Explication du code

Projet TransDisciplinaire

IA Aventuriers du Rail - 05/2019

Rédacteurs : Guillaume Grosse & Lorraine Vannel

Equipe Projet : A.De Barros - H.Fournier - G.Grosse - C.Hugot - M.Marlier - L.Vanel

Client-tuteur : Victorien Marchand

Sommaire

1. Modélisation

A. Structure

Nous avons cherché à modéliser les différents éléments du jeu des Aventuriers du Rail. Pour cela, nous avons utilisé des notions de théories des graphes pour faciliter la manipulation des objets créés. Nous avons donc créé **les classes** ci-dessous :

- **Le Plateau** est une classe stocke également toutes les informations liées au plateau de jeu, telles la liste des villes, les cartes destination, les tronçons, ...

  Elle comporte également un **graphe** des tronçons représenté par une matrice 36x36 qui au coefficient (i,j) prend la valeur du poids de l'arête entre les villes i et j si un chemin les relie sur le plateau et 0 sinon. Ces coefficients sont voués à changer au cours de la partie.

- **Les Villes** du plateau (les **sommets**) sont associées à un entier qui les repère de façon unique.

- **Les Tronçons** (les arêtes) correspondent au chemin reliant deux villes adjacentes, ils possède une longueur, une couleur, un poids, ...

- **Les Types de Trajets** sont définis par l'expert qui distingue les destinations grâce à des repères géographiques, par exemple Horizontale, Verticale Est, etc.

- **Les Itinéraire** est un ensemble de tronçons qui relie des villes entre elles dans l'ordre dans lequel elles sont visitées. Il est notamment caractérisé par sa longueur, son poids et son nombre tronçons.

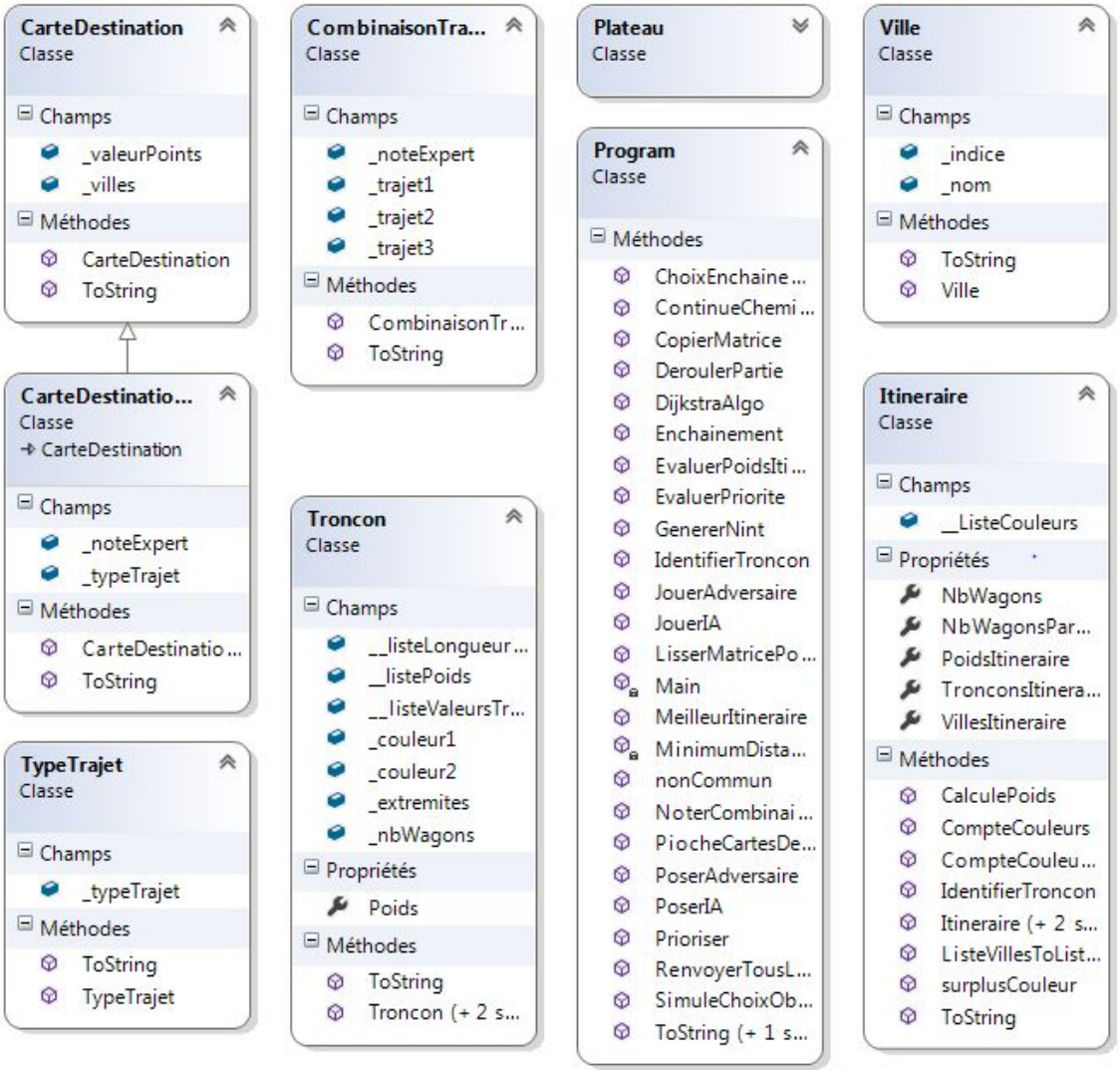
- **Les Cartes Destination** correspondent en tous points à celles de la pioche : on leur associe deux villes et un nombre de points.

- **CarteDestinationExpert** est une classe fille de Carte Destination et prend en compte les conseils de l'expert. Elle attribue à une Carte Destination un Type de Trajet et une note attribuée lors des entretiens.

B. Diagramme des Classes utilisées dans notre algorithme

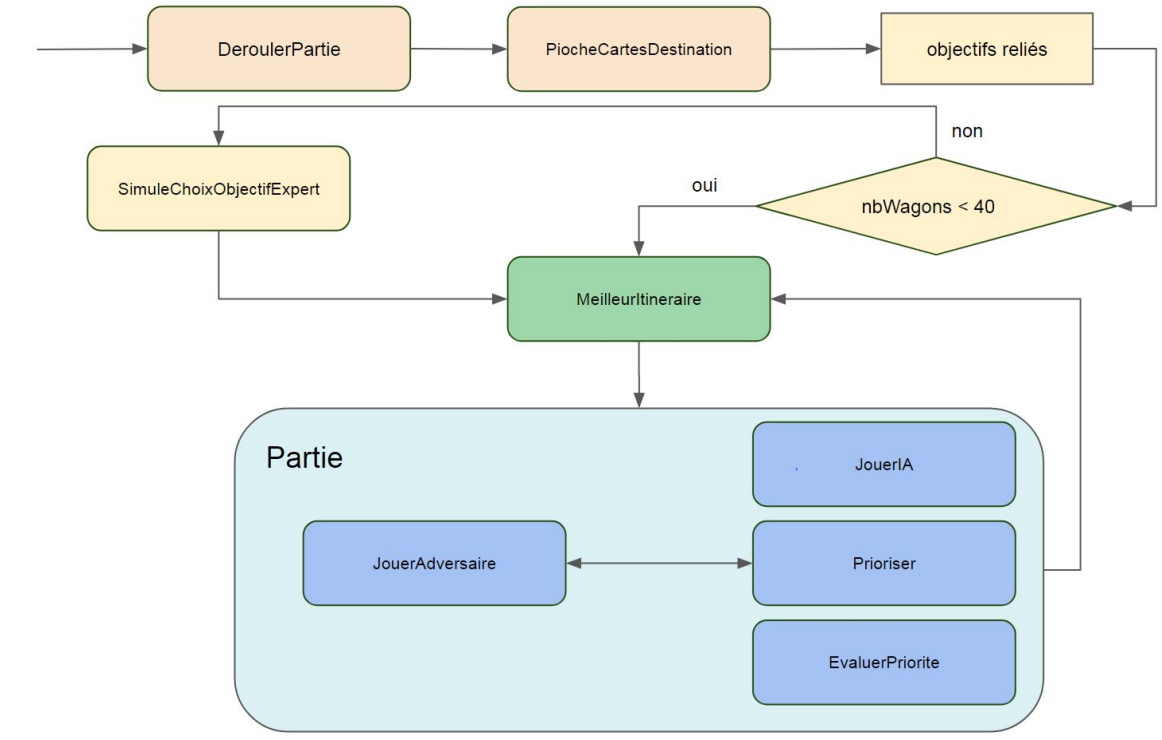
En particulier, cette représentation montre qu'il s'agit d'un **code de recherche** : un grand nombre de méthodes de classes sont implémentées dans le 'Program' car les classes ne sont survenues que plus tard.

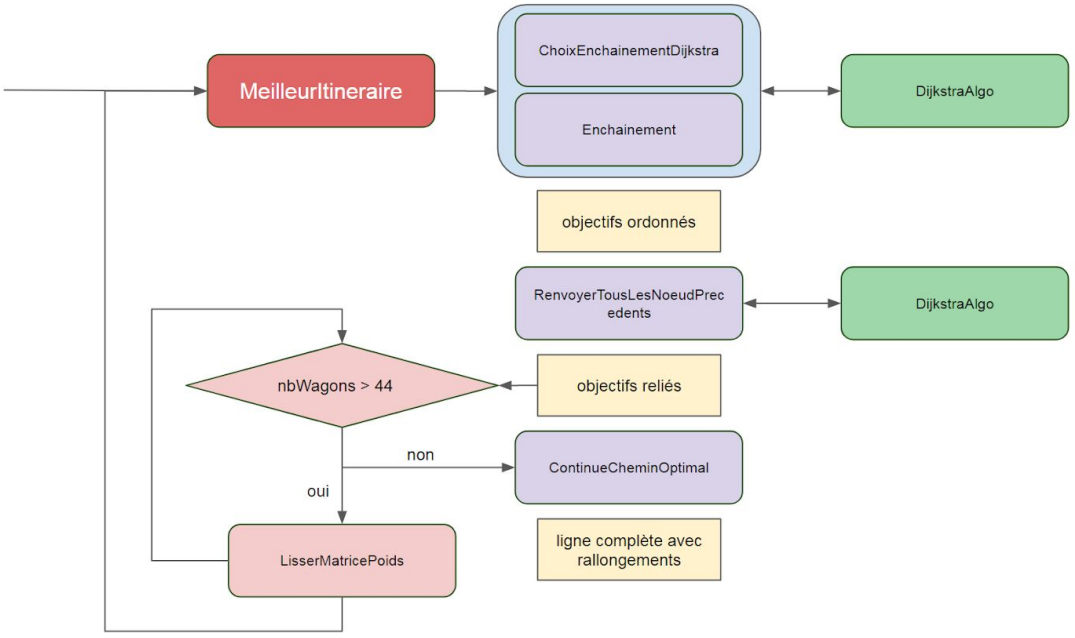
Cependant, un remaniement de ces différentes fonctions pourrait être réalisé en cas de poursuite du projet !



B. Fonctions utilisées

Le détail des fonctions schématisées ci-dessous est donné en annexe.





C. Règles utilisées

Cette initiative de programmer l'algorithme s'inspire directement du travail effectué sur les règles recueillies lors des entretiens avec le joueur expert. Bien que nous n'ayons pas pu tout incorporer dans le code, nous avons essayé de prendre en compte les règles les plus importantes.

Règles implémentées - Règles partiellement implémentées - Règles non implémentées

Règle	Explications
Règles générales	
Faire la ligne la plus longue	L'algorithme cherche à tout prix à construire une ligne continue en décourageant fortement une double utilisation du même tronçon
Capturer le plus de lignes de 6 possibles	Les lignes de 6 sont fortement privilégiées lors des calculs d'itinéraire grâce à leur faible poids et lors de la pose des tronçons
Capturer le plus de grandes lignes	Même principe, les longues lignes sont privilégiées car leur poids est plus faible.
Compléter les objectifs	Le chemin donné par l'algorithme relie absolument toutes les villes-objectif.
Ne pas piocher de locomotives face visible	Phase de pioche non implémentée
Ne pas commencer à relier ses objectifs avec un train continu.	Les wagons sont posés dans un ordre de priorité qui n'est pas linéaire.
Ne pas bloquer avec des tronçons de 1 en début de partie	Non implémenté : la notion de blocage efficient est très complexe
Ne pas commencer à poser sans avoir les cartes	Phase de pioche non implémentée
PHASE 1 : Choix des objectifs	
Utilise l'algorithme expert de choix des objectifs, de calcul des chemins et tableaux de classement des destinations	

PHASE 2 : Phase de pioche	
A chaque tour, piocher 2 cartes face cachée	Pioche non implémentée
PHASE 3 : Phase des 10 cartes et pose	
Pose les lignes de 6 d'abord.	L'ordre de priorité, tel qu'il est calculé (cf. explication de la fonction **prioriser**) privilégie les lignes de 6.
La prioritisation des tronçons (et donc leur ordre de pose) dépend de leur valeur dans l'itinéraire : s'ils sont indispensables (et que sans eux le trajet s'empire beaucoup) il sera prioritaire.	Priorisation calculées par la fonction **prioriser** qui les ordonne pour pouvoir créer un ordre de pose optimal.
Les tronçons manquants pour l'itinéraire prennent aussi de l'importance.	Adapté lors de la partie
Continuer à piocher face cachée jusqu'à 40 cartes sauf si : 1/ l'adversaire commence à poser les routes de votre chemin prévu dans ce cas, il faut poser et sécuriser les routes les plus importantes (routes de 6 dans le chemin et/ou routes qui ont pas d'alternative) 2/ il vous manquera une certaine couleur pour compléter le chemin prévu dans ce cas, piocher face visible les couleurs, en sachant que les couleurs les plus importantes sont dans l'ordre : le vert, le orange, le noir, le blanc et le jaune.	Pas de gestion de priorité entre pioche et pose, instructions assez peu claires et difficiles à implémenter
Ensuite, poser les lignes, en priorité : 1/ les lignes de 6 dans le chemin prévu - qui connecte à une ville importante - qui ne nécessite pas de locomotive - qui nécessite une locomotive - qui nécessite plusieurs locomotives 2/ les lignes de 5 du chemin prévu	Les lignes les plus longues ont un poids faible et sont donc privilégiées dans la pose (leur perte représente un gros détour) Les lignes de 6 sont encore plus favorisées grâce à un rehaussement de leur poids La notion de ville importante n'est pas implémentée

PHASE 4 : Fin de partie	
Allonger le chemin	Une fois les objectifs finis, la fonction **ContinueCheminOptimal** permet de proposer d'autre lignes optimales jusqu'à ce qu'il ne reste qu'un seul wagon.
Bloquer l'adversaire en rattachant à son train	Non implémenté : la notion de blocage efficient est très complexe
Bloquer l'adversaire	Non implémenté : la notion de blocage efficient est très complexe
Piocher face visible ce qu'il manque pour effectuer les autres actions	Pioche non implémentée

D. Démarche

Ce projet n'avait pas pour objectif initial d'implémenter le code en dur mais de fournir un pseudo code pouvant servir de base à un futur projet.

Cependant, après avoir implémenté dans un premier temps un **choix des objectifs** qui s'est avéré très satisfaisant et souhaitant continuer sur cette piste, nous avons décidé de travailler d'abord sur une modélisation du plateau en graphe (graphe, sommets, arêtes).

Avec cette modélisation, nous avons pu commencer à tester des **algorithmes de parcours de graphes** en vue d'un calcul d'itinéraire et le Dijkstra s'est avéré être assez intuitif et facile d'utilisation : chercher le plus court chemin (au sens du poids des arêtes) pour relier 2 villes.

Nous avons transposé cette méthode à la **construction d'itinéraires plus complexes** passant par toutes les ville-objectif piochées (voir plus loin) et prolongeables par de nouvelles fonctions.

Une fois que nous avons eu accès au chemin optimal réalisant tous les objectifs piochés, nous avons pu donner un **ordre de priorité** à ces chemins pour les poser par ordre de priorité.

2. Déroulement en langage naturel

A. Faire un choix d'objectifs

En début de partie, il faut renseigner quels objectifs sont piochés. On utilise alors plusieurs critères pour discriminer la **meilleure combinaison possible**.

D'abord, on utilise un **critère heuristique** qui permet de pallier au fait que nous n'avons jamais pris en compte les combinaisons de 3 types de trajets avec l'expert.

On cherche si, en imposant une **forte contrainte d'utilisation des tronçons de 6**, l'itinéraire calculé par notre programme (traité plus loin) et reliant les villes des 3 cartes destination piochées, compte **40 wagons ou moins**. Si tel est le cas, on considère *de façon heuristique* que le trajet est réalisable avec les 3 cartes.

En effet, avec la contrainte sur les lignes de 6, le trajet est généralement **beaucoup plus long que nécessaire** avec de nombreux contournements. Autrement dit, si les 3 cartes sont retenues, il est assez **facile de dévier de l'itinéraire optimal** en cas de blocage par l'adversaire.

Ensuite, si ce premier calcul ne donne pas de résultat satisfaisant, on détermine la meilleure combinaison d'objectifs avec les **critères expert**, en fonction du ***TypeTrajet*** et des ***CarteDestinationExpert*** (ou **objectifs**) qui figurent dans les résultats des entretiens avec l'expert.

La note d'une combinaison est égale à la somme des notes de chaque ***CarteDestinationExpert*** et de celle de la combinaison de ***TypeTrajet*** . Dans de nombreux cas, un maximum est discriminé. Dans les cas limites d'égalités, nous avons choisi d'appliquer le **critère arbitraire du maximum de points d'objectifs.**

Exemple : Entre une combinaison d'une valeur de 25 points et une autre d'une valeur de 18 points, nous retenons celle de 25.

Nous avons ajouté plusieurs cas particuliers suite aux entretiens avec l'expert : ainsi, on retiendra systématiquement toute combinaison de 3 objectifs de *TypeTrajet* ***Vertical*** et ***Vertical Ouest***, ainsi que toute triplette de *TypeTrajet* ***Vertical Est***.

Récapitulatif du choix d'objectifs :

- si calcul itinéraire <= 40 wagons : on retient les 3 cartes
- cas particulier sur TypeTrajet : on retient les 3 cartes
- maximum déterminé avec critère expert : on retient la meilleure combinaison
- critère expert insuffisant : on retient la combinaison ayant le plus de points

B. Proposer un itinéraire optimal au début de partie

A ce stade nous avons effectué un choix d'objectifs, soit 3 à 6 ville-objectif, il reste à savoir quelles villes raccorder et dans quel ordre !

Nous avons pour cela implémenté un **algorithme de Dijkstra** qui, à partir des données du plateau (distances, noeuds, …) et de deux villes, fournit la distance entre ces deux villes et le dernier noeud du **chemin optimal reliant ces deux villes**. On procède ensuite par récurrence pour reconstruire la liste des Villes du chemin.

Il convient de noter que cet algorithme prend seulement en compte **un poids d'arête**, ou distance, mais que ce poids peut tout à fait être négatif ! A chaque tronçon est donc **associée une distance arithmétique** (on parlera sans distinction de poids et de distance par la suite).

Il s'agit de fixer pour chaque tronçon la **distance qui reflète le mieux son importance**, car le nombre de points rapportés par un tronçon n'est pas proportionnel à la longueur de celui-ci.

$$Poids\ du\ tronçon = \frac{1000}{NbWagons + PointsTronçon}$$

Nous avons choisi de représenter chaque poids sous la forme d'un entier, car c'est une contrainte que nous ont imposé les sorties de nombreuses fonctions déjà implémentées au moment de la création de la matrice des poids. L'ensemble des poids des tronçons est représenté dans la ***MatricePoids*** de 36*36 (voir plus haut).

Ensuite, on calcule le **meilleur chemin entre chaque ville objectif** avec le Dijkstra : on obtient une matrice de distances n*n avec n le nombre de villes. On procède comme suit pour savoir quelles villes objectif relier et dans quel ordre :

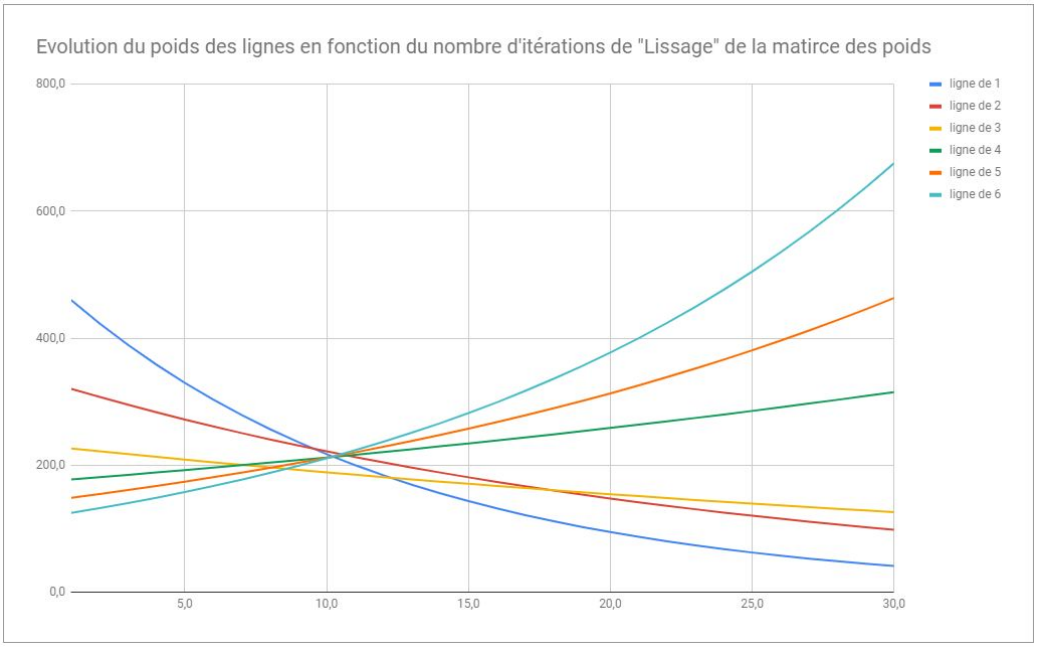
- on définit une ville objectif comme étant la ville de départ,
- on va de **proche en proche** jusqu'à la dernière ville (sans jamais revenir sur une ville visitée) et on retient la distance associée,
- on compare la distance obtenue pour chaque ville de départ,
- on retient l'**enchaînement de ville-objectif** qui a le poids le plus faible.

Exemple : Pour le cas des ville-objectif A, B, C et D, le chemin de plus court déterminé pourrait être C -> D -> B -> A. Attention, les villes intermédiaires ne sont pas encore déterminées, cette suite ne constitue pas un itinéraire réel mais plutôt un rendu à vol d'oiseau !

Effectivement, on doit ensuite relier toutes les ville-objectif en **déterminant les villes intermédiaires**. On utilise pour cela le **Dijkstra de façon récursive** sur chaque paire de ville-objectif à relier.

Illustration : On souhaite relier C à D. On exécute le Dijkstra sur (C,D), qui nous apprend que l'avant dernière ville reliant C&D est ville 1. On applique ensuite la fonction sur (C,ville 1), et ce récursivement jusqu'à ce que ville k = C.

Si l'itinéraire calculé est trop long, on procède à un **lissage de la matrice des poids** pour utiliser des lignes moins longues et **réduire la taille de l'itinéraire**. Les poids les plus extrêmes sont fortement impactés, le poids médians le sont beaucoup moins.



Graphe illustrant l'évolution du poids des différents tronçons en fonction de leur taille

Une fois que cet itinéraire, qui ne relie que les villes des Cartes Destinations choisies, est déterminé, on peut construire un élément itinéraire et travailler indifféremment sur les villes ou les tronçons. Le programme va ensuite chercher à **prolonger cet itinéraire** avec les lignes les plus longues possible.

On s'intéresse aux deux extrémités de la liste ordonnée des tronçons de l'Itinéraire et on considère les tronçons non encore visités qui y sont reliés.

On détermine ensuite quel tronçon est le **meilleur prolongement direct**, vérifie qu'il reste suffisamment de wagons disponibles et dans ce cas, l'ajoute à l'Itinéraire en tant que nouvelle extrémité. Ce processus continue tant qu'il reste au moins un wagon non posé.

Nous avons donc un **Itinéraire continu**, **satisfaisant tous les objectifs** et qui utilise **le plus de wagons possible**.

Récapitulatif :

- **on détermine le meilleur enchaînement de ville-objectif**
- **on détermine les villes intermédiaires entre les ville-objectif de l'enchaînement**
- **on prolonge l'itinéraire déterminé avec les tronçons les plus longs possibles**

C. Jouer et s'adapter

La pioche n'est pas implémentée dans notre solution, par conséquent l'option proposée "piocher des cartes" s'apparente plutôt à un "je donne la main au joueur suivant".

A chaque tour de jeu, on met à jour la ***MatricePoids*** pour rendre compte de l'état du plateau après chaque pose : les poids passent à une **valeur très élevée** lorsque l'adversaire capture un tronçon, et à une **valeur très faible** lorsque l'IA fait de même.

Lorsque c'est son tour, l'IA **calcule un itinéraire optimal** pour le plateau courant comme présenté dans le C.

Ensuite, elle réalise un classement des tronçons par **ordre de priorité** et les **pose** par ordre d'importance décroissante. La priorité est définie par la différence de poids entre le chemin optimal à l'instant t et le chemin alternatif ne passant pas par ce tronçon.

En outre, on suit la règle expert annonçant de **prioriser la pose des lignes de 6** en ajoutant 200 au poids de celles-ci.

C'est à cette phase de jeu que certains problèmes de calcul d'itinéraire surviennent…

Récapitulatif

- **on met à jour la matrice des poids de tronçon à chaque tour de pose**
- **on calcule l'itinéraire optimal à l'instant t**
- **on évalue l'ordre de priorité de pose des tronçons**
- **on pose le tronçon de priorité la plus élevée**

3. Critique de la solution

A. Limites

Notre code reflète un peu l'**aspect expérimental** de notre projet.

De plus, notre code ne respecte pas toutes les règles du système expert car certaines sont très **difficilement implémentable**, ce qui nous a poussé à **compléter le code par nos propres résultats et raisonnements**. Cela ne représente pas une limite du code mais plutôt une limite du respect du système expert.

D'un point de vue technique, notre algorithme a tendance à suivre des **règles absolues** qui peuvent porter préjudice à l'efficacité de la solution. En effet, il est obligé de faire un **ligne continue** qui relie absolument toutes les ville-objectif. Ces règles, qui sont un avantage dans la plupart des cas, s'opposent dans certains cas à ce qu'aurait fait l'expert.

B. Perspectives

Ce projet offre un grand nombre de perspectives intéressantes, listées dans la partie **C.Liste des fonctionnalités implémentables.**

Parmi ces perspectives, les plus immédiatement implémentables sont la **gestion des couleurs**, l'adaptation du **choix d'objectifs** pour les cas limites et l'**affinement du calcul d'itinéraire.**

C. Liste de fonctionnalités implémentables

Pioche

- implémenter la ***pioche***
- adapter la ***pioche des dernières cartes*** au meilleur itinéraire envisagé

Choix d'Objectifs

- ne choisir de ***réaliser qu'un seul objectif*** si l'itinéraire rapporte plus de points que n'importe quelle autre combinaison de cartes objectifs (en adaptant la seconde carte à retenir)

Couleurs

- introduire des ***contraintes sur les couleurs*** utilisées pour le calcul d'itinéraire
- produire l'itinéraire optimal en fonction d'une ***main de cartes déterminée*** et d'un plateau donné

Gestion des Priorités

- gérer la priorité entre ***pose et pioche***
- prendre en compte les ***tronçons posés de l'adversaire*** dans la pose

Calcul d'Itinéraire

- adapter l'itinéraire aux ***contraintes de couleurs***
- implémenter une recherche plus efficiente de ***prolongements***
- adaptation du chemin flexible mais utilisant au mieux les ***tronçons déjà posés***

Blocage & Prediction

- détecter un ***blocage potentiel facile*** (induisant une déviation peu coûteuse)
- détecter un ***blocage probablement critique*** pour l'adversaire
- adapter la priorité des tronçons aux ***itinéraires probables*** de l'adversaire

Generalites

- introduire une ***adaptabilité et plus de flexibilité*** sur l'utilisation des règles

4. Conclusion

Cette solution a pour but d'**imiter les mécanismes de jeu d'un expert** lors d'une partie des aventuriers du rail, voire même d'aller encore plus loin. Notre algorithme, expérimental, offre des résultats intéressants bien qu'il soit encore incomplet.

Il est notamment **efficace dans les phases de jeu préliminaires** mais les phases ultérieures n'ont pas pu être traitées dans leur entièreté par manque d'informations précises et quantifiées et de temps.

En particulier, notre IA est **plus performante** à bien des égards à celle de la **version en ligne des Aventuriers du Rails** - sentiment partagé par notre expert - grâce à l'incorporation de certaines règles.

Un test intéressant serait de confronter notre IA à celle disponible en ligne une fois la **gestion des couleurs** implémentée ...

Cependant notre code comporte plusieurs faiblesses : des critères arbitraires sont appliqués sur le **choix d'objectifs** et le **calcul d'itinéraire** en début de jeu, les **couleurs** ne sont pas prises en compte dans la solution, les calculs d'**adaptation d'itinéraire** en cours de jeu posent parfois des problèmes et les règles sont appliquées de façon **trop catégorique** pour permettre une flexibilité suffisante de l'IA.

Par contre, ce système suit plutôt bien **un certain nombre de règles définies** à partir des entretiens avec l'expert : choix d'objectifs, route continue, priorisation des longues lignes, en particulier de celles de 6 ... Et donne des **perspectives d'évolution** prometteuses !

5. Liste des Fonctions Implémentées

PiocheCartesDestination : Si on entre false en argument, la pioche n'est pas aléatoire et le joueur doit entrer les 3 indices des cartes destinations qu'il souhaite. Si l'entrée est true, l'algorithme pioche 3 cartes destinations distinctes parmi les objectifs disponibles.

SimuleChoixObjectifExpert : Renvoie le choix des objectifs choisis ou donnés aléatoirement selon le système expert qui prend en compte plusieurs facteurs tels que le type de trajet ou la note donnée à l'itinéraire par l'expert.

DeroulerPartie : Fonction appelée en premier, elle gère le déroulement de la partie et certains choix du système expert. Le booléen donné en entrée correspond à la réponse à "Souhaitez-vous piocher les cartes aléatoirement ?".

JouerAdversaire : Si le joueur entre 1, cela signifie que l'adversaire a pioché, s'il tape 2, l'algorithme, le joueur peut entrer le code du tronçon posé par l'autre joueur et simule ainsi la pose de ce tronçon.

JouerIA : Si le joueur entre 1, il pioche, s'il tape 2, l'algorithme appelle la fonction MeilleurChemin et pose le tronçon le plus haut dans l'ordre de priorité.

DijkstraAlgo : Reprise de l'algorithme classique de Dijkstra qui calcule le chemin optimal (de poids le plus faible, qui privilégie donc ici les lignes longues qui sont initialisées avec un poids faible) entre 2 villes-objectif. Il renvoie l'avant dernière ville du chemin.

RenvoyerTousLesNoeudPrecedents : Fonction récursive qui affiche tous les noeuds précédant ville_arrivée jusqu'à ville_départ. Elle trouve l'enchaînement des villes qui constituent le chemin optimal calculé par le Dijkstra en appelant ce dernier.

Enchainement : On ne considère que les 4 ou 6 villes-objectifs. La fonction part de la ville entrée en argument et trouve la ville-objectif non encore visitée la plus proche en faisant appel au Dijkstra et continue jusqu'à les avoir toutes visitées. Elle renvoie alors la liste ordonnée des villes parcourues.

ChoixEnchainementDijkstra : Calcule les différentes combinaisons de Dijkstra en appliquant la fonction précédente sur chaque ville-objectif. Elle compare alors les chemins et renvoie l'itinéraire optimal.

PoserAdversaire : Permet de simuler la capture d'une ligne par l'adversaire en modifiant son coefficient dans la matrice et la rendant virtuellement imprenable pour l'algorithme.

PoserIA : Permet à l'algorithme de poser une ligne sur le plateau. Cette ligne ne pourra plus être reprise plus tard, mais son poids qui sera alors modifié pour devenir très faible permettra à ce tronçon de pouvoir être facilement utilisable dans les itinéraires créés par l'algorithme.

Prioriser : Trie les tronçons dans l'ordre d'importance. Pour cela, on calcule pour chaque tronçon la différence de poids entre le chemin optimal et le chemin sans ce tronçon (et donc les détours nécessaires). Cette différence permet de classer les tronçons dans un ordre de priorité qui influencera l'ordre de la pose des wagons.

EvaluerPriorite : Renvoie le niveau de difficulté à contourner un tronçon en particulier (donc son niveau de priorité calculé par la fonction précédente).

LisserMatricePoids : Permet d'abaisser les coefficients les plus élevés de la matrice des poids et de rehausser les plus bas pour créer une matrice plus homogène

ContinueCheminOptimal : À partir du chemin optimal fourni par la fonction ChoixEnchainementDijkstra, renvoie un itinéraire qui utilise tous les wagons disponibles en ajoutant des rallongements stratégiques.

MeilleurItineraire : Fonction qui fait appel aux fonction ChoixAlgorithmeDijkstra et ContinueCheminOptimal pour renvoyer le chemin optimal à partir du plateau et des objectifs.