B) : \infty, (A, C) : 1, (A, F) : 1, (B, F) : 1, (B, D) : 1, (C, F) : \infty, (C, E) : \infty, (F, D) : \infty, (D, E) : \infty, (F, E) : \infty\}$.

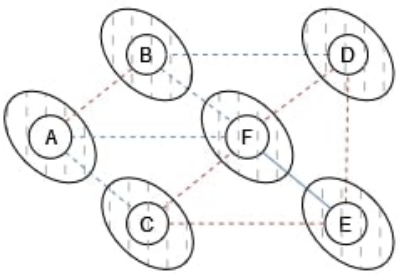


Nous allons donc réitérer l'opération **boucle**. On se demande donc maintenant si à l'intérieur de chaque partie, T_1 et T_2 sont connexes. Ici, on constate que T_2 est connexe dans les deux parties et que T_1 ne l'est dans aucune, en effet les composantes connexes de T_1 sont : $\{\{A\}, \{B\}, \{C\}, \{D\}, \{E, F\}\}$.
On raffine donc la partition de $\mathcal{G}$ afin que T_1 soit connexe dans chaque partie et l'on obtient donc :



P_2

On actualise le tableau L :
$L = \{(A,B) : 2, (A,C) : 1, (A,F) : 1, (B,F) : 1, (B,D) : 1, (C,F) : 2, (C,E) : 2, (F,D) : 2, (D,E) : 2, (F,E) : \infty\}$
Puis on raffine afin que T_2 soit connexe dans chaque partie.



P_3

Enfin nous avons bien une partition où T_1 et T_2 sont connexes au sein de chaque partie. On met à jour L : $L = \{(A,B) : 2, (A,C) : 1, (A,F) : 1, (B,F) : 1, (B,D) : 1, (C,F) : 2, (C,E) : 2, (F,D) : 2, (D,E) : 2, (F,E) : 3\}$

Ici, on a donc P_3 qui est la partition finale, on la note donc P_∞. On compte le nombre d'arêtes entre les éléments de P_∞ qu'on note α. Ici, on a donc $\alpha = 10$. À cette étape, si $\alpha < 2 \cdot |P_\infty| - 2$ alors le graphe est tel que la stratégie gagnante s'applique pour CUT et il devra sectionner les arêtes que nous venons de compter afin de gagner. Ici on $\alpha = 10 > 2 \cdot 6 - 2 = 9$. L'algorithme n'est donc pas terminé et nous allons modifier les deux arbres afin de réitérer les étapes que nous venons d'effectuer.

Pour cela, nous allons chercher un cycle de T_2 qui traverse deux parties de P_∞. Un tel cycle existe toujours, en voici la preuve :

Démonstration. On note $c = |P_\infty|$ et n le nombre de sommets de $\mathcal{G}$. T_1 étant connexe dans chaque élément de P_∞, le nombre d'arêtes de T_1 à l'intérieur de ces parties est $n-c$ (forêt à c composantes connexes). Le nombre d'arêtes de T_1 entre les éléments de P_∞ est donc $n-1-(n-c)) = c-1$. Le nombre d'arêtes de T_2 entre les éléments de P_∞ est donc $\alpha-(c-1) \geq 2 \cdot c-2-(c-1) = c-1$ car $\alpha \geq 2 \cdot c - 2$ par hypothèse. Cependant T_2 n'était pas connexe à l'origine et comme T_2 est

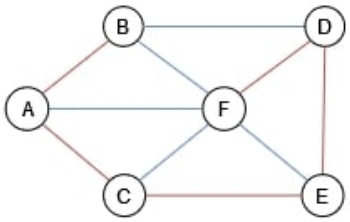
connexe dans chaque partie de P_∞, il existe une partition de P_∞ que l'on note $X = \{X_1, X_2\}$ où $X_1 \cup X_2 = P_\infty$, où il n'y a pas d'arêtes de T_2 qui relie un élément de X_1 à un élément de X_2 et où nous avons :

— Soit il y a $|X_1|$ arêtes de T_2 entre les parties de X_1, c'est-à-dire qu'il existe un cycle entre les éléments de X_1

— Soit il y a $|X_2|$ arêtes de T_2 entre les parties de X_2, c'est-à-dire qu'il existe un cycle entre les éléments de X_2

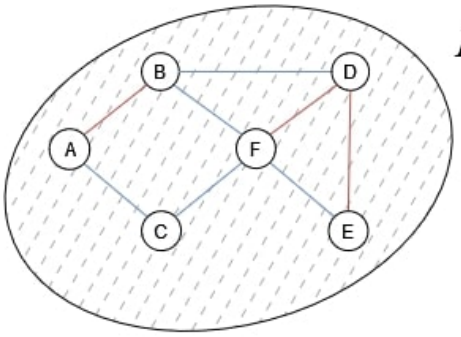
Si aucun de ces deux cas n'était rempli il y aurait au plus $|X_1| - 1 + |X_2| - 1 = c - 2$ arêtes de T_2 entre les éléments de P_∞. Or nous savons qu'il y a $c - 1$ arrêtes de T_1 entre les éléments de P_∞ il y aurait donc au total moins de $2 \cdot c - 3$ de $\mathcal{G}$ entre les éléments de P_∞ ce qui est faux par hypothèse. Il y a donc un cycle de T_2 qui traverse au moins deux parties de P_∞. □

On note donc ce cycle A. Ici, on choisit le cycle $A = \{(C, F)(F, D)(D, E), (E, C)\}$ qui traverse même 4 des parties de P_∞. Ensuite, on choisit une arête de niveau minimum dans A. Ici, les trois ont le même level on va donc choisir arbitrairement (C, F). On la retire de T_2 et on l'ajoute à T_1. On retire également de T_1 l'arête de plus petit niveau (hormis celle que l'on vient d'ajouter) dans le cycle formé par l'ajout de l'arête de T_2 (comme T_1 était un arbre, l'ajout d'une arête créée toujours un cycle). On ajoute cette arête à T_2. Ici, cela serait (A, C) ou (A, F) car elles sont de level 1 toutes les deux. On choisit donc arbitrairement (A, C). On obtient donc le nouveau graphe sur lequel nous allons pouvoir appliquer à nouveau l'algorithme :

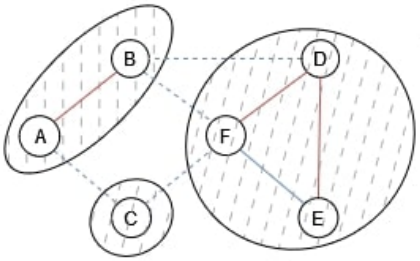


Ici on constate que T_1 est toujours un arbre couvrant ce qui est normal, mais on constate également que T_2 est connexe. On peut donc conclure et dire que le graphe possède deux arbres couvrants et qu'on les a trouvés. Ces arbres sont donc T_1 et un arbre couvrant de T_2 que l'on trouvera facilement avec un parcours en profondeur. L'algorithme s'arrête donc ici, et la stratégie gagnante est donc applicable pour SHORT.

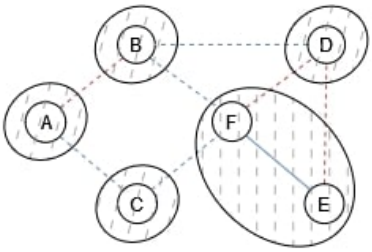
Pour donner un exemple au lecteur, on reprend l'algorithme avec un graphe qui nous donnera une stratégie gagnante pour CUT.



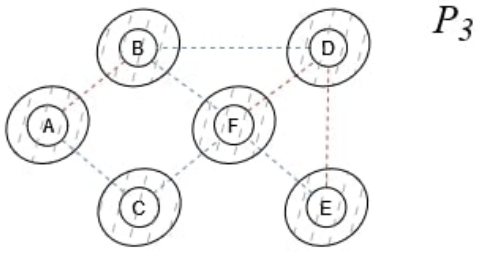
P_0



P_1



P_2



P_3

On a donc P_3 qui est la partition finale, on la note P_∞. On compte le nombre d'arêtes entre les éléments de P_∞ qu'on note à nouveau α. Ici, on a $\alpha = 8$. Or $\alpha = 8 < 2 \cdot 6 - 2 = 10$. L'algorithme est donc terminé et la stratégie gagnante s'applique pour CUT. Ici le graphe est tel que l'ensemble des arêtes que CUT devra couper pour gagner correspond à l'ensemble des arêtes du graphe, mais ce n'est pas toujours le cas.

Nous avons donc exposé deux cas de déroulement de l'algorithme de la stratégie gagnante, un pour CUT et un pour SHORT.

5 Analyse des résultats

5.1 Analyse des performances

5.1.1 Utilisation de la mémoire

La quantité de mémoire utilisée par le programme sur l'écran d'accueil varie en fonction du nombre d'étoiles visibles à l'arrière-plan. L'utilisateur a la possibilité de modifier ce nombre en utilisant un curseur situé dans le coin supérieur gauche de l'écran. Voici les diverses mesures collectées en fonction du nombre d'étoiles :

Nombre d'étoiles	Mémoire utilisée en Mo
0	324
4 000	598
8 000	739

Table 1 – Utilisation de la mémoire en fonction du nombre d'étoiles.

Les paramètres se sauvegardent automatiquement et restent lors d'une réouverture du programme.

En cours de jeu, la consommation de mémoire reste constante dans les parties opposant des joueurs, qu'elles se déroulent en local ou en ligne. Toutefois, lorsqu'on affronte une IA ou lors de combats IA contre IA, on remarque une augmentation de la consommation de mémoire, qui dépend de la profondeur de l'IA. Par exemple, nous avons observé que le programme peut utiliser jusqu'à 1 Go de mémoire lorsque l'IA en mode difficile s'affronte elle-même.

Il est également à noter que lorsqu'on joue contre une IA utilisant une stratégie gagnante, la consommation de mémoire reste stable. Cela s'explique par le fait que les calculs sont effectués avant le début de la partie.

5.1.2 Temps de génération d'un graphe

Comme expliqué dans la section 3.2.2, l'algorithme de génération de graphes planaires dispose de beaucoup de contraintes. Nous proposons ici de donner le temps de génération de 10 000 graphes planaires selon le nombre de sommets désiré.

Nombre de sommets	Temps en secondes
5	0,259
10	1,038
20	6,565
40	43,574
45	64,539

TABLE 2 – Temps de génération de 10 000 graphes planaires en secondes en fonction du nombre de sommets.

Nous observons une croissance exponentielle du temps, et si nous étendons cette analyse à un nombre de sommets de 50, nous constaterions l'impossibilité de générer de tels graphes. Cette limitation découle de la tentative de l'algorithme de placer de plus en plus de sommets, chacun à une certaine distance de ses voisins, dans un espace de taille fixe.

5.2 Tests unitaires

Nous avons également fait plusieurs tests unitaires pour tester les primitives de la classe Graphe. Puisque la méthode *estConnexe* est à la base du jeu entier, nous testons cette méthode sur plusieurs exemples de graphes, comme les graphes complets, qui sont évidemment connexes, et des graphes qui ne le sont pas. Nous testons également les fonctions permettant d'ajouter des sommets et des arêtes sur le graphe, ainsi nous disposons d'un test permettant de générer le graphe complet à k sommets, pour un certain k fixé, et nous vérifions si le nombre d'arêtes est égal à $\frac{k(k-1)}{2}$. Enfin nous testons également la méthode permettant de renvoyer un arbre couvrant.

```java
@Test
    void testGrapheCompletNonConnexe() {
        Graph g = new Graph();
        int k = 100;
        // Création graphe complet à k sommets avec un sommet isolé (k+1 sommets au total)
        for (int i = 0; i < k; i++) {
            Vertex v = new Vertex(0, 0);
            g.addVertex(v);
            for (int j = 0; j < i; j++)
                g.addEdge(new Pair<>(g.getVertices().get(i), g.getVertices().get(j)));
        }
        g.addVertex(new Vertex(0, 0));
        assertEquals(g.getEdges().size(), k * (k - 1) / 2);
        assertFalse(g.estConnexe());
    }
```

FIGURE 13 – Création d'un graphe complet à k sommets, et test de connexité.

6 Mode en ligne

Comme dit dans la section 2.2, notre projet comporte un mode en ligne, nous allons expliquer son fonctionnement.

6.1 Technologies utilisées

Pour concevoir un mode en ligne, nous nous sommes basés sur le modèle client-serveur. Côté client, en Java, deux classes permettent de communiquer avec le serveur :

— WebSocketClient
— HttpsClient

La classe **WebSocketClient** permet de communiquer avec un autre client à travers une connexion web socket. C'est la façon la plus simple de procéder, car cela offre un canal de communication bidirectionnel. C'est à travers cette classe que le client peut demander au serveur la création d'une partie, envoyer des coups à l'adversaire et transmettre le graphe entre les deux joueurs.

La classe **HttpsClient** sert à demander ou à envoyer certaines informations au serveur sans la nécessité pour celui-ci de devoir les envoyer à un second client.

Dans notre cas, cette classe est utilisée à deux reprises :

1. À chaque fin de partie, lorsque le client envoie au serveur un résumé de celle-ci. Ce résumé contient :
   — le type de partie : en ligne, IA contre IA, joueur contre IA, etc.
   — l'identité du gagnant : SHORT ou CUT
   — une seed : l'explication de ce qu'est cette seed sera donnée dans la prochaine section.
2. Lors d'une création de compte ou de connexion pour accéder au mode compétitif. Le mot de passe et le pseudo sont transmis à l'aide du protocole **https**.

Concernant la partie serveur, nous avons choisi d'utiliser le langage **Rust** pour ses performances et sa sécurité. De plus, une base de données *MongoDB* [7] est utilisée pour le stockage des parties et des comptes utilisateurs. L'accès à cette base de données se fait uniquement par le serveur : le client ne peut pas émettre de requêtes à la base de données sans passer par le serveur.

6.2 Fonctionnement du mode en ligne

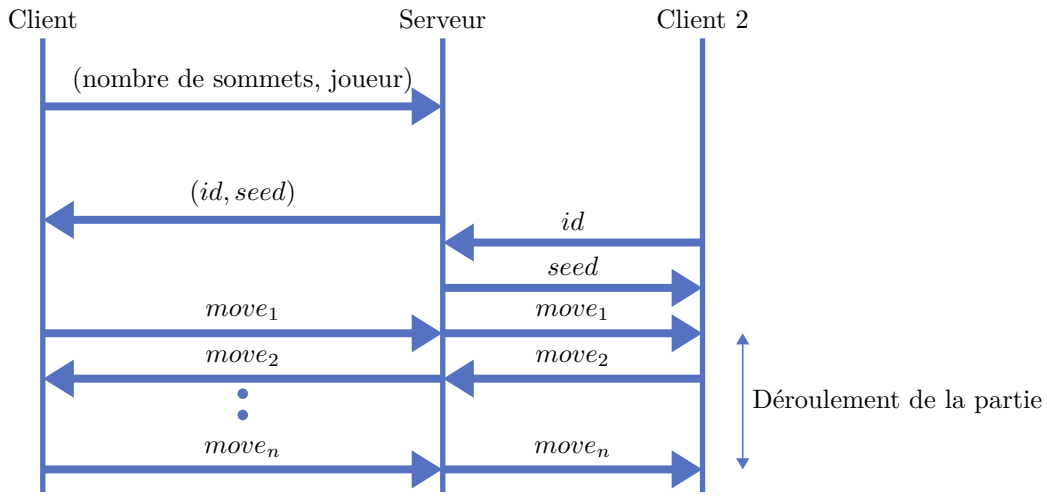
Sur l'écran d'accueil, l'un des modes est le « Joueur vs Joueur online », dans ce mode de jeu un utilisateur a deux choix :

— Configurer une partie et cliquer sur « Créer ».
— Mettre dans le champ de texte prévu à cet effet un **id** pour rejoindre une partie.

Lorsqu'un utilisateur clique sur « Créer », le client envoie au serveur une donnée permettant de créer une partie, cette donnée contient deux champs : le tour du créateur (est-ce que le joueur qui crée une partie veut être CUT ou SHORT ?) et le nombre de sommets désiré. Le serveur répond avec deux informations : un identifiant de partie et une seed. La seed est inconnue du joueur, mais l'identifiant de partie s'affiche sur l'écran.

Pour rejoindre une partie, un joueur doit rentrer un identifiant de partie dans le champ de texte. Si une partie avec un tel identifiant existe, alors la partie se lance pour les deux joueurs. Une observation importante est qu'à aucun moment le graphe n'a été transmis à travers le réseau, pourtant celui-ci est généré aléatoirement et est identique pour les deux joueurs. C'est ici qu'intervient le rôle de la seed. Cette seed est reçue par les deux joueurs et est transmise au constructeur **Random** de la classe **Graph** afin de fixer le hasard. Les deux clients sont donc à présent en mesure de générer le même graphe à condition qu'ils utilisent la même version du programme. En effet, si les versions diffèrent, des modifications dans l'algorithme de génération

du graphe peuvent survenir, entraînant des différences dans les graphes produits. Étant donné que la seed est un entier codé sur 64 bits, cela permet de générer au maximum 2^{48} graphes différents (à isomorphisme près) d'après la documentation de Java [2].



6.3 Mode compétitif

Le mode compétitif offre la possibilité de créer un compte (sans adresse mail requise) en choisissant un nom d'utilisateur ainsi qu'un mot de passe. Lors de la création du compte, il est demandé de choisir un pseudo qui ne comporte pas d'espace et composé d'au moins trois caractères. Ensuite, vous devrez choisir un mot de passe ayant comme seule restriction de ne pas comporter d'espace. Enfin, il vous sera demandé de confirmer le mot de passe en le saisissant à nouveau.

Pour se connecter à son compte, il faut sélectionner « se connecter » puis rentrer le pseudo et le mot de passe.

Les données sont transmises au serveur via le protocole **https** et ne sont donc pas susceptibles d'être interceptées. Une fois reçues, le serveur vérifie la validité des informations. Si celles-ci sont valides, le mot de passe est haché à l'aide de l'algorithme *Bcrypt* [6] avec un coût de 12. Cet algorithme a été choisi pour sa robustesse et le fait qu'un salage du mot de passe soit inclus par défaut.

```
_id: ObjectId('66310961fea8a38068868c07')
pseudo : "Julien"
password_hash : "$2b$12$NUteMmv4sLAUyk9bWkNZDux8jbITYuSzN8KVvXrAncbqloLNlEexS"
elo : 1212
nb_games : 3
```

FIGURE 14 – Enregistrement d'un joueur dans la base de données.

L'intérêt d'un mode compétitif réside dans le fait que chaque compte possède un classement Elo. Celui-ci est de 1200 lors de la création du compte et évolue en fonction des parties jouées contre d'autres joueurs connectés : il augmente en cas de victoire et diminue en cas de défaite.

Notons qu'un joueur connecté peut faire une partie contre un joueur qui n'a pas de compte, mais dans ce cas, le classement Elo ne sera pas affecté.

7 Bilan et conclusions

Le projet satisfait les exigences énoncées dans le cahier des charges initial. Les principales fonctionnalités ont été intégrées avec succès, en voici un récapitulatif :

— Implémentation du jeu de Shannon :

Étant la principale et plus importante exigence du projet, ce fut la première tâche réalisée par notre groupe. La première version de notre projet permettait à deux joueurs de s'affronter en local.

— IA :

L'intégration de notre IA basique suivie de l'implémentation de l'algorithme Minimax a permis de mettre à la disposition de l'utilisateur une IA opérationnelle avec laquelle jouer.

— Interface graphique :

L'interface graphique a été soigneusement conçue pour offrir une expérience intuitive. Les éléments visuels ont été pensés pour évoquer le thème de l'espace tout au long de l'expérience de jeu, offrant ainsi à l'utilisateur une immersion agréable et immersive.

— Algorithmes :

Les algorithmes essentiels au bon fonctionnement de l'application ont été développés. Parmi ces derniers, on trouve par exemple la vérification de la connexité avec « estConnexe », l'évaluation du joueur gagnant avec « cutWon » et « shortWon », la génération de graphes planaires ou l'algorithme de 6-Coloration

— Mode online :

Ce mode offre la possibilité de jouer en ligne avec un adversaire, qu'il soit présent sur le même réseau ou non. De plus, il intègre un système d'Elo pour apporter une dimension compétitive à notre jeu.

— Page de statistiques :

Sur l'écran d'accueil, nous pouvons apercevoir dans le coin supérieur gauche un point d'interrogation rouge, en cliquant dessus, une page de statistiques sur les parties jouées s'affiche. On y trouve le nombre de parties jouées au total, le nombre de fois que SHORT ou CUT a gagné ainsi que le nombre de parties en ligne effectuées.

Bien que le projet ait été mis en œuvre avec succès, certaines fonctionnalités intéressantes qui pourraient y être ajoutées.

Tout au long du projet, nous avons rencontré des difficultés avec l'interface graphique. Dès le début, nous avons opté pour Java afin de développer notre jeu. Après une petite étude des bibliothèques graphiques existantes, les candidats possibles pour nous étaient sur Swing, JavaFX et Tkinter. Swing étant le prédécesseur de JavaFX et n'étant plus du tout utilisé, nous n'avons rien tenté avec cette bibliothèque. Nous avons alors commencé le projet en Tkinter, mais la bibliothèque étant très limitée, nous sommes directement passés sur JavaFX. Au début, la bibliothèque était parfaite pour notre utilisation. Puis lorsque le projet était plus avancé, nous avons décidé d'améliorer considérablement notre interface graphique par l'ajout de planètes, d'un fond en mouvement et de boutons plus futuristes par exemple. Nous avons alors rencontré certaines limites avec JavaFX qui furent un peu plus compliquées à contourner, de ce fait, une amélioration intéressante serait de rendre le programme « responsive ».

Une autre piste d'amélioration serait l'implémentation d'autres IA pour permettre à l'utilisateur de choisir l'IA de son choix. On peut alors citer la recherche arborescente Monte-Carlo, Negamax et NegaScout qui sont des variantes de Minimax.

Notre mode en ligne peut lui aussi être amélioré en rajoutant des modes compétitifs tels que des tournois. Il serait aussi possible d'ajouter des salons de discussion pour que les joueurs puissent communiquer entre eux. Une autre problématique, liée au mode en ligne, est qu'il est impossible de proposer un bouton pour changer le mot de passe de son compte sans avoir à refaire le système à l'aide d'un token d'authentification. De plus, actuellement, le jeu de Shannon n'est pas implémenté côté serveur, cela veut dire que le serveur ne vérifie pas la légalité des coups joués par les joueurs. Cela pose deux problèmes :

— Les joueurs peuvent modifier le client afin de tricher, et le serveur ne sera pas en mesure de détecter la triche.
— Les parties en cours peuvent être modifiées par un agent externe en possession de l'identifiant de partie.

Une solution pour régler ce problème serait de recoder entièrement la logique de jeu côté serveur, et de transmettre un token de connexion par client.

Pour finir, le dernier point d'amélioration concerne l'accessibilité de notre jeu qui n'est pas adapté pour des gens ayant un handicap visuel. Il faudrait potentiellement rajouter un mode daltonien, audiovisuel ou même photosensible.

En conclusion, le projet du jeu de Shannon répond au cahier des charges et aux objectifs que nous nous étions fixés. Néanmoins, il reste quelques points ouverts et le jeu peut encore évoluer pour offrir aux utilisateurs une expérience toujours plus agréable.

Références

[1] M. Garner. In : *The Second Scientific American Book of Mathematical Puzzles and Diversions.* NY : Simon et Schuster, 1961, p. 86-87.

[2] *Java Platform, Standard Edition 8 API Specification - Class Random.* URL : https://docs.oracle.com/javase/8/docs/api/java/util/Random.html.

[3] Tomáš Kaiser. "A short proof of the tree-packing theorem". In : *Discrete Mathematics* 312.10 (2012), p. 1689-1691. ISSN : 0012-365X. DOI : https://doi.org/10.1016/j.disc.2012.01.020. URL : https://www.sciencedirect.com/science/article/pii/S0012365X12000349.

[4] *Kruskal's algorithm.* URL : https://en.wikipedia.org/wiki/Kruskal%27s_algorithm.

[5] Alfred Lehman. "A Solution of the Shannon Switching Game". In : *Journal of the Society for Industrial and Applied Mathematics* 12.4 (1964), p. 687-725. DOI : 10.1137/0112059. eprint : https://doi.org/10.1137/0112059. URL : https://doi.org/10.1137/0112059.

[6] Niels Provos et David Mazières. "A Future-Adaptable Password Scheme". In : *1999 USENIX Annual Technical Conference (USENIX ATC 99).* Monterey, CA : USENIX Association, juin 1999. URL : https://www.usenix.org/conference/1999-usenix-annual-technical-conference/future-adaptable-password-scheme.

[7] *Qu'est-ce que MongoDB ?* URL : https://www.mongodb.com/fr-fr/company/what-is-mongodb.

[8] *Théorème de Zermelo.* URL : https://en.wikipedia.org/wiki/Zermelo%27s_theorem_(game_theory).

8 Annexe

8.1 Diagramme de classes

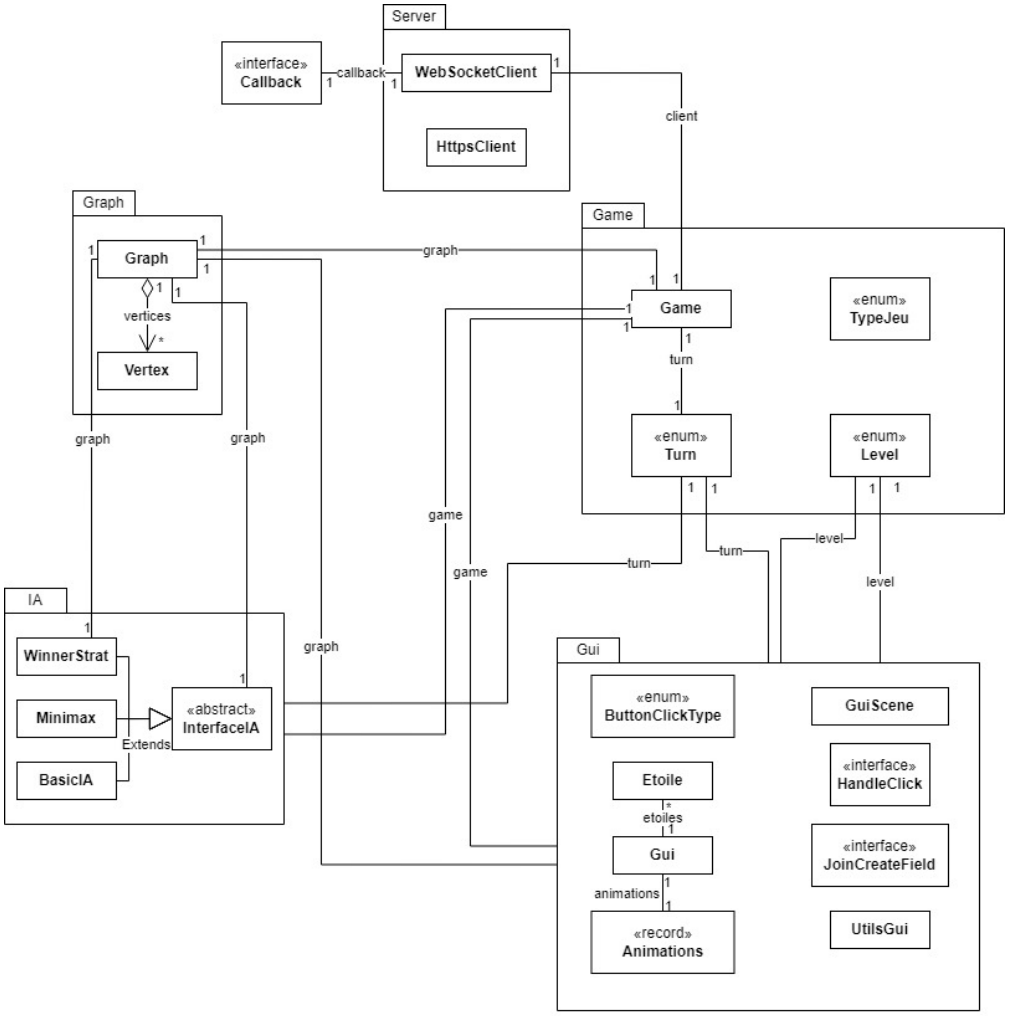


FIGURE 15 – Diagramme de classes.

8.2 Complément de preuve

Tout graphe planaire connexe admet un sommet de degré au plus 5.

Démonstration. Une conséquence de la formule d'Euler pour les graphes planaires connexes ayant au moins trois sommets, est que

$m \leq 3n - 6$ avec m le nombre d'arêtes du graphe et n, le nombre de sommets.

Supposons par l'absurde que tout sommet soit de degré au moins 6, étant donné que le nombre d'arêtes d'un graphe est égal à la moitié de la somme des degrés des sommets, nous obtenons :

$$m = \frac{1}{2} \sum_{v \in V} \deg(v) \geq \frac{1}{2} \sum_{v \in V} 6 = \frac{1}{2} 6n = 3n$$

Ce qui contredit la formule d'Euler. □

8.3 Images

Pour tester l'algorithme de la stratégie gagnante, nous avons créé un mode développeur permettant d'afficher les deux arbres couvrants à support d'arêtes disjointes s'ils existent :

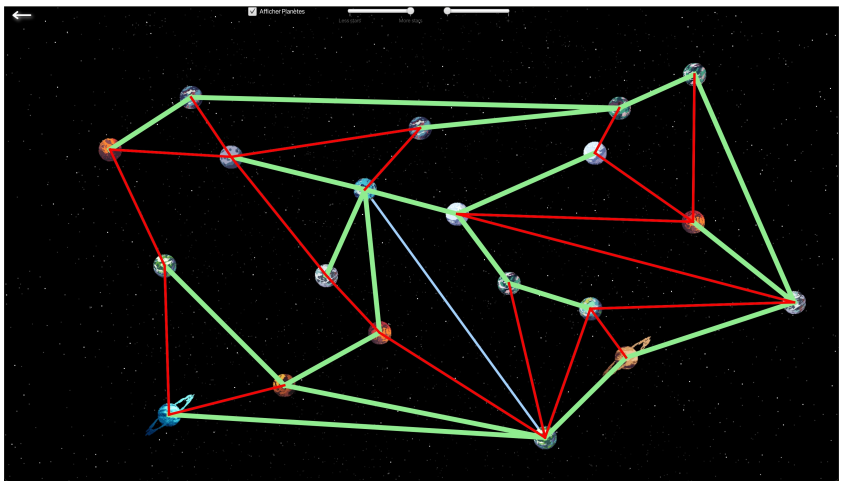


FIGURE 16 – Affichage des deux arbres couvrants disjoints.

Voici une capture d'écran montrant une 6-coloration du graphe :

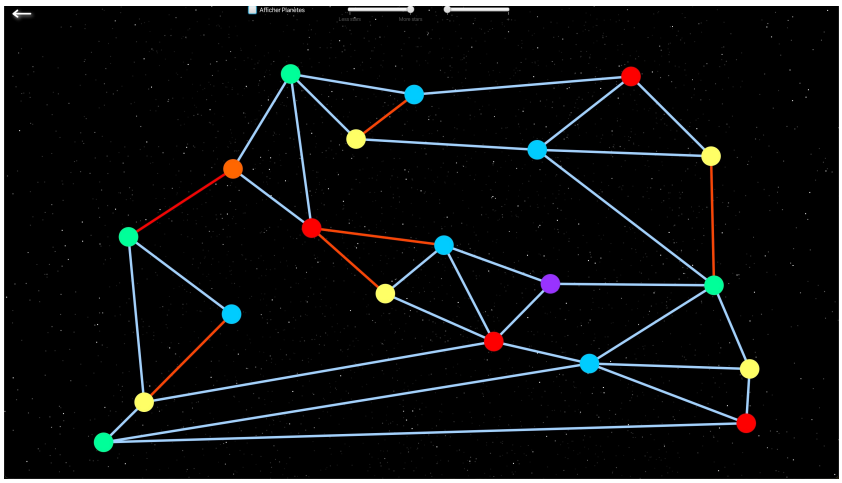


FIGURE 17 – Affichage d'une 6-coloration.