ENSEIRB-MATMECA

Semestre 6

Projet en C

Projet Rails

I1
Alexandre Herve
Maeva Grondin
Nicolas Hannoyer
Simon Brouard

Mardi 10 Mai 2016

SOMMAIRE

1 Les casaniers du Rail

1.1 Les règles du jeu

Les casaniers du Rail est un jeu basé sur les règles du jeu de société les aventuriers du rail. Les joueurs utilisent une aire de jeu commune composée de villes et de rails reliant ces villes. Chaque joueur joue à tour de rôle. À chaque tour le joueur a le choix entre trois actions:

- prendre deux cartes wagon distribuées aléatoirement.
- défausser k cartes wagon de la même couleur afin de s'approprier une liaison de longueur k
- piocher un objectif supplémentaire, qui s'ajoute à la liste.

Le jeu s'arrête lorsque l'un des joueurs dispose de moins de 2 wagons dans sa réserve et on termine alors le tour en cours.

1.2 Le but du jeu

Le gagnant de la partie est le joueur qui possède le plus de point à la fin de la partie. Il existe plusieurs façons de gagner des points:

- poser des rails rapporte un nombre de points dépendant de la longeur du rail : $(nbr\_points(k) = 1 + \frac{k*(k-1)}{2})$ avec k la longueur du rail.
- chaque objectif rapporte des points s'il est rempli, mais dans le cas contraire en fait perdre.

Le score de chaque joueur est calculé par le serveur à la fin de la partie. Il dépend alors de la stratégie adoptée par chaque joueur, en effet chaque joueur détermine une stratégie visant à trouver le ou les chemins optimaux pour relier les villes de ses objectifs tout en prenant le maximum de points avec les rails.

2 Conception du jeu

Le projet sur les casaniers du rails consiste à implémenter le jeu afin qu'un ensemble de clients puisse jouer. Pour cela, il a été choisi de créer un serveur représentant le plateau de jeu et ses informations, et qui intéragit avec les clients au cours du déroulement de la partie. L'interaction serveur-client nécessite une interface, c'est pourquoi après discussion avec les autres groupes dirigés par David Bonnin, nous avons décidé de faire une interface commune à tous les groupes afin de pouvoir lancer un client quelconque sur un serveur quelconque.

2.1 Définition de l'interface

La réalisation de l'interface s'est faite grâce à l'étude des règles des casaniers du Rail et des actions possiblement effectuées par un joueur durant une partie. Ainsi dans un premier temps ont été définies les informations que le serveur doit transmettre au client pendant la partie, ces informations sont séparées en deux catégories : les informations générales sur la partie (transmises à l'initialisation des clients) et les informations sur les modifications apportées au jeu à chaque tour (transmises à chaque tour à tous les clients). Ainsi, chaque client actualise lui même à chaque tour, dans un espace personnel de données, les changements que lui envoie le serveur. Chaque client a

donc sa propre vision du jeu et n'agit pas directement sur le plateau de jeu général géré par le serveur, car il n'y a pas accès, et cela évite donc de possibles tricheries.

Voici les différentes informations données aux clients au moment de leur initialisation :

- l'identifiant du joueur
- le nombre de villes, le nombre de rails, le nombre d'objectifs, le nombre de wagons donnés aux joueurs en début de partie
- le nombre de joueurs
- les informations sur les rails du plateau de jeu (les deux villes reliées, la longueur, la couleur)
- les informations sur les objectifs du jeu (les deux villes à relier, le nombre de points de l'objectif). Le choix de donner tous les objectifs à chaque joueur vient du fait que lorsque nous jouons régulièrement au aventuriers du rail, nous finissons par connaître les objectifs.

Les informations données par le serveur à chaque tour sont les suivantes (ces informations concernent uniquement le dernier tour):

- le nombre de wagons utilisés par chaque joueur
- le nombre de cartes piochées ou défaussées par chaque joueur
- le nombre d'objectifs piochés par chaque joueur
- le nombre des rails posés par chaque joueur
- les informations sur les rails posés (les identifiants du rail et du joueur qui a posé ses wagons sur ce rail)
- les objectifs piochés par le joueur au dernier tour
- les couleurs des cartes wagon piochées ou défaussées par le joueur au dernier tour

Ensuite, nous nous sommes intéressés aux données que doit rendre le client au serveur, celles-ci correspondent aux choix ou actions que fait un joueur pendant une partie. Le client renvoie donc :

- l'action qu'il souhaite effectuer (lorsqu'on l'appelle pendant un tour de jeu)
- son choix des objectifs (si lors du tour de jeu l'action choisie est d'en piocher)

En receuillant toutes ces informations échangées entre le client et le serveur nous avons abouti à l'interface que l'on peut trouver dans le fichier d'entête $interface.h$.

2.2 Conception du serveur

La conception du serveur est plutôt simple puisqu'une partie consiste tout d'abord à initialiser le plateau de jeu avec ses informations (informations données par un fichier texte), les différents clients avec les informations que nous avons citées précédemment et les informations transmises aux clients par le serveur à chaque tour. Puis à lancer la partie où à chaque tour le serveur demande l'action souhaitée par le client, avant de vérifier si cette action est valide et de l'effectuer si tel est le cas. Enfin quand le dernier tour est terminé (lorsqu'il reste moins de 2 wagons à l'un des joueurs), le serveur évalue les points de chaque client et établit le gagnant.

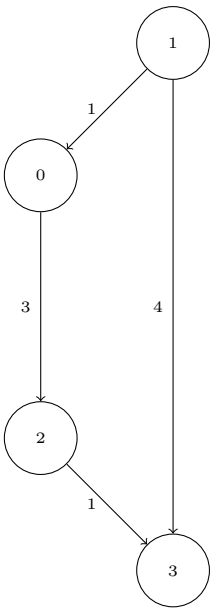
2.3 Conception des clients

Le client doit au cours de la partie renseigner le serveur sur les actions à effectuer ou sur les objectifs qu'il souhaite garder après en avoir piochés.

Pour cela le client le plus simple à implémenter est le client aléatoire, il choisit l'une des trois actions aléatoirement, puis s'il doit poser il choisit un rail de façon aléatoire sans considérer si celui-ci est déjà pris ou non (la serveur établira si le client triche), et s'il doit choisir parmis plusieurs objectifs son choix est également aléatoire. Cependant ce type de client n'est pas efficace puisqu'il n'atteindra que très rarement ses objectifs, les points d'objectifs deviennent alors des malus et peuvent amener à un score négatif.

C'est pourquoi un deuxième type de client à été conçu, celui-ci utilise la notion d'arbre couvrant de poids minimal. Il cherche donc un ensemble de chemins de somme de poids minimale qui relie tous les sommets du plateau. Il construit alors son train en suivant les arêtes de cet arbre. Avec cette capacité de remplir les objectifs, deux stratégies principales peuvent voir le jour. La première étant que lorsque le client finit les objectifs qu'il possède, il en pioche d'autres, l'arbre couvrant de poids minimal met en avant cette stratégie car pour un grand nombre de villes à relier les chemins sont souvent optimaux. La deuxième stratégie qui n'utilise l'arbre couvrant de poids minimale que dans le but de trouver un chemin, consiste à finir les objectifs initiaux puis à construire les rails de plus grande longueur sur le plateau, cette stratégie ne cherche donc pas à obtenir les points par les objectifs mais par la construction de rails. Notre groupe de projet a préféré choisir la deuxième stratégie car ainsi le score final ne sera pas affecté par les malus dûs aux objectifs non faits. De plus nous savions par expérience du jeu qu'un grand rail rapporte énorment de points et que ces points compensent généralement les points obtenus par objectifs, puis il y a le fait de ne pas perdre de tour à piocher des objectifs ce qui permet de terminer la partie plus rapidement, ce qui laisse moins de temps aux autres joueurs pour finir leurs objectifs.

Si le joueur possède un seul objectif ou peu d'objectifs, la méthode utilisant l'arbre couvrant minimal n'est pas forcément la plus efficace. Regardons l'exemple ci-dessous :



Dans ce cas ci, imaginons que le joueur possède l'objectif ($1 \rightarrow 3$), la méthode utilisant l'arbre couvrant de poids minimal choisira pour relier la ville 1 à la ville 3 le chemin $1 \rightarrow 0 \rightarrow 2 \rightarrow 3$ car il suivra le chemin indiqué par l'arbre. Or le chemin le plus court entre 1 et 3 est de relier 1 à 3 directement. Dans un problème d'optimisation des chemins, il aurait fallu implémenter une stratégie résolvant le problème de Steiner afin de ne tenir compte que des villes objectif. En effet un abre couvrant de Steiner relierait seulement les villes des objectifs et les villes de passage (villes nécéssaires pour relier deux villes objectif), on aurait dans l'exemple si dessus le chemin $1 \rightarrow 3$ directement. Cette nouvelle stratégie n'est pas implémentée dans le projet.

3 Le serveur de jeu

Le serveur de jeu est la partie qui gère l'ensemble des clients ainsi que l'état global du jeu. Il transmet à chaque tour à chaque client les modifications du jeu et répond à leurs requêtes. Le serveur est également le garant des règles et doit donc vérifier la bonne application de celles-ci. Il est le seul à pouvoir mettre fin à une partie.
Le serveur possède la version officelle de l'aire de jeu. Il doit donc garantir la préservation du plateau de jeu et modifier ce dernier en fonction des actions des clients.

3.1 Le plateau de jeu

Le plateau de jeu peut être vu comme un graphe, et l'une des représentations courantes des graphes est la matrice d'adjacence. C'est pourquoi dans le serveur le plateau est représenté par une matrice. Un rail possède plusieurs informations nécessaires au déroulement de la partie (ville a, ville b, longueur, couleur), pour mémoriser ces informations une structure *Rail* a été créée. Le plateau aurait alors pu être une matrice de structure *Rail* toutefois certains cas nécessitent une représentation plus complexe. En effet il est nécessaire de pouvoir avoir plusieurs rails entre une ville a et une ville b. Pour cela l'utilisation d'une autre structure (la structure *Link*) a paru adéquate, elle fournit le nombre de rails entre deux sommets et un tableau de structures My_rail. Pourquoi utiliser My_rail plutôt que *Rail* ? En supplément des informations données par la structure *Rail* il est nécessaire de connaître l'état du rail, c'est à dire s'il est utilisé par un joueur et si oui par quel joueur. Un identifiant du joueur *id_joueur* est donc ajouté à la structure.

Cependant le plateau de jeu n'est pas seulement l'ensemble des villes et des rails, on peut aussi considérer que la pioche, que ce soit de cartes wagon ou d'objectifs, et le nombre de wagons donnés aux joueurs composent également le plateau. Le serveur a donc une structure Map_info qui regroupe ces différentes informations (nombre de villes, nombre de rails, nombre de wagons donnés aux joueurs en début de partie). De plus, si le jeu possède plusieurs couleurs de wagons, il faut identifier ces couleurs, d'où l'utilisation du tableau *col*.

Un plateau de jeu en général devrait s'arrêter à ces informations, la notion de plateau de jeu a cependant été adaptée au projet. Le groupe a décidé de vérifier si les IA trichent. Or, pour savoir si une IA triche, il est nécessaire de connaître sa main. Les mains de cartes wagon et de cartes objectif ont donc été ajoutées au plateau. Ainsi, le serveur n'acceptera de placer un joueur sur un rail demandé que s'il a au préalable vérifié que le joueur qui émet la demande a bien le nombre de wagons nécessaires, ainsi que la couleur de cartes-wagons en bon nombre.

3.2 Le déroulement de la partie

Le déroulement d'une partie de casaniers du rail se décompose en trois parties : l'initialisation des diverses informations (informations du plateau, des joueurs et celles nécessaires au tour de jeu), le jeu (c'est à dire l'enchainement des actions des différents joueurs jusqu'à la fin de la partie) et le calcul des points pour désigner le gagnant.

3.2.1 Initialisation

Le serveur initialise dans un premier temps le plateau de jeu. Un fichier texte est fourni en entrée afin de donner les informations nécessaires à cette initialisation. Un parser a été implémenté pour transférer les données du fichier texte aux structures correspondantes. Il lit les lignes non commentées une par une. La première ligne correspond aux informations : nombre de villes,

nombre de rails, nombre d'objectif et nombre de wagons par joueurs. Ensuite il analyse les lignes suivantes d'après leur nombre de mots, la ligne correspond soit à une déclaration de ville (2 mots), soit à une déclaration de rail (4 mots), soit à une déclaration d'objectif (3 mots). Puis les transmet les données aux tableaux correspondants en changeant leur type si besoin (ex : une chaîne de caractère contenant un entier est transformé en entier). Un problème avec le fichier France.txt initialement donné apparait à cause de ce comptage de mots: prenons la ligne "14 Le Mans", cette ligne compte 3 mots au lieu de deux et est considérée comme une déclaration d'objectif. Le fichier initial a donc été modifié, la ligne "14 Le Mans" devient "14 Le_Mans".

Comme dit précédemment, le serveur gère également les mains des joueurs. Que ce soit pour les wagons ou pour les objectifs les mains sont réprésentées de la même manière, c'est à dire par un tableau de mains de longueur le nombre de joueurs et par un tableau de nombre de cartes par main coincidant avec le tableau de mains. Chaque main est elle représentée par un tableau de cartes (les cartes sont des entiers correspondant soit à une couleur soit à l'indice de l'objectif dans le tableau des objectifs) de longueur le nombre de cartes présentes dans la main.

Les informations sur les joueurs sont les fonctions qu'ils utilisent, l'interface définit le prototype de ces fonctions mais elles sont propres aux IA. L'initialisation range les fonctions des IA dans un tableau de structures f_player (la structure possède les 4 fonctions $init\_player$, $play\_turn$, $choose\_objective$ et $free\_player$ nécessaires à l'interface pour un joueur).

Pour que chaque joueur ait connaissance des données de la partie il faut lancer les fonctions $init\_player$ pour chacun d'entre eux. Cette fonctions leurs donne leur numéro de joueur, le nombre de joueurs total de la partie, le nombre de villes, le nombre de rails, le tableau de rails (qui contient l'ensemble des rails du plateau), le nombre de wagons qu'ils possèdent au début, le nombre d'objectifs ainsi que le tableau d'objectifs (qui contient l'ensemble des objectifs).

Après avoir initialiser ces données, on pourrait lancer la partie. Cependant à chaque tour les clients ont besoin de connaître les informations modifiées pendant le dernier tour. Il nous faut donc initialiser ces informations et les stocker avant de lancer le jeu. C'est ce que fait la fonction $init\_turn\_informations$. Quelques noms donnés aux informations de tour peuvent être ambigüs: le tableau *objectives* correspond aux nombre de cartes objectif piochées par les joueurs durant le dernier tour et le tableau *cards* correspond aux couleurs des cartes wagons piochées ou défaussées (pour savoir si elles ont été piochées ou défaussées il suffit de regarder le tableau $wagon\_cards\_modif$).

Le serveur peut enfin lancer la partie.

3.2.2 Le jeu

La première action que l'on fait lorsqu'on commence une partie d'aventuriers du rails est de choisir parmis les objectifs distribués préalablement ceux que l'on souhaite garder. Il en est de même dans les casaniers du rail, un appel de la fonction *choose_objective* est donc lancé sur chaque joueur.

Le premier joueur peut alors commencer son tour, les règles définissent trois actions possibles : piocher des cartes wagon, construire un rail ou piocher des cartes objectif. Le serveur demande au joueur quelle action celui ci souhaite effectuer grâce à la fonction *play_turn*, l'action rendue dépend de la stratégie adoptée par l'IA. Une fois l'action connue du serveur, il vérifie que l'action est valide. Si l'action est de piocher des cartes wagons, cette action est forcément valide puisque nous considérons une pioche infinie et que la main du joueur peut être aussi grande qu'il le veut. Ce n'est pas valable pour les cartes objectifs car le nombre d'objectifs est limité par le fichier de départ et parce qu'un objectif ne peut être pioché qu'une fois. Pour évaluer si poser des wagons sur un rail est valide ,il vérifie trois informations : si le rail est déjà pris, si le joueur possède assez de wagons pour les poser et s'il possède assez de cartes wagons de la couleur du rail. Si ces trois vérification

sont passées, le serveur applique alors ces modifications au plateau de jeu et aux informations de tour.

Ces suites d'opérations effectuées sur chaque joueur constitue un tour de jeu, lorsque que le nombre de wagons d'un joueur tombe en dessous de 2, un dernier tour de jeu est effectué et commence par le joueur suivant celui qui a moins de 2 wagons. Une fois ce dernier tour terminé, il ne reste plus qu'à compter les points afin de déterminer le vainqueur.

3.2.3 Le comptage des points

Les points des joueurs sont donnés dans un tableau , les fonctions de comptage modifient la case d'un joueur pour lui ajouter ou lui enlever des points. Le comptage des points par le serveur se fait en deux temps.

Tout d'abord il calcule les points donnés par la construction des trains. Pour cela il parcourt les rails dans la matrice d'adjacence, et ajoute aux points du joueur se trouvant sur le rail (s'il y en a un) les points correspondant au rail. Rappelons qu'un rail de longueur k donne $1 + \frac{k*(k-1)}{2}$ points.

Ensuite le serveur cherche à savoir si les objectifs de chaque joueur sont réalisés. Il prend un objectif, et fait un parcours de l'arbre constitué de rails du joueur en partant de la ville de départ, s'il existe un chemin menant à la ville de destination, les points de l'objectif sont attribués au joueur, dans le cas contraire ils lui sont retirés.

Pour déterminer le vainqueur, il suffit de trouver le maximum des éléments du tableau de points, le vainqueur est alors le joueur à qui on a attribué ce nombre de points. Cette détermination du vainqueur ne prend pas en compte le cas d'égalité, en effet si deux joueurs ont le même nombre de points le premier sera désigné vainqueur.

4 Le client

Chaque client incarne un joueur différent de la partie, ils ont pour but de gagner la partie de casaniers du rail, ils transmettent donc une suite d'actions au serveur dépendant de leur stratégie et de l'évolution du jeu comme le feraient des joueurs humains.

4.1 Les actions du client

Premièrement, le client doit être capable d'avoir une vision globale du jeu et de son déroulement. Lorsque *init_player* est appelée, le client stocke les informations données, et crée sa propre matrice de jeu pour avoir une vision globale de ce qu'il se passe, et y noter les changements que le serveur lui communiquera à chaque tour de jeu. De plus, il crée aussi une structure joueur qui lui sera propre et qui contiendra à la fois les informations globales sur le jeu (son identifiant; le nombre de villes, de couleurs, d'objectifs, de joueurs; le tableau contenant tous les objectifs, et celui contenant tous les rails), et les informations sur son déroulement (le tableau des objectifs qu'il possède et leur nombre; la tableau des cartes wagons qu'il possède et leur nombre; un tableau signifiant le nombre d'objectifs, de wagons et de cartes wagons restants pour chaque joueur, principalement).

Ensuite, les clients doivent pouvoir échanger au serveur les bonnes informations, c'est à dire les informations nécessaires aux fonctions de l'interface. Lorsqu'ils sont appelés avec la fonction *play_turn*, ils vont tout d'abord actualiser les informations envoyées par le serveur (informations sur le dernier tour de jeu écoulé, et passées en paramètres de la fonction *play_turn*). Ces actualisations se font, pour la plupart, en parcourant les tableaux contenant les informations sur les changements

survenus passés en paramètres de la fonction. Mais pour certaines actualisations, le joueur possède des fonctions propres lui permettant de s'actualiser, notamment une fonction lui permettant de déterminer le nombre d'objectifs qu'il a déjà complétés. Le principe de fonctionnement de ces fonctions auxiliaires est expliqué plus loin, dans la partie "Aspects algorithmiques du jeu".

Ils auront ainsi une vision de l'état du jeu au moment où on les appelle et pourront ainsi l'analyser et réagir selon leur propre stratégie. Celle-ci détermine à quel moment le client pioche des cartes wagon ou objectif et à quel moment il pose ses wagons. Suivant la stratégie adoptée et l'état du jeu, le joueur transmettra au serveur, par le biais du retour de la fonction *play_turn*, l'action qu'il souhaite effectuer et ses caractéristiques (rail sur lequel il veut se poser, couleurs qu'il souhaite piocher, ou fait qu'il veuille piocher de nouveaux objectifs).

Dans le cas où le joueur a demandé au serveur de piocher des objectifs supplémentaires, il sera rappelé par le serveur avec la fonction *choose_objective* qui lui demandera de sélectionner un certain nombre d'objectifs parmis une sélection des objectifs du jeu. Ici encore, le joueur pourra sélectionner un objectif plutôt qu'un autre suivant la nature de sa stratégie.

Enfin, étant donné que le joueur alloue lui-même sa mémoire (pour stocker ses propres informations), il la libère également lui-même lorsque le serveur l'appelle en fin de partie avec la fonction $free\_player$.

4.2 Les différentes IA

4.2.1 Le client aléatoire

Le premier client ayant été implémenté est le client aléatoire, identifié par le nom AI_0 (="Artificial Intelligence $n°0$"). La stratégie adoptée par ce client est d'effectuer absolument tous ses choix de manière aléatoire. Il choisit donc un nombre aléatoire d'objectifs parmis ceux proposés lorsque le serveur l'appelle avec la fonction *choose_objective*, et les objectifs sont eux aussi choisis au hasard. De même, lorsqu'il est appelé avec *play_turn*, il choisit une des trois actions au hasard (poser un rail, piocher une carte wagon, ou une carte objectif) en demandant un rail ou une couleur (selon le cas) choisis aléatoirement.

Ce joueur est ainsi susceptible de tenter de tricher de manière fréquente, car il ne vérifie jamais s'il a les moyens d'effectuer une action avant de la demander au serveur (nombre suffisant de wagons ou de cartes wagon par exemple, ou si le rail demandé est bien libre). Mais, étant donné que le serveur gère les tentatives de triche en passant le tour du joueur lorsqu'il ne peut pas effectuer l'action demandée, alors cela ne gêne tout de même pas le déroulement de la partie.

Cependant, puisque tous les joueurs démarrent avec peu de cartes wagon, alors le temps que le client aléatoire en pioche assez pour pouvoir poser un rail, et le temps qu'il tombe sur un rail qui lui soit accessible compte tenu des actions effectuées par les autres, est assez élevé. La partie risque donc de tourner très longtemps, voire indéfiniment, s'il n'y a que des clients aléatoires qui jouent.

On ne peut ici donc pas à proprement parler "d'intelligence", ni de "stratégie". Et ce type de client n'est également pas très adapté au jeu, surtout s'il est le seul type de joueur à y participer, car il y a un risque que la partie ne se termine pas. Cependant, c'est le type de client le plus simple à implémenter, et permet de pouvoir tester le fonctionnement du serveur.

4.2.2 Le client basé sur l'arbre couvrant de poids minimal

Le second client ayant été implémenté, AI_1, est le client se basant sur le calcul d'un arbre couvrant de poids minimal (le poids d'un rail étant sa longueur).

Explication de la stratégie globale

La stratégie globale de ce client est la suivante:

- Parmis les objectifs qu'il doit piocher en début de partie, il choisit d'en prendre le nombre minimal imposé, et de garder ceux dont le nombre de points est minimal. Le nombre de points étant le plus petit possible, ces objectifs sont donc susceptibles d'être réalisés assez rapidement au cours du jeu. Il va ensuite traiter ces objectifs dans l'ordre.

- Pendant le déroulement du jeu: Tant qu'il n'a pas complété tous ses objectifs, alors il prend le plus petit des objectifs non complétés et il cherche un rail parmis ceux qui sont encore libres et qui pourrait participer à la réalisation de cet objectif. Ceci se fait par l'utilisation de l'arbre couvrant de poids minimal.

- Il vérifie ensuite qu'il a les moyens de placer ce rail: assez de wagons restants, la bonne couleur de cartes wagon et en bon nombre. Si il en a les moyens, alors il demande au serveur de se placer sur ce rail (par le biais du retour de la fonction *play_turn*). Si il n'en a pas les moyens, alors il demande à piocher des cartes wagons, pour être plus en mesure de réaliser ses objectifs dans les tours suivants.

- A partir du moment où il a complété tous ses objectifs, alors il cherche à écouler le plus vite possible tous ses wagons dans le but de finir le plus rapidement possible la partie pour laisser le temps possible à ses adversaires pour remplir leurs objectifs. Pour cela, il va chercher parmis les rails non-occupés celui le plus long. Si il a les moyens de se poser dessus, alors il émet cette requête au serveur; sinon, il demande à piocher des cartes wagons.

Détail de l'utilisation de l'arbre couvrant de poids minimal

Soit un objectif donné et non entièrement complété, le joueur va donc tenter de placer un rail lui permettant d'avancer vers la réalisation de cet objectif. Pour cela, l'utilisation de l'arbre couvrant de poids minimal se déroule comme suit (et que l'on retrouve au début de la fonction $AI\_0\_play\_turn$):

- Il sélectionne d'abord l'ensemble des arrêtes du jeu qui sont soit libres, soit déjà en sa propriété.

- A partir de ces arrêtes, il va construire l'arbre couvrant de poids minimal. Pour ce faire, il suit l'algorithme de $Kruskal$: il commence donc par trier l'ensemble des arrêtes par longueurs croissantes.

- Il les prend ensuite une à une pour les mettre dans l'ensemble des arrêtes de l'arbre couvrant minimal. Pour chaque nouvelle arrête traitée, si celle-ci ne forme pas un cycle avec les arrêtes déjà sélectionnées, alors on la rajoute dans l'ensemble des arrêtes de l'arbre couvrant minimal. Ceci se base sur la définition d'un arbre couvrant de poids minimal, à savoir: un arbre reliant tous les sommets du graphe et dont la somme des arrêtes est la plus petite possible (le graphe considéré ici étant l'ensemble des villes accessibles par des rails libres ou possédés par le joueur lui-même).

- Une fois l'arbre couvrant minimal construit, il s'agit de déterminer s'il existe un chemin dans cet arbre reliant les deux villes de l'objectif. Si oui, il s'agira de déterminer ce chemin; sinon, le joueur considère qu'il ne peut pas placer de rail à ce tour de jeu.

- Pour déterminer s'il existe un chemin reliant les deux villes voulues, il va effectuer un parcours en profondeur du graphe (par le biais des fonctions $rail\_free\_in\_ACM$ et $free\_tab\_rail\_free\_in\_ACM$) où il part d'une ville et parcourt tout l'arbre en notant petit à petit dans un tableau les villes se trouvant sur son chemin direct (sans détour inutile) depuis la ville de départ jusqu'à trouver la ville d'arrivée.
- Une fois ce chemin trouvé, il cherche, parmis les rails concernés, le premier rail qui n'est pas déjà en sa propriété.

Ainsi, ce rail participera à la réalisation de l'objectif, car étant sur le bon chemin; tout en n'étant pas un des plus mauvais choix, car se trouvant dans l'arbre couvrant de poids minimal; même si, comme vu auparavant, ceci n'est pas la stratégie la plus optimale.

5 Aspects algorithmiques du jeu

5.1 Stratégies de parcours adoptées

Dans la plupart des fonctions où l'on cherche à relier deux villes (par exemple les fonctions vérifiant si un objectif est réalisé, ou cherchant quels rails se trouvent entre deux villes que l'on veut relier), se basent sur le principe suivant:

- On crée un tableau de booléens dont chaque case représente une ville.
- On prend la ville de départ (de l'objectif à réaliser, par exemple). Et on parcourt toutes les villes qui sont accessibles à partir de cette ville (selon un critère précis: rail libre, par exemple) en les mettant à 1 dans le tableau précédent lorsqu'elles ont été visitées, et on rappelle récursivement la fonction sur ces villes sans reparcourir celles déjà visitées (à 1 dans le tableau).
- Lorsque l'on arrive au bout de la fonction, si l'on n'a pas atteint la ville d'arrivée que l'on voulait, alors on remonte les villes en les remettant à zéro et en visitant d'autres chemins.
- Si à un moment donné on arrive à la ville que l'on souhaitait, alors l'ensemble des villes sur le parcours de la ville de départ à cette ville sont indexées en 1 dans le tableau de départ.

  Une autre stratégie utilisée, est de faire le même parcours en profondeur que cité ci-dessus, mais en créant deux tableaux (un pour les villes accessibles depuis la ville de départ, et un pour celles accessibles depuis la ville d'arrivée). Les villes qui seront en commun dans les deux tableaux seront donc accessibles depuis l'une et l'autre des deux villes que l'on souhaite relier (selon un certain critère fixé à l'avance: que le joueur soit présent sur chacun des rails, par exemple). On pourra donc déduire de la présence d'une ville dans les deux tableaux qu'un certain joueur a pu aller d'une ville à l'autre d'un objectif donné, et donc qu'il a réalisé l'objectif.

5.2 Complexité des fonctions de parcours

La complexité de ces fonctions de parcours, utilisées à de nombreuses reprises dans le projet est donc linéaire en le nombre de rails, car on effectue un parcours en profondeur, tout en ne reparcourant pas les endroits déjà visités, ni les chemins ayant échoué d'accéder à leur cible.

6 Déroulement du projet

6.1 Tests effectués

Avant de faire marcher le serveur avec nos deux clients implémentés, nous avons mis en place des tests unitaires sur les fonctions principales de l'ensemble de jeu. Ainsi, nous avons des tests sur l'ensemble des fonctions d'initialisation (notamment *test_init*, *test_init_info* ou $test_link$), sur les fonctions du serveur (*test_cards*, *test_play_turn test_winner*) et sur les intelligences artificielles (*test_AI_1* et *test_AI_0*). On vérifie ainsi que les fonctions retournent bien les résultats attendus. Ces fonctions nous ont permis de mettre en évidence des disfonctionnements au niveau de nombreuses fonctions (dépassements de tableaux, problèmes de réallocations, boucles infinies, fonctions récursives retournant des résultats faux, par exemple), que nous avons donc pu régler au fur et à mesure.

6.2 Problèmes rencontrés et freins au bon déroulement du jeu

Cependant, bien que les tests aient permis de mettre en évidence et de régler de nombreux problèmes, nombre de nouveaux problèmes sont arrivés lors de la mise en relation du serveur et des clients.

Après de multiples utilisations de valgrind, nous avons pu identifier beaucoup de petits problèmes immiscés au coeur de certaines fonctions, notamment beaucoup de problèmes de réallocations et de libérations de pointeurs. Cependant, malgré notre acharnement à régler chacun des défauts un à un, nous sommes encore en présence de boucles infinies et de lectures et/ou écritures sur des cases mémoires situées avant ou après le bloc alloué, selon l'exécution. Notre programme tourne donc indéfiniment sur la plupart des exécutions, mais termine en retournant bien un gagnant dans certains cas rares.

7 Conclusion

Pour travailler sur ce projet, nous étions en groupes de quatre personnes. Nous nous sommes donc divisés en deux groupes de deux pour travailler les uns sur le serveur et la lecture du fichier de départ, et les autres sur l'initialisation de la matrice et sur l'implémentation des intelligences artificielles. Lorsque nous étions encore en groupes séparés et que nous avons fait nos tests, nos fonctions semblaient bien fonctionner en elles-mêmes, mais lorsque nous avons mis le serveur en fonctionnement avec les joueurs, de nombreux disfonctionnements et problèmes, notamment des problèmes de mémoire entre les pointeurs que le serveur et les clients se partageaient, sont apparus. Nous avons alors pu appréhender la difficulté de lecture de code, et de debogage, lorsque l'on met notre travail en commun avec d'autres programmeurs. Malgré acharnement, nous n'avons donc cependant pas pu aboutir au bon fonctionnement du jeu. Mais cela nous aura permis de nous familiariser avec les systèmes clients-serveur, d'utiliser et de mieux connaître les outils de debogage (valgrind et gdb), et de développer nos compétences organisationnelles.