POLYTECH SORBONNE

Sorbonne Université
2024-2025

Concours de programmation

Les Aventuriers du Rail

Les Aventuriers du Rail (*Ticket To Ride* ou *T2R*, en anglais) sont un jeu de société créé par Alan Moon en 2004, et édité par Days of Wonder[1]. Le but du jeu est de construire des lignes de chemin de fer en déposant ses wagons sur les cases correspondantes au moyen de cartes *wagons* de couleur, et ainsi de relier des villes entre elles selon des cartes d'objectifs. C'est un jeu de plateau de stratégie qui peut se jouer de 2 à 5 joueurs. Nous y jouerons dans la version à 2 joueurs.

Le but du projet est d'écrire un programme capable de jouer au jeu des Aventuriers du Rail, en respectant les règles. Ce programme pourra jouer contre d'autres programmes (du plus stupide au plus malin), et pourra participer au *Tournoi de programmation* qui aura lieu mi-janvier.

1 Règles du jeu

Le jeu "Les Aventuriers du Rail" se joue sur un plateau du réseau ferroviaire (plusieurs plateaux existent, avec des petites variantes dans leurs règles ; nous jouerons à plusieurs plateaux, dont le plateau USA comme sur la version originale du jeu). Les joueurs (de 2 à 5 joueurs, mais nous ne jouerons ici qu'à 2) ont à disposition 45 wagons chacun et 144 cartes Wagon de 9 couleurs différentes (violet, blanc, bleu, jaune, orange, noir, rouge, vert et des cartes Locomotives pouvant être utilisées pour n'importe quelle carte, comme un joker). Il y a aussi des cartes

[1] `https://www.daysofwonder.com/fr/univers/les-aventuriers-du-rail/`

Destinations qui sont des objectifs à réaliser, en reliant deux villes du plateau par une voie ferrée.

Le but de jeu est d'obtenir le plus de points. Ils peuvent être gagnés soit en revendiquant une route entre deux villes voisines, soit en reliant les deux villes figurant sur une carte Destination, soit en construisant le chemin le plus long.

Au début de la partie, chaque joueur reçoit 4 cartes Wagon et 3 cartes Destination. Il peut garder toutes les 3 cartes Destination ou en choisir 2 et en rendre une. Il y a une pioche de cartes Wagon, ainsi que 5 cartes Wagon face visible.

À chaque tour de jeu, le joueur décide de faire une et une seule des trois actions suivantes :

- Prendre deux cartes Wagons (ou une seule si la carte Wagon choisie est une carte face visible Locomotive); les cartes peuvent être prises dans la pioche ou bien parmi les 5 cartes face visible. Quand une carte face visible est prise, elle est remplacée par une carte de la pioche. Quand 3 cartes visibles sont des cartes Locomotives, on remplace toutes les cartes visibles par de nouvelles cartes visibles prises dans la pioche.

- Prendre trois cartes Destinations et en conserver au moins une ;

- Revendiquer une route entre deux villes. Cela se fait en défaussant autant de cartes Wagon qu'il y a de wagons qui relient les deux villes, et en posant ses wagons sur la route en question. Les cartes doivent être de la même couleur que la route entre les deux villes. Si la route entre les deux villes est en gris, alors n'importe quelle couleur peut être utilisée (mais toutes les cartes doivent être de la même couleur, avec ou sans cartes Locomotive). Si deux routes existent entre deux villes, alors on peut choisir laquelle des deux couleurs utiliser. Enfin, si une route est prise par un joueur, alors il n'est plus possible de revendiquer la route entre ces deux villes (même s'il existe une route double).

Lorsqu'un des joueurs possède 2, 1 ou aucun wagon après avoir joué son tour, alors le dernier tour du jeu a lieu et le jeu se termine. On compte alors les points de la manière suivante :

- Chaque route posée rapporte 1, 2, 4, 7, 10 ou 15 points selon sa longueur de la route (pour 1, 2, 3, 4, 5 ou 6 wagons de longueur, respectivement) ;

- Chaque objectif réalisé rapporte le nombre de points indiqué sur l'objectif. Par contre, si l'objectif n'est pas réalisé, on perd le nombre de points indiqué ;

- Enfin, le joueur qui a réalisé le chemin le plus long remporte 10 points supplémentaires (ou les deux si égalité).

En cas d'égalité de score à la fin, nous avons rajouté les règles supplémentaires (dans cet ordre):

- celui qui a le plus long chemin gagne ;

- si égalité, celui qui a le plus de cartes objectif gagne ;

- si égalité, celui qui a le moins de cartes en main gagne ;

- et si égalité, alors on tire à pile ou face.

Le détail des règles se trouve sur le site du jeu ou bien sur le git du projet :

`https://gitlabsu.sorbonne-universite.fr/hilaire/tickettorideapi`

2 Description de l'API

Tout d'abord, un serveur permettant de gérer les parties (avec la règle du jeu) a été mis en place. Ce serveur (*Coding Game Server*), qui est utilisé tous les ans pour un projet de ce genre, gère les différentes parties avec le calcul des scores et le respect des règles du jeu, l'affichage au format texte de la partie, les différents *bots* qui vous permettront de tester vos programmes, et même l'organisation de tournoi (ainsi qu'un serveur web pour afficher tout un tas d'informations – en cours de réalisation).

Ensuite, une petite librairie vous est donnée, avec les fichiers suivants et ce document[2]:

- `ticketToRide.h` qui contient les types et prototypes des fonctions que vous devez utiliser pour jouer au jeu.
- les fichiers `ticketToRide.c` et `codingGameServer.c` qui contient son implémentation, ainsi que les fonctions internes utiles à la discussion avec le serveur.
- le fichier `json.h` qui contient un parseur json minimal[3].

L'API proposée est susceptible d'évoluer au cours du temps (pas les prototypes, juste les implémentations), ainsi que le serveur, pour rajouter de nouvelles fonctionnalités (l'affichage risque d'évoluer) et bien sûr corriger tous les *bugs* qui pourraient arriver (je préviendrai sur le Discord pour que vous fassiez un `git pull`).

N'hésitez pas à me faire remonter tout bug ou problème par email (`thibault.hilaire@lip6.fr`), en précisant le problème, comment il arrive et en précisant (si possible) le numéro de la partie (les parties sont loggées).

N'oubliez pas de lire les commentaires du fichier `ticketToRide.h`, car il contient tout le détail d'utilisation des fonctions.

2.1 Les types

Quelques types structurés vous sont fournis pour vous aider à communiquer avec le serveur et à jouer. Veuillez vous référer au fichier pour les détails (les commentaires et les noms utilisés sont en anglais[4] ; vous pouvez suivre cette convention ou non pour votre programme).

- `ResultCode` : code de retour de toutes les fonctions. Elles renvoient `ALL_GOOD` si tout se passe bien, et il faut vérifier leur retour.

[2]Repository Git : `https://gitlabsu.sorbonne-universite.fr/hilaire/tickettorideapi`

[3]`https://github.com/zserge/jsmn`

[4]Des fautes d'orthographe doivent trainer ; n'hésitez pas à me les signaler pour que je les corrige !

- `MoveState` : indique si un coup joué est normal (`NORMAL_MOVE`), donc le jeu continue, ou bien si le coup provoque la fin du jeu et qu'on a gagné (`WINNING_MOVE`), perdu (`LOSING_MOVE`) ou illégal (`ILLEGAL_MOVE`), donc perdu.
- `CardColor` : définit les couleurs des cartes Wagons (et les cartes des routes).
- `Action` : définit les coups possibles. Il y a 5 types de coups :
  - *claim a route* (`CLAIM_ROUTE`) : pour revendiquer une route entre deux villes (et poser ses wagons pour se l'approprier)
  - *draw a card* (`DRAW_CARD`) : prendre une carte parmi les cartes faces visibles
  - *draw a blind card* (`DRAW_BLIND_CARD`) : prendre une carte dans la pioche
  - *draw objectives* (`DRAW_OBJECTIVES`) : prendre des cartes Objectif
  - *choose objectives* (`CHOOSE_OBJECTIVES`) : choisir, parmi les cartes Objectif que l'on vient de tirer, le ou les objectifs que l'on veut garder. Cette action (prendre des cartes objectif et en choisir) se fait en deux coups (à jouer successivement) car il y a plusieurs échanges avec le serveur (tout d'abord on récupère les objectifs qu'on nous propose, et ensuite on dit ceux/celui que l'on garde)
- `Objective` : définit une carte Objectif, avec les deux villes de l'objectif et le score associé ;
- `MoveData` : définit un coup et contient les données le définissant. Un coup peut-être parmi les 5 coups possibles définis par `Action` et chacun d'entre eux possède un certain nombre de données le définissant. Grâce au mot-clé `union`[5], un `t_move` peut avoir un champ `claimRoute`, `drawCard` ou `chooseObjectives` qui sont respectivement de type `ClaimRouteMove`, `CardColor` ou `ChooseObjectives`.
  Ces champs contiendront les données pour définir un coup.
- `MoveResult` : contient les données renvoyées par le serveur après qu'un coup est joué. Il contient l'état (de type `MoveState`) pour indiquer si le coup est gagnant ou pas. Ainsi que la couleur de la carte (champ `card`) lorsque que l'on pioche une carte ou encore les objectifs lorsqu'on pioche des cartes objectif. Il contient aussi un potentiel message envoyé par l'adversaire (juste pour s'amuser) et potentiellement un message du serveur en cas d'erreur.

2.2 Fonctions

Vous avez accès aux fonctions suivantes (dont les prototypes se trouvent dans `ticketToRide.h`, ainsi que les détails dans les commentaires) :

- `connectToCGS` et `quitGame` pour se connecter au serveur et fermer la connection; ce sera la première et dernière chose à faire dans le programe.

[5] cf le cours de C, ou bien cette référence, notamment sur les unions anonymes définies dans C11 et utilisés ici : `https://zestedesavoir.com/tutoriels/755/le-langage-c-1/notions-avancees/les-unions/`

- `sendName` et `sendGameSettings` pour envoyer le nom de son bot et attendre que le serveur nous place dans une partie en nous donnant les données du jeu (le plateau, les cartes faces visibles et les 4 cartes qui sont distribuées au début).
- `getMove` permet de récupérer le coup que l'adversaire à jouer (le type de coups et les données qui sont visibles par tous, comme par exemple si une carte face visible a été tirée) ;
- la fonction `sendMove` qui permet de jouer un des cinq coups possibles ;
- `printBoard` permet d'afficher le plateau (l'affichage se fait en mode texte, avec des couleurs; malheureusement, dans un terminal le support des couleurs est limité et dépend même de vos réglages (il n'y a pas d'orange, par exemple; on a ici un jaune plus foncé). Et le rendu devrait être meilleur sur fond blanc.).
- `printCity` permet d'afficher le nom d'une ville à partir d'un numéro.
- `sendMessage` permet d'envoyer, au cours du jeu, des commentaires pour l'adversaire (parfaitement inutile, mais à utiliser en tournoi pour narguer son adversaire !).

3 Premiers programmes

On procède par étapes. On donne ici les indications pour créer pas à pas un programme capable de jouer au jeu des Aventuriers du rail.

3.1 Se connecter

Tout d'abord, on cherche à se connecter au serveur et à créer une partie. La fonction `connectToCGS` permet de se connecter au serveur. Il faut ensuite envoyer le nom de son programme avec `sendName`.
On peut ensuite demander à participer à une partie avec la fonction `sendGameSettings` qui permet de préciser tous les paramètres de la partie. Les différents champs du paramètre d'entrée de type `GameSetting` sont les suivants :

- `gameType` : permet de s'entrainer et contre un bot (`TRAINING`), de faire un match contre un autre adversaire (`MATCH`) ou encore de participer à un tournoi (`TOURNAMENT`).
- `botId` : permet d'indiquer l'id du bot contre lequel on veut jouer : pour le moment `RANDOM_PLAYER` qui joue aléatoirement (pioche des cartes et pose des routes aléatoirement, pas trop loin de ses objectifs, quand il a les cartes pour) et `NICE_BOT` qui est un peu plus malin en cherchant à poser les routes qui sont sur les plus courts chemins de ses objectifs.
- `timeout` : permet de définir le temps maximum (en secondes) pour jouer avant d'être déclaré perdant (par défaut 10s). À modifier (passer à 60) quand vous voulez jouer manuellement.
- `seed` : permet de définir la graine du générateur de nombres aléatoire. En utilisant toujours la même graine, on obtient toujours la même partie (ce qui

est très utile pour débugger). La valeur 0 permet d'avoir une nouvelle partie aléatoire (la *seed* est déterminée en fonction de la date et l'heure où la partie est lancée).

- `starter` : permet de définir qui commence.

Le serveur est accessible à l'adresse `cgs.valentin-lelievre.com` sur les ports `15001`.

Une fois connecté, on peut récupérer les données du plateau dans la structure `getMap`.

On n'oubliera pas de se déconnecter du serveur (fonction `closeConnection`) en fin de programme.

Question 1 *Écrivez un programme qui démarre une partie avec le* bot *DO_NOTHING, récupère les données, et se déconnecte.*

3.2 Boucle de jeu

On peut maintenant avoir une boucle de jeu permettant de jouer, à tour de rôle, avec un *bot* (par exemple `PLAY_RANDOM`).
Il faut donc une boucle où l'on va :

- afficher le plateau et l'état du jeu (utile pour débugger) avec la fonction `printBoard` ;
- si c'est à notre tour de jouer, on va jouer un coup (fonction `PlayMove`). Cinq coups sont possibles, et défini par le type `MoveData`:
  1. Prendre possession d'une route
  2. Prendre une carte de la pioche
  3. Prendre une carte face visible
  4. Prendre des cartes objectifs (en deux coups, d'abord un coup de type `DRAW_OBJECTIVES` puis un de type `CHOOSE_OBJECTIVES`)
- si c'est au tour de l'adversaire, on récupère son coup, avec la fonction `getMove` (dont on ne s'occupera pas dans cette section).

Attention, les deux premiers tours de jeu des deux adversaires doivent être de type `DRAW_OBJECTIVES` et `CHOOSE_OBJECTIVES`.

Les fonctions `getMove` et `sendMove` récupèrent le retour du serveur (les cartes qui ont été tirées, les messages, etc.) dans la variable de type `MoveResult`.

Cette boucle de jeu ne s'arrête que lorsque le champ `state` du `MoveResult` n'est pas `NORMAL_MOVE` (mais `WINNING_MOVE`, `ILLEGAL_MOVE` ou `LOSING_MOVE`).

Question 2 *Complétez le programme précédent en permettant de jouer une partie, jusqu'à sa fin (un joueur perd). Comme coup, on ne fait que tirer deux cartes dans la pioche (dans le même tour de jeu), et on récupère le ou les coups de l'adversaire. On ne s'occupera pas pour le moment du coup de l'adversaire ou des valuers remplies par l'appel à* `sendMove`*.*
Testez et jouez une partie contre le bot *(qui devrait s'arrêter quand il n'y a plus aucune carte de disponible).*

3.3 Structure de données et découpage du code

On voit bien dans notre programme que beaucoup de données concernant le jeu ont été créées (par encore toutes remplies ou utilisées) : le nombre de villes, de routes, les cartes en main, les cartes visibles, le joueur actif, etc.
Nous allons évidemment créer des fonctions pour gérer les différents coups possibles (mettre à jour nos cartes, les cartes visibles, etc.), et il nous faut donc créer les structures de données pour faciliter l'échange entre les fonctions et organiser tout cela.

Une façon (mais cela reste une suggestion) de faire est d'avoir plusieurs types structurés[6], que vous allez remplir au fur et à mesure :

- une structure pour le plateau de jeu, contenant le nombre de villes, le nombre de routes et le tableau de routes (que l'on va traiter en section 3.6.3) ;
- une pour un joueur, contenant le nombre de wagons disponibles, le nombre de cartes en main, le nombre d'objectifs, un tableau de cartes en main (par exemple, un tableau qui indique combien on a de `PURPLE`, de `WHITE`, etc.), un tableau d'objectifs (le type `Objective` a déjà été créé pour vous dans `ticketToRide.h`); on notera que pour ces deux derniers tableaux, on ne sera pas en mesure de le remplir pour le joueur adverse car on ne connaît pas toutes ses cartes (seulement les cartes face visible qu'il peut prendre, et encore). On n'a jamais plus de 20 objectifs dans sa main (en général 3 ou 4, voir jusqu'à une dizaine pour les bons joueurs dans certaines parties chanceuses).
- et enfin une structure pour la partie contenant le numéro du joueur qui est en train de jouer, notre numéro de joueur, les cartes face visible, le plateau de jeu et un tableau de deux joueurs ;

Certains champs vont être remplis dans les sections suivantes.

Question 3 *Une fois ces types structurés créés, modifiez votre programme pour qu'il les utilise. Découpez-le en fonctions (créer la partie, la carte, les joueurs) que l'on remplira au fur et à mesure des étapes suivantes.*
Vous pouvez vous écrire une fonction d'affichage de la partie (cartes en main, objectifs, etc.) pour vérifier que vos structures sont bien remplies.
Vous pouvez aussi commencer à découper votre programme en plusieurs fichiers si nécessaire.

Le but des questions suivantes est de gérer les différents coups possibles (que l'on joue ou que l'adversaire joue) et de mettre à jour les données concernant la partie (cartes, objectifs, routes, etc.).

L'idée est d'avoir les structures de données nécessaires pour connaître l'état du jeu et être capable de calculer les prochains coups à jouer.

3.4 Initiation des données

En début de partie, grâce aux fonctions `sendGameSettings` on récupère les informations nécessaires pour initialiser les mains des joueurs (on débute avec 45 wagons et les 4 cartes du départ et zéro objectif; les objectifs du départ seront pris avec les deux permiers coups joués).

[6] On aurait pu vous proposer ces types structurés, mais cela vous aurait trop guidé et contraint dans vos choix d'implémentation.

Question 4 *Écrire la fonction qui initialise le jeu avec les données de départ.*

3.5 Jouer un coup

Nous allons définir plus tard une fonction qui va décider de quel coup notre joueur va jouer (pour le moment, ce sera une fonction qui demande à l'utilisateur d'indiquer quel coup il veut jouer; ce sera très pratique pour débugger le programe). Suite à cette fonction, il nous faut une fonction qui va indiquer au serveur quel coup nous jouons.

Question 5 *Écrivez une fonction qui demande à l'utilisateur de choisir un coup (et toutes les données nécessaires pour jouer un coup). Il n'y a pas besoin de vérifier les données entrées (ce sera vous l'utilisateur).*

3.6 Mettre à jour les informations du jeu à chaque coup

Puisque nous avons deux joueurs qui vont jouer des coups définis par une variable de type `MoveData` avec un retour du serveur de type `MoveResult`, nous devons maintenant écrire une fonction qui met à jour les données du jeu (cartes de chaque joueur, cartes face visible, objectifs, etc.) en fonction d'un coup joué.

3.6.1 Gérer les objectifs

Nous allons ici gérer les objectifs (par exemple contenu dans le tableau d'objectifs de chaque joueur) et la mise à jour des informations pour les deux coups possibles `DRAW_OBJECTIVES` (qui consiste à récupérer 3 cartes objectif) et `CHOOSE_OBJECTIVES` (qui consiste à dire quels objectifs ont veut garder; au moins 2 pour le 1er tour, au moins 1 pour les fois suivantes).

Question 6 *Complétez votre fonction de mise à jour pour gérer les coups de type `DRAW_OBJECTIVES` et `CHOOSE_OBJECTIVES`. Testez en jouant contre un* bot.

3.6.2 Gérer ses cartes wagons

Nous gérons ici les cartes wagons.

Question 7 *Écrivez une fonction pour ajouter une carte donnée chez un joueur (ne modifie que le tableau de cartes et le nombre de cartes). Idem pour enlever une carte. Vous appelerez ces fonctions quand un joueur prend des cartes ou en dépose.*
Si ce n'est pas encore fait, initialisez les cartes de votre joueur (`sendGameSetting` permet de récupérer un tableau de 4 cartes, il n'y a qu'à utiliser votre fonction d'ajout de carte pour chacune).
Mettez à jour les cartes de l'adversaire quand celui-ci pioche des cartes (`DRAW_BLIND_CARD` et `DRAW_CARD`), ou quand il place une route (`CLAIM_ROUTE`).
Testez en jouant contre un bot.

Pensez aussi qu'un joueur (vous ou l'adversaire) doit rejouer si il a

- tiré une 1ère carte dans la pioche ;
- ou tiré une 1ère carte face visible et que ce n'est pas une locomotive ;
- ou tiré des objectifs.

3.6.3 Gérer les routes entre les villes

On s'occupe ici des données concernant la carte, c'est-à-dire la liste (que l'on traitera comme un tableau, vu que la taille nous est donnée par `GameData`) des routes entre les villes.
Les données nous sont fournies (cf le commentaire de la fonction `sendGameSettings`) par 5 entiers pour chaque route (identifiant des deux villes, longueur en wagon de la route, couleurs de la route). Il y a plein de façons de stocker ces données, et dépendent de l'utilisation qui va en être faite. Très probablement, vous aurez besoin de connaître si une route entre deux villes existent et quelles sont ses caractéristiques. Une façon (pas forcément la meilleure en terme d'utilisation mémoire, mais la plus facile à mettre en œuvre) est de créer un type pour une route (longueur, couleurs, savoir si elle est libre ou non, etc.), et de créer un tableau à deux dimensions de routes (ou plutôt de pointeurs vers une route), les deux entrées étant l'identifiant des villes. Vous pouvez faire un tableau statique d'une taille suffisamment grande, ou si vous vous sentez le courage, un tableau 2D alloué dynamiquement. Si le tableau est `T`, alors `T[city1][city2]` vous donne la route entre `city1` et `city2`, si elle existe (et vous convenez d'un moyen d'indiquer que la route n'existe pas, comme par exemple le pointeur `NULL` si vous avez utilisé un tableau de pointeurs de routes).

Question 8 *Créez votre propre structure de données pour les routes. Initialisez vos routes avec le tableau donné par `sendGameSettings`.*
Mettez à jour vos données quand une route est jouée (on note la route comme appartenant à un joueur, on diminue le nombre de cartes du joueur, on diminue le nombre de wagons).
Testez contre un joueur.

4 Écrire un programme qui joue seul (et à peu près intelligemment)

Maintenant que vous avez toute la structure de données, et qu'elle est à jour à chaque coup, vous allez pouvoir remplacer votre fonction qui demande à l'utilisateur de rentrer au clavier un coup, par une fonction qui va trouver un coup à jouer pour remplir les objectifs. Les explications qui suivent ne sont que des indications pour ceux n'ont aucune idée de comment leur programme doit trouver quel coup jouer. Ce n'est *qu'une* façon de faire, pas la seule, et pas forcément la meilleure. Cette section contient des suggestions ou des idées à mettre en œuvre pour un tel programme.

4.1 Stratégie de jeu

Pour cela, il faut avoir déjà fait une ou deux parties pour voir quelle peut être une *stratégie* de jeu. Quand un joueur joue, ce qu'il fait le plus souvent, c'est de planifier ses coups. Il regarde ses objectifs, prévoit une route pour relier ses villes (généralement la plus courte, ou bien celle qui amène le plus de points, ou celle qui sera la plus rapide à poser[7]) et donc garder en tête une liste de routes que

[7] Il ne faut pas oublier que dans le meilleur des cas, il faut un tour pour récupérer deux cartes, plus un tour pour poser les wagons. Donc si vous avez le choix entre une route à six wagons et trois routes à deux wagons, il vous faut quatre tours dans le 1er cas, et six tours dans le deuxième (à condition d'avoir les bonnes cartes...).

l'on souhaiterait prendre. Ensuite, à chaque tour de jeu, on prend les cartes (face visible ou dans la pioche) en fonction des couleurs dont on a besoin. Par exemple, si on a comme objectif de relier Seattle à Toronto, on peut se dire qu'on veut faire Seattle→Helena, Helena→Duluth et Duluth→Toronto, et donc dans les tours suivants prendre tous les cartes jaunes, oranges et violettes que l'on peut. Et dès qu'on a les cartes pour poser un tronçon, on le pose[8]

Donc l'algorithme pour jouer au jeu se décompose en trois parties :

1. On établie la liste des routes que l'on veut prendre ;
2. On regarde si on a fini ses objectifs. Si oui, on en prendre de nouveaux[9] ;
3. Si on peut poser une des routes parce qu'on a le bon nombre de cartes, on la pose.
   Et sinon, si une des couleurs qui nous intéresse est face visible, on la prend.
   Et sinon, on pioche.

On peut aussi avoir envie, après avoir lister les routes à prendre, de les trier dans un ordre particulier, du plus important au moins important (du plus pressé à prendre au moins pressé).

4.2 Trouver le chemin le plus court entre deux villes

L'étape 1 (ainsi que la vérification si les objectifs sont atteint) passe par trouver le chemin le plus court entre deux villes. Pour cela, nous avons à notre disposition plusieurs algorithmes que l'on peut utiliser. Le plus simple est l'algorithme du plus court chemin de Dijkstra[10]. Il trouve le plus court chemin, dans un graphe, entre un nœud et tous les autres nœuds. Dans notre cas, il trouvera le plus court (ou celui qui amène le plus de points, ou le plus rapide) chemin entre deux villes.

L'algorithme est le suivant:

[8] Évidemment, un joueur expérimenté va aussi se dire qu'il peut faire Seattle→ en passant par Calgary et Winnipeg si finalement il a pioché beaucoup de cartes blanches, ou par Omaha si il a beaucoup de cartes rouges... Mais je laisse les plus avancés programmer cette fonctionnalité.

[9] Sauf si on est tout près de la fin de la partie, c'est-à-dire qu'il reste peu de wagons à un des deux joueurs, auquel cas on choisit des routes à poser au hasard parmi celles qu'on peut poser facilement avec nos cartes

[10] `https://fr.wikipedia.org/wiki/Algorithme_de_Dijkstra`

Algorithme 1 : Algorithme de Dijkstra

input : l'indice (entier) `src` de la ville de départ
input : un tableau `G` de $N \times N$ entiers représentant le graphe (`G[i][j]` donne la distance entre les villes `i` et `j`, vaut $+\infty$ si non reliées)
Data : un tableau `Visité` de N booléens : `Visité[i]` indique si la ville d'indice `i` a été visitée par l'algorithme
Data : un indice (entier) `u` qui indiquera la ville considérée dans la boucle
output : un tableau `D` de N entiers: `D[i]` indique la plus courte distance entre les villes `src` et `i`
output : un tableau `Prec` de N entiers : `Prec[i]` indique de quelle ville il faut venir pour aller de `i`

```
1  à src // Initialisation
2  for i from 0 to N − 1 do
3  |  D[i] ← +∞
4  |  visité[i] ← false
5  end
   // On part de la source
6  D[src] ← 0
   // On cherche le plus court chemin pour aller de src à
      chaque ville
7  repeat N − 1 times
   |  // On trouve la ville non visité la plus proche
8  |  u ← distanceMini(D, visité)
   |  // On marque cette ville comme visitée
9  |  Visité[u] ← true
   |  // On met à jour chaque ville reliée à c
10 |  for v from 0 to N − 1 do
   |  |  // On ne met à jour que les villes non visités, où il
   |  |     y a un chemin entre u et v et où il est plus court
   |  |     d'arriver à v en passant par u
11 |  |  if Visité[v]==false and G[u][v]≠ +∞
12 |  |    and D[u] + G[u][v] < D[v] then
13 |  |  |  D[v] ← D[u] + G[u][v]
14 |  |  |  Prec[v] ← u
15 |  |  end
16 |  end
17 |  return D et Prec
18 end
```

La fonction `distanceMini` trouve l'indice de la ville non encore visitée qui est la plus proche. Son algorithme est le suivant:

Algorithme 2 : Fonction intermédiaire `distanceMini`

input : un tableau `D` de N entiers: `D[i]` indique les distances entre les villes `src` et `i`

input : un tableau `Visité` de N entiers: `Visité[i]` indique si la ville d'indice `i` a été visitée par l'algorithme

Data : `min` et `indice_min` (entiers) pour la valeur du minimum et son indice

output : l'indice de la ville non encore visitée la plus proche

```
   // Initialisation
1  min ← +∞
   // Recherche de la ville non visitée la plus proche
2  for i from 0 to N − 1 do
3  |  if Visité[i]==false and D[i]<min then
4  |  |  min ← D[i]
5  |  |  indice_min ← i
6  |  end
7  end
8  return indice_min
```

Le fonctionnement de l'algorithme de Dijkstra sera détaillé prochaine dans une vidéo.

Il peut être facilement utilisé dans le projet car on dispose déjà quasiment du tableau `G` (très probablement que dans votre cas, il s'agit d'un tableau de pointeurs de routes, donc il est facile d'en récupérer la distance de la route). De même, là où on teste `G[u][v]`$\neq +\infty$ pour savoir si la route entre `u` et `v` existe, il faut dans notre cas vérifier si la route existe et si elle n'est pas déjà prise par l'adversaire. Dans le cas où elle est prise par nous, on peut considérer que la distance est égale à 0 (on a déjà posé le chemin!).

Dans notre cas où seul le plus court chemin entre deux villes nous intéresse, on peut transformer la boucle ligne 7 de l'algorithme de Dijkstra en une boule **While u** $\neq$ **dest** où `dest` est l'indice de la ville de destination, et où `u` a été initialisé à la ville de départ.

4.3 Déduire la liste des routes à prendre

Une fois l'algorithme de Dijkstra a été utilisé, il est possible de retrouver le chemin pour aller d'une ville à la ville de départ en utilisant le tableau `Prec` renvoyé par cet algorithme. `Prec[i]` nous indique de quelle ville il faut venir pour prendre le chemin le plus court. Il suffit donc de remonter le chemin, c'est-à-dire que pour aller de `src` à `dest`, il faut partir de `dest` puis aller à `Prec[dest]` (son prédécesseur sur le chemin), puis à `Prec[Prec[dest]]`, etc. et ce jusqu'à arriver à `src`.

Si on voulait afficher le chemin, on pourrait utiliser l'algorithme suivant

Algorithme 3 : Afficher le chemin de `dest` à `src`

```
   input  : l'indice de départ src
   input  : l'indice de l'arrivée dest
   // On démarre à la fin
1  v ← dest // On remonte le chemin jusqu'à arriver au départ
2  while v ≠ src do
   |  // Afficher le chemin qui va de Prec[v] à v
3  |  v ← Prec[v]
4  end
```

Vous n'avez plus qu'à adapter cet algorithme pour avoir les routes qui vous intéressent entre les deux villes de votre objectif (évidemment, on écartera les villes qui sont déjà en votre possession). Le plus simple est de les stocker dans un tableau (suffisamment grand), en gardant dans une variable le nombre d'éléments. Cela fera fonction de "liste" et sera plus simple à utiliser qu'une liste chainée vue en cours.

4.4 Choisir le coup à jouer

Une fois que l'on a la liste des routes que l'on souhaite prendre, il y a plusieurs façons de procéder. Vous pouvez par exemple passer en revue toutes vos routes à avoir et voir si vous avez les cartes suffisantes pour les jouer (de préférence en les parcourant par ordre de priorité, peut-être de la plus grande à la plus petite, pour poser en 1er les routes les plus grandes?). Puis si vous n'avez de quoi poser aucune route, alors, vous pouvez ré-itérer sur chacune de vos routes et voir si les couleurs qui vous manquent sont dans les cartes face visible. Si oui, vous pouvez la prendre. Et si non, il reste la pioche.

Enfin, grâce au plus court chemin entre deux villes, il vous est possible de savoir si vous avez réussi un objectif ou non (si la distance entre deux villes est zéro – puisque vous avez considéré les routes vous appartenant comme des routes de longueur 0 – alors elles sont reliées par vos routes, et l'objectif est atteint). Si tous vos objectifs sont atteints, alors il est temps de prendre de nouveaux objectifs (grâce à l'algorithme du plus court chemin, vous saurez quelle distance il vous reste à parcourir pour finir chaque objectif; gardez ceux qui sont atteignables avec une distance inférieure à votre nombre de wagons restants !)

5 Et ensuite...?

Voilà, avec tous ses éléments, vous devriez être capable d'avoir un joueur qui peut gagner facilement sur `PLAY_RANDOM`. Alors il sera le temps de vous attaquer à un autre bot (`NICE_BOT`, qui joue à peu près comme décrit ici) ou aux programmes de vos camarades !

Vous pouvez penser à modifier l'algorithme du plus court chemin, et modifiant la notion de *distance* (vous ne voudrez peut-être pas le chemin le plus court, mais le chemin qui rapporte le plus de points ou qui se joue en le moins de coups). Vous pouvez aussi ordonner vos coups différemment (certains tronçons comme Seattle→Portland sont stratégiques quand on joue à deux sur la carte USA), tenter d'avoir le plus d'objectifs possibles ou bien encore de terminer le 1er la partie pour faire perdre l'adversaire... Il y a beaucoup de possibilités à tester...
Qui saura relever le défi et battre tous ses adversaires ??

À vous de ~~jouer~~ programmer !!!

Et n'hésitez pas à aller sur les salons Discord associés pour poser vos questions et avoir de l'aide si nécessaire !