<u>Ce rapport a été réalisé par :</u>

-Uprety Dikesh

-Melila Yanis

-Merah Darina

-Goncalves Theo

-Belkhamsa Wassila

Bonne lecture!

« Les aventuriers du rail » est un jeu de société apparu en 2004, et qui a été vendu pas moins de 12 millions de fois. Dans le cadre de l'UE projet de programmation, nous avons développé une version du jeu en Java. Notre version numérique vise à offrir une expérience similaire, tout en incorporant des mécaniques et des fonctionnalités adaptées au support informatique. Ce rapport détaille notre approche de conception et d'implémentation du jeu « Les aventuriers du rail » -ED2, proposé par Mme. Enricha Duchi, en mettant l'accent sur les choix techniques, les algorithmes utilisés et les fonctionnalités développées. Nous discuterons également des défis rencontrés, des solutions apportées et des perspectives pour améliorer et étendre le jeu à l'avenir.

Message important aux évaluateurs :

À la fin de chaque page de ce rapport, vous trouverez le nom de la personne ayant rédigé cette section spécifique. Nous vous serions très reconnaissants si vous pouviez considérer le travail individualisé de chaque membre de l'équipe, tant dans ce rapport que dans le projet en général.

Nous vous prions (gentiment ;) à évaluer le contenu et la qualité du travail fourni par chaque individu, plutôt que de vous concentrer uniquement sur le nombre de commits (qui peuvent être nombreux mais contenir quelques de lignes de code).

Nous vous remercions par avance de votre attention à cet aspect et de votre compréhension. Cela garantira une évaluation juste et équilibrée du travail de chaque étudiant impliqué dans ce projet.

PARTIE MERAH DARINA :

Sommaire

Ces sections fourniront une compréhension approfondie de la conception, de l'implémentation et des choix techniques réalisés dans le cadre de ce projet de programmation inspiré par Les Aventuriers du Rail.

PARTIE MERAH DARINA et Uprety Dikesh:

I. ***Rappel bref des règles du jeu :***

Ce jeu de plateau classique met les joueurs au défi de construire un réseau ferroviaire reliant des villes à travers des routes stratégiquement choisies. Les joueurs collectent des cartes représentant différentes lignes ferroviaires et utilisent ces cartes pour revendiquer des routes sur le plateau de jeu. Notre projet s'inspire de ce concept en apportant une dimension algorithmique avec l'utilisation de graphes pour modéliser le réseau de voies ferrées et calculer des métriques importantes (plus long chemin, graphe aléatoire, vérifier si deux sommets sont liés) .

1. Début du Jeu : Chaque joueur choisit une main initiale de cartes de destination (une carte qui relie deux villes) et un ensemble de cartes de train (aussi appelle cartes wagons). Le plateau de jeu est affiché avec des villes connectées par des lignes représentant les routes ferroviaires.

2. Tour de Jeu :

- Piocher des cartes :À chaque tour, un joueur peut piocher des cartes de train, ou des cartes destinations. Attention. On ne peut piocher qu'une seule carte locomotive (carte de toute couleur confondue). En revanche, nous pouvons choisir deux cartes wagons par tour.

- Revendiquer une route : Le joueur peut revendiquer une route en dépensant un ensemble de cartes de train correspondant à la couleur et la longueur de la route qu'il souhaite revendiquer.

3. Marquer des points : Une fois qu'une route est revendiquée, le joueur marque des points en fonction de la longueur de la route.

1 Wagon → 1point
2 Wagons → +2points
3 Wagons → +5points
4 Wagons → +8points
5 Wagons → +11points
6 Wagons → +15points

4. Cartes de Destination : Les joueurs ont également la possibilité de piocher des cartes de destination qui leur rapportent des points supplémentaires s'ils relient avec succès les villes indiquées sur ces cartes.

5. Fin du Jeu : Le jeu prend fin lorsque l'un des joueurs a moins de 2 wagons (<=2). Le joueur ayant le plus grand nombre de points à la fin du jeu est déclaré vainqueur.

Le score est calculé selon la forme :

+-Points de cartes destinations (en fct de si elle a été accomplie ou non)+ nombre de wagons posés+ points supplémentaires (pour la personne ayant le plus long chemin.

Pourquoi ?

Nous avons choisi "Les Aventuriers du Rail" comme sujet de projet pour plusieurs raisons. Tout d'abord, parmi les jeux proposés, c'était l'un de nos favoris en raison de sa combinaison unique de stratégie et de

PARTIE MERAH DARINA Uprety Dikesh:

tactique, ainsi que de son thème captivant de la construction de chemins de fer (on y a joué longtemps en ligne). Ensuite, ce jeu nécessite une représentation basée sur les graphes pour modéliser le réseau de routes reliant les villes, ce qui nous a permis d'explorer et d'appliquer des concepts d'algorithmique et de théorie des graphes.

De plus, le défi supplémentaire d'implémenter un jeu de société complet en dehors du cadre du programme habituel nous a semblé stimulant et enrichissant d'un point de vue académique. En choisissant ce projet, nous espérions non seulement approfondir nos connaissances en programmation orientée objet et en interfaces graphiques avec Java Swing, mais aussi développer notre capacité à résoudre des problèmes complexes (djikstra, depth-first search, algo de triangulation...)

Enfin, ce projet nous offre une occasion unique d'appliquer nos compétences théoriques à un contexte pratique et ludique, tout en nous permettant d'apprendre et de grandir en tant que développeurs logiciels (même si on est encore bien loin...) Mais tout a un début ! En résumé, "Les Aventuriers du Rail" représente pour nous un choix motivant et enrichissant qui allie passion personnelle pour les jeux de sociétés et apprentissage des techniques de programmation.

II. Explication du choix de la représentation du graphe :

Nous expliquerons dans cette section pourquoi nous avons choisi la liste d'adjacence pour représenter notre graphe de villes et de routes. Cette section discutera des avantages de cette structure de données par rapport à d'autres, comme la matrice d'adjacence.

Plateau de Jeu et Modélisation du Réseau Ferroviaire :

- Naturellement, nous avons modélisé le réseau de villes et de voies ferrées par un graphe. Dans le cadre de notre projet, les arêtes représentent les routes, et les sommets correspondent aux villes.

Les graphes sont en règle générale représentés soit par :

a) Des matrices d'adjacence :

Une matrice d'adjacence est une structure de données bidimensionnelle, où les lignes et les colonnes représentent les sommets du graphe (dans notre cas les villes). Si un sommet i est connecté à un sommet j par une arête (donc deux villes liées par une route), alors l'entrée correspondante $M[i][j]$ sera non nulle (généralement 1). Sinon, elle sera nulle (0). Pour un graphe non orienté (c'est-à-dire qui n'a pas forcément de direction), la matrice est symétrique par rapport à la diagonale principale.

- Avantages Majeurs :

-Temps constant pour la récupération d'arêtes(c'est-à-dire lorsque l'on souhaite vérifier si deux villes sont liées). Pour récupérer l'information sur une arête particulière (une route) entre deux sommets u et v, il suffit d'accéder directement à l'élément matrix[u][v]. Cela se fait en temps constant $O(1)$.

-Efficacité d'espace pour les graphes denses (lorsque le nombre d'arêtes est proche de n^2, où n est le nombre de sommets du graphe.) En d'autres termes, chaque ville est liée avec toutes les autres.

- Inconvénients :

-Inefficacité d'espace pour les graphes dispersés (espace de stockage de O(n^2)). Car la plupart des cases seront égales à 0 (ce qui est notre cas).

PARTIE MERAH DARINA Uprety Dikesh :

- Inefficace pour les graphes dynamiques (ajout/suppression de villes), ce qui n'est pas notre cas.

b) Les listes d'adjacences :

Une liste d'adjacence est constituée d'un tableau de listes (ou tableau dynamique de listes dans certains langages- ArrayList pour java) où chaque élément du tableau représente un sommet du graphe. Chaque liste associée à un sommet contient tous les sommets adjacents à ce sommet.

- Avantages Majeurs :

-Efficacité d'espace pour les graphes dispersés (espace requis de O(n+m)) ou n est la nb de sommets et m le nombre de graphe.

-Efficacité mémoire, consommant moins de mémoire par rapport aux matrices d'adjacence.

-Itération rapide sur les voisins. Pour chaque ville v, accéder à sa liste de voisins adj[v] se fait en temps constant O(1), ce qui nous importe plus lors de la construction de routes.

- Inconvénients :

-Récupération d'arêtes plus lente (car on doit parcourir la liste).

-Surcoût mémoire supplémentaire pour stocker des références pour chaque sommet.

Nous avons finalement opté pour la seconde option. Pourquoi ?

Dans notre cas, le graphe n'est pas dispersé. Nous avons donc choisi les listes d'adjacence afin d'éviter une consommation inutile de mémoire avec une matrice d'adjacence. Vérifier s'il existe une route entre deux villes/pays n'est pas une opération fréquente dans le contexte du projet (mis à part dans l'algo de vérification d'une carte destination). Cette représentation nous permet de gérer efficacement le réseau de routes tout en conservant une utilisation optimale des ressources système. Même si les graphes ne sont pas explicitement au programme, en faisant quelques recherches, nous avons pu conclure que la représentation par listes d'adjacence est souvent préférée pour des cas comme les nôtres. Cette représentation offre une meilleure efficacité d'espace et de mémoire pour les graphes dispersés, tout en permettant une gestion efficace des routes et des connexions entre les villes.

Pour résumer : notre graphe sera sous forme de tableaux de listes, où l'indice i représente une ville, et la liste associée à cet index représente les villes qui lui sont adjacentes (si l'indice i représente la ville de Lyon, alors la liste associée à cet indice correspond aux villes qui possèdent un chemin la liant à Lyon).

III. Explication du design pattern choisi (observer et observable) lié avec le MVC et justification du choix :

Dans cette section, nous présenterons le design pattern en question utilisé en lien avec le modèle MVC. Cette section expliquera comment ce choix architectural facilite la mise à jour et la notification des vues lors de modifications dans le modèle, assurant ainsi une séparation claire des responsabilités.

L'utilisation du design pattern Observer/Observable dans ce contexte est judicieux pour plusieurs raisons :

PARTIE MERAH DARINA :

1. **Séparation des préoccupations :**

- Modèle (Model) : Il représente les données de l'application et la logique (majoritairement la classe GameBoard). Il va donc étendre l'interface observable étant donné qu'il est l'objet que la vue observe. A chaque modification importante du code, les différents observateurs (vue) seront notifié du changement afin de mettre à jour leur interface (à travers la méthode update).

- Vue (View) : Elle est responsable de l'affichage des données au format souhaité par l'utilisateur. La Vue utilise le pattern Observer pour s'enregistrer auprès du Modèle et recevoir les notifications de changements.

- Contrôleur (Controller) : Il agit comme un intermédiaire entre le Modèle et la Vue. Lorsque l'utilisateur interagit avec la Vue, le Contrôleur déclenche les actions nécessaires sur le Modèle, ce qui peut entraîner des changements notifiés à la Vue via le pattern Observer. La vue n'interagit donc jamais directement avec le modèle.

Pourquoi ce design pattern ?

- Le pattern Observer permet d'ajouter ou de supprimer des vues (observers) sans modifier le modèle (observable) ou les autres vues. Cela permet une meilleure évolutivité de l'application (limite les dépendances, et l'ajout des futures extensions se fait plus aisément). Il est également possible d'ajouter de nouveaux types de données (observables) sans affecter les vues existantes (Exemple concret avec ChoicePanel qui s'enregistre autant observateur de de InfoTime). Cette architecture favorise la maintenabilité en isolant les composants qui changent souvent (vues) de ceux qui changent rarement (modèles).

- L'utilisation du pattern Observer permet de rendre l'interface utilisateur réactive aux changements de données sans nécessiter de polling continu (comme dans le jeu du premier semestre) ou de mise à jour manuelle de l'affichage.

En conclusion, l'association du design pattern Observer/Observable avec l'architecture MVC est judicieuse car elle permet une meilleure séparation des préoccupations, une extensibilité et une maintenabilité accrues, ainsi qu'une réactivité améliorée de l'interface utilisateur aux changements de données. Cela rend le code plus robuste et plus facile à gérer, en particulier dans les applications à interface utilisateur dynamique.

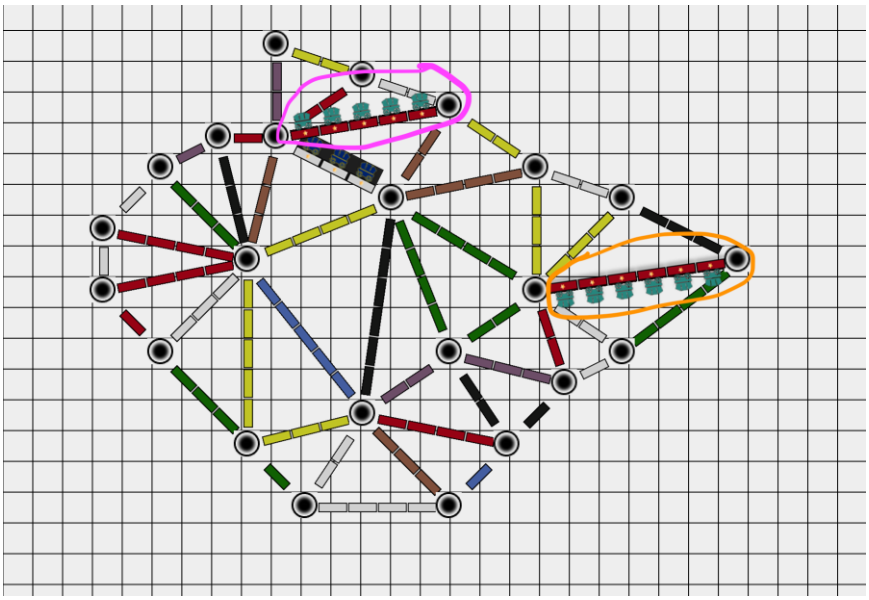
PARTIE Melila Yanis:

IV. Explication de l'algorithme qui calcule le plus long chemin :

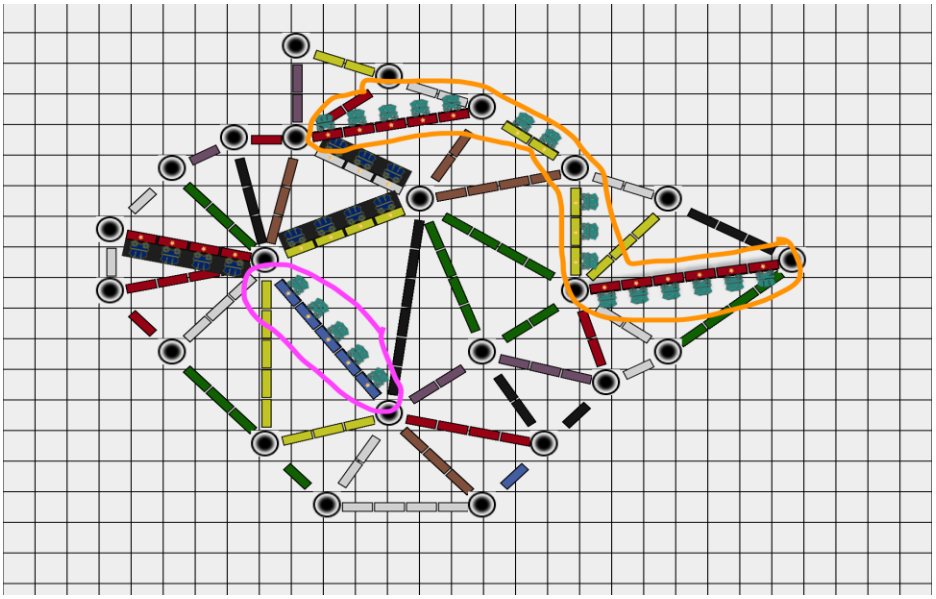
Nous avons développé un algorithme utilisant l'algorithme de Dijkstra pour calculer le plus long chemin dans le réseau du joueur. Ce chemin est important car il rapporte des points supplémentaires.

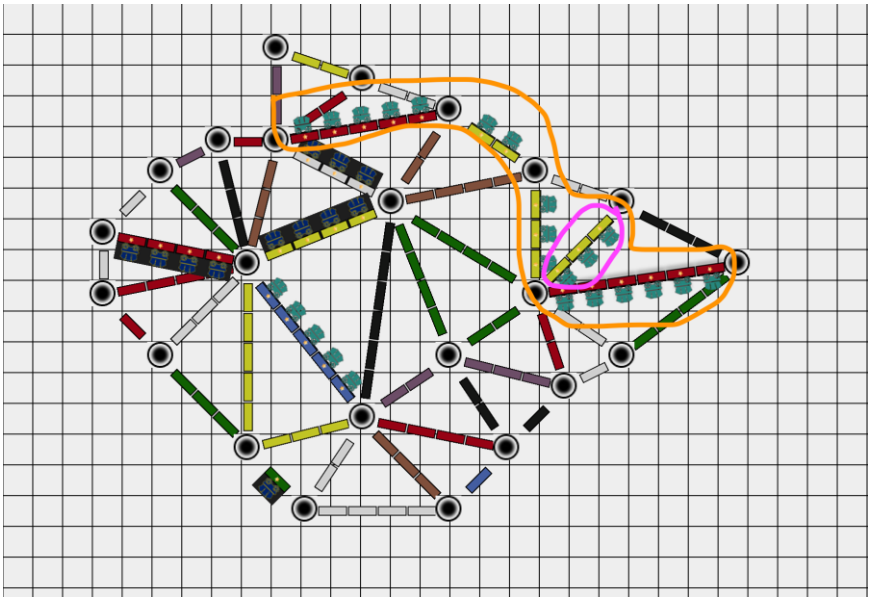
Différents cas détaillés :

Cet algorithme vérifie plusieurs cas pour calculer le plus long chemin. Pour montrer cela, on va le faire par illustration :



Dans ce cas de figure, le joueur qui jouera les wagons verts aura comme plus longue distance, une longueur de taille « 6 » (car 6 > 5) qui corresponds aux wagons encerclé en orange. Il ne comptabilisera pas les wagons que le joueur vert à mis dans l'encerclé rose car les deux chemins ne sont pas relié pour le moment.
Dans ce cas de figure là, le joueur vert aura une longueur de taille « 16 » car les chemins qu'il aura remplis sont reliés (16 = 6 + 3 + 2 + 5) et à la différence du chemin vert qu'il a placé dans l'encerclé roses, 16 reste un plus long chemin que 5.





Ici, le joueur vert à continuer en ajoutant le chemin dans l'encerclé rose.
Même si ce chemin est relié au chemin orange qui est le plus long actuellement, il ne va pas le comptabiliser car il est sur un chemin différent. En d'autres termes, il se situe sur un autre embranchement.
En somme, le plus long chemin dans cette figure restera le même que celui de la figure précédente, à savoir, de taille 16.

C'est là qu'on remarque pourquoi il est nécessaire d'utiliser cet algorithme. Il faut vérifier à chaque liaison d'un chemin, le chemin qui est le plus long et récupérer celui là au lieu de l'autre qui est plus court.

PARTIE Uprety Dikesh:

V. ***Explication de l'algorithme qui vérifie si deux villes sont bien connectées*** :

Cette section présentera l'algorithme (inspire de depthFirstSearch) utilisé pour vérifier la connectivité entre deux villes dans notre graphe.

Dans ce jeu, les joueurs doivent choisir des cartes destinations qui indiquent deux villes qu'ils doivent relier en posant des wagons sur la carte. L'objectif est de vérifier si le joueur a réussi à relier effectivement les deux villes figurant sur la carte destination.

Pour réaliser cette vérification, on a choisi d'utiliser la technique de recherche en profondeur (depth First Search):

1*. Ajout des routes :*

- Lorsque le joueur réussit à relier deux villes en posant des wagons, on ajoute ces routes dans une ArrayList. On ajoute également les routes en sens inverse pour refléter le fait que les routes sont bidirectionnelles. Par exemple, si le joueur relie la ville A à la ville B, j'ajoute les routes A->B et B->A dans la liste. Cela garantit la cohérence du graphe (la matrice d'adjacence aurait bien servi...)

2. Vérification de la liaison entre les villes :

- Pour vérifier si les deux villes sur la carte destination sont reliées, on utilise la méthode hasPath(String source, String destination). Cette méthode initialise un ensemble appelé visited pour suivre les villes déjà visitées. Ensuite, on fait appel à la méthode privée `depthFirstSearch` pour effectuer une recherche en profondeur à partir de la ville source vers la destination.

3. Méthode hasPath(String source, String destination) :

- Cette méthode vérifie s'il existe un chemin entre deux villes spécifiées dans le graphe. Elle fait appel à la méthode privée depthFirstSearch pour effectuer la recherche en profondeur.

4. Méthode privée depthFirstSearch(String source, String destination, Set<String> visited):

- Cette méthode effectue une recherche en profondeur récursive à partir de la ville source jusqu'à ce que la ville de destination soit atteinte.

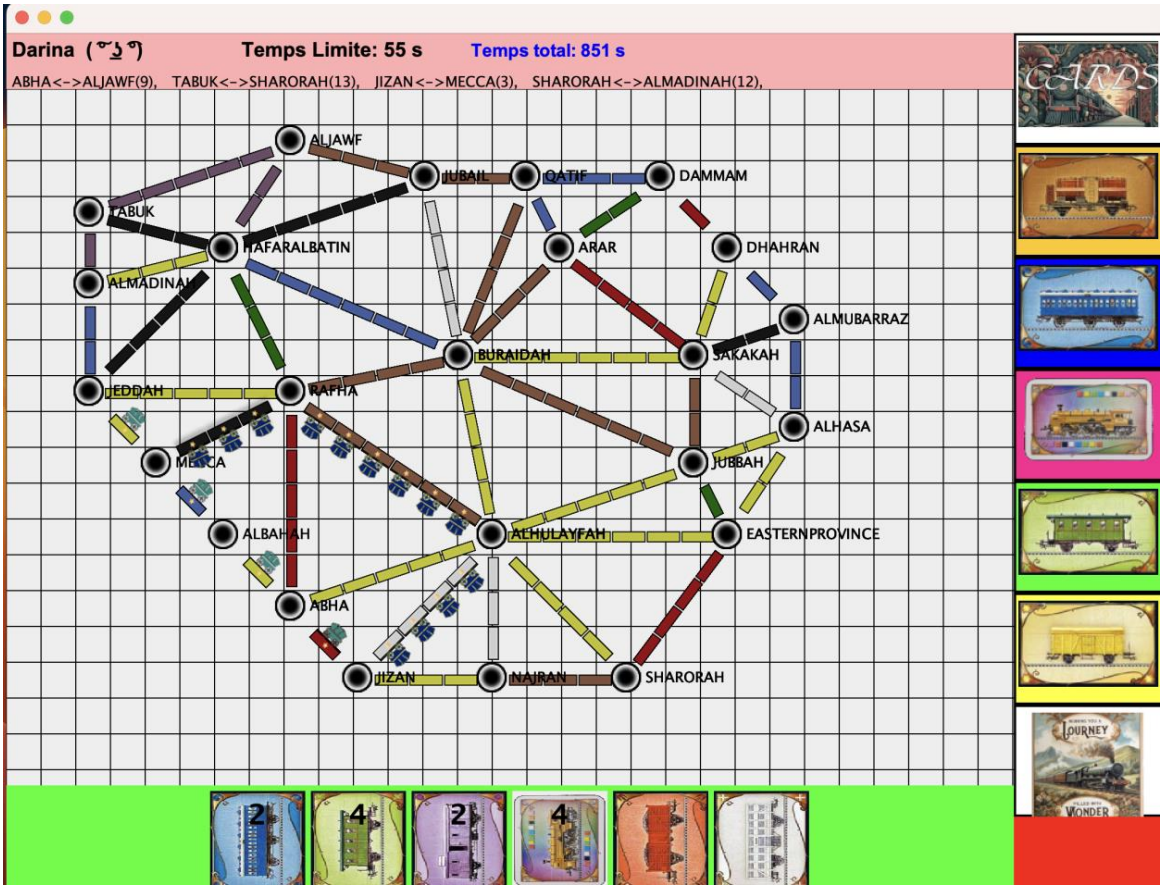
- Pour éviter les boucles infinies, elle ajoute la ville source à l'ensemble visited.

- Elle parcourt chaque route dans la liste map(attribut ou on ajoute les routes) et vérifie si la ville source correspond à la source de la route. Si oui, elle récursivement appelle depthFirstSearch avec la ville de destination de la route actuelle comme nouvelle source. Elle retourne true si un chemin est trouvé, sinon false.

La complexité totale de cet algorithme dépend principalement de la densité du graphe (qui n'est pas si dense que ça), mais elle peut généralement être approximée à O(V+E), où V est le nombre de villes

et E est le nombre de routes. Cette complexité est assez efficace pour des graphes de taille modérée.

PARTIE Uprety Dikesh:




```
JIZAN---------LES VILLES MISES DENDANS----------ALHULAYFAH
JIZAN---------LES VILLES MISES DENDANS----------ALHULAYFAH
RAFHA---------LES VILLES MISES DENDANS----------ALHULAYFAH
RAFHA---------LES VILLES MISES DENDANS----------ALHULAYFAH
DIKESH
    DIKESH       MECCA <--------------->ABHA  il reussi a relié ou pas  ?800true
DIKESH
    DIKESH       JEDDAH <--------------->JIZAN  il reussi a relié ou pas  ?800true
########################
Darina
    Darina       ABHA <--------------->ALJAWF  il reussi a relié ou pas  ?800false
Darina
    Darina       TABUK <--------------->SHARORAH  il reussi a relié ou pas  ?800false
Darina
    Darina       JIZAN <--------------->MECCA  il reussi a relié ou pas  ?800false
Darina
    Darina       SHARORAH <--------------->ALMADINAH  il reussi a relié ou pas  ?800false
########################
MECCA---------LES VILLES MISES DENDANS----------RAFHA
MECCA---------LES VILLES MISES DENDANS----------RAFHA
DIKESH
    DIKESH       MECCA <--------------->ABHA  il reussi a relié ou pas  ?850true
DIKESH
    DIKESH       JEDDAH <--------------->JIZAN  il reussi a relié ou pas  ?850true
########################
Darina
    Darina       ABHA <--------------->ALJAWF  il reussi a relié ou pas  ?850false
Darina
    Darina       TABUK <--------------->SHARORAH  il reussi a relié ou pas  ?850false
Darina
    Darina       JIZAN <--------------->MECCA  il reussi a relié ou pas  ?850true
Darina
    Darina       SHARORAH <--------------->ALMADINAH  il reussi a relié ou pas  ?850false
########################
```

On remarque pour exemple de dikesh (représenter par wagons vert) que il a réussit a faire lié MECCA et ABHA, JEDDAH et JIZAN. D'où les valeurs sont true.

On remarque également de Darina (représenter par wagons bleu) a réussi a lié JIZAN et MECCA d'où les c'est true mais juste avant false parce que avant elle avait pas réussi. (e

PARTIE MERAH DARINA :

VI. Généralisations

- Nous avons introduit deux généralisations pour enrichir le jeu :

1. Le joueur a le choix entre presque 15 cartes de pays différents (ce qui n'est pas le cas dans le jeu initial) et 10 cartes destinations par pays choisi.

2. Un mode aléatoire qui génère un graphe connexe et presque ;) planaire (à 2-3 routes près), ce qui n'est pas une condition initiale dans les généralisations proposées.

a) Choix des différents pays :

Nous expliquerons dans cette partie comment les différents continents et pays ont été représentés à l'aide d'un parseur de texte. Cela inclura le processus de lecture et de chargement des données des fichiers textes pour construire le graphe des villes et des routes.

1. Classe `MapsParser` (là où l'on peut voyager aux 4 coins du monde...) :

- Cette classe est utilisée pour lire et interpréter les fichiers texte contenant les informations sur les cartes des pays. Plus précisément c'est la Méthode `parseMapFile(String continent, String country) qui construit le chemin du fichier en ajoutant le nom du continent à un chemin de ressources prédéfini.

Fonctionnement :

```
1   france
2   2->1,21   5->14,6   20->1,18   19->18,17   13->17,16   16->15,17,24   10->22,15
3   russie
4   27->26,28,29,30,38   30->31,32   32->33,35,36,31,44   33->34   34->37   38->39
5   finlande
6   46->48   48->49   49->51   51->52   52->53   53->54,61,62   54->55,62,57,58   55-
```

- Lit le fichier texte intitule <continent>.txt ligne par ligne à partir du pays spécifié.

- Appelle `extractRoutes(String city)` pour extraire les routes entre les villes qui divise la chaîne de la ville source et ses destinations séparées par `->`.

- Stocke ces informations dans une `HashMap`.

2. Classe `City` et Enum `CityType`** :

- La classe City représente une ville avec des informations spécifiques à la ville comme le type de ville (le nom) et ses coordonnées. Elle contient une liste des noms de ville ainsi que leurs coordonnées (qui seront donc plus tard liées à la classe City).

PARTIE MERAH DARINA :

Amélioration possible ?

L'utilisation d'une énumération pour représenter les types de ville nous a semblé initialement judicieuse en raison de sa simplicité et de sa facilité d'utilisation dans le code. Voici pourquoi cela nous paraissait comme un choix initial judicieux :

1. Simplicité et Clarté : Une énumération est simple à déclarer et à utiliser. Elle permet de définir clairement une liste de valeurs possibles pour un type spécifique (City dans notre cas).

2. Type-Safety : En Java, les énumérations offrent la sécurité de type. Cela signifie qu'on ne peut pas attribuer autre chose que les constantes définies dans l'énumération à une variable de ce type.

3. Facilité d'Extension : Etant donne qu'on a une liste finie de types de ville, une énumération est un bon choix car elle peut être facilement étendue et modifiée.

Cependant, nous nous sommes rendu compte avec l'ajout des différents pays que l'énumération peut rapidement devenir conséquente en termes de taille. Il devient assez difficile de modifier les coordonnées d'une ville assez rapidement.

Par conséquent, utiliser un fichier texte pour stocker et charger dynamiquement les types de ville aurait été préférable

PARTIE GONCALVES THEO

b) Algo graphes aléatoires + taches :

Mon implication dans le projet de modélisation du jeu a été multifacette. J'ai principalement travaillé sur les aspects essentiels tels que le joueur lui-même et le plateau de jeu. Un de mes principaux axes de travail a été le panneau situé en bas à gauche de l'écran, qui affiche la main du joueur. Cela a nécessité un travail important sur les graphismes, où j'ai proposé plusieurs designs qui ont été soumis au groupe pour évaluation.

En collaboration avec Wassila, j'ai également contribué à la conception de la partie du profil du joueur. J'ai pris en charge la création de la carte du joueur, en mettant l'accent sur la représentation visuelle précise et attrayante des informations pertinentes.

Dans le développement des fonctionnalités de jeu, j'ai initialement conçu une fonction tour(), mais elle a été remplacée ultérieurement par le Timer, une solution plus efficace adoptée par l'équipe. J'ai également travaillé sur l'identification du chemin le plus long, bien que la solution proposée par Yanis se soit révélée plus optimale, donc nous l'avons intégrée.

Un aspect sur lequel j'ai consacré beaucoup de temps a été le mode aléatoire, visant à générer un graphe connexe aléatoire. Bien que je n'aie pas réussi à implémenter efficacement l'algorithme de la Triangulation de Delaunay en raison de contraintes de temps et de ma propre expertise, j'ai néanmoins intégré une méthode alternative. Mon algorithme relie les deux villes les plus proches de

chaque ville pour garantir la connectivité du graphe, tout en veillant à ce que chaque ville soit accessible à partir de n'importe quel point et soit connectée à au moins deux autres villes.

Mon travail sur la modélisation du jeu a donc été une combinaison d'efforts pour rendre l'expérience de jeu aussi fluide, visuellement attrayante et fonctionnelle que possible, en dépit des défis techniques rencontrés.

VII. ***Diagramme des classes :***

Un diagramme détaillé des classes est fourni, montrant les relations entre les différentes classes du système, y compris les modèles, les vues, les contrôleurs et les observateurs. Les fichiers ressources/musique.. ne seront pas compris. Vous le trouverez en pdf dans un fichier joint.

PARTIE Belkhamsa Wassila

IX. ***Explications de taches***

Pendant le projet j'ai participer a divers taches au debut :j'avais le panel playerIdPanel qui m'a ete attribuer j'ai donc travailer sur l'affichage des joueurs avec tout leur attributs et les photo de profil j'ai fait en sorte que cette partie sois relier au model pour que cette partie recupere en temps reel toute les information du joueur, cette partie etait aussi relier au menue de debut de jeu car selon les information rentrer dans le menu le nombre de joueur et les attributs de chaque joueur se metait a jour dans cette partie j'ai eu quelques difficulte en ce qui concerne le lien de ma partie avec toutes les autres parties. apres avoir terminer cette partie ma tache est devenue la creation des maps de chaque pays dans les differents continents j'ai creer les differentes villes de chaque pays sous forme de point en faisant en sorte que les differentes villes dessinent la forme des different pays en faisant en sorte que les coordonnees dessinent le bon shema j'ai aussi creer les differentes routes qui relie toutes les villes de chaque pays en faisant en sorte d'obtenir a la fin un graphe connexe. ensuite je me suis occuper de creer les fonds de cartes pour les metre en arriere plan de chaque pays avec un logiciel special. apres ca ma derniere tache etait de m'occuper de la fin de partie en ce qui concerne l'affichage des scores et du classement des joueur apres la fin de partie et aussi de definir le joueur gagnant j'ai rencontrer des difficulter d'affichage mais qui se sont regler tres vite. voila pour mon implication dans le projet j'ajouterai que j'ai beaucoup aimer travailer avec mon groupe car il y'avait de l'entraide entre nous on reglais les problemes ensembles et on se concertais a chaque difficulte ce qui nous a permis d'apprendre l'un de l'autre et nous a eviter de bloquer trop longtemps sur nos difficultes.

PARTIE Merah Darina et Uprety Dikesh :

Conclusion :

Notre jeu est une version Java du jeu de plateau "Les Aventuriers du Rail" revisite à notre sauce ;), mettant l'accent sur des aspects algorithmiques importants. Notre projet comprend une interface utilisateur simple et des algorithmes de graphes pour calculer les chemins optimaux et évaluer les objectifs atteints. Ce rapport détaille les choix de conception, les algorithmes implémentés et les améliorations possibles pour l'avenir du jeu.