Rapport de Projet

Les Aventuriers du Rail

Zachary BARBON-EVIS
Lancelot RAMIS

10/05/2025

Table des matières

Introduction

Les Aventuriers du Rail est un jeu de société stratégique dans lequel les joueurs collectent des cartes pour construire des lignes de chemin de fer entre des villes d'un continent. Chaque joueur tente de remplir des objectifs secrets tout en bloquant, si possible, les adversaires. Le jeu repose sur la prise de décisions tactiques, la gestion de ressources, et une vision globale du plateau.

Dans le cadre de notre formation en informatique, ce projet a pour but de traduire ces mécaniques ludiques en une architecture logicielle modulaire et cohérente. Il constitue une application concrète de la programmation orientée objet, de la modélisation par diagrammes de classes, et de la structuration de projet. Il s'inscrit également dans une démarche progressive, où l'on passe d'un modèle de jeu manuel à une simulation automatique puis intelligente, tout en respectant des contraintes de clarté, de testabilité et d'extensibilité du code.

Le projet repose sur plusieurs objectifs pédagogiques :

— Mettre en place une modélisation complète du jeu : carte, routes, joueurs, cartes wagon et cartes objectif.

— Déterminer et implémenter les actions possibles à chaque tour, en respectant les règles du jeu.

— Proposer un joueur automatique, d'abord aléatoire, puis optimisé pour chercher à atteindre ses objectifs.

— Calculer les scores en prenant en compte les objectifs réalisés, les routes posées et le plus long chemin.

— (Optionnel) Améliorer l'expérience utilisateur avec une aide au jeu, une IHM graphique ou des alertes intelligentes.

Ce rapport présente l'état actuel de l'avancement, la structure générale du programme, les principales classes et méthodes, ainsi que les perspectives pour la suite du développement.

Chapitre 1

Analyse du problème

1.1 Définition exacte du problème

Le projet consiste à modéliser informatiquement une version simplifiée du jeu de société *Les Aventuriers du Rail*. Il s'agit de simuler le déroulement d'une partie à l'aide d'une modélisation orientée objet, en respectant les règles de base du jeu original.

Le cœur du problème réside dans la mise en place d'une structure de données représentant :

- la carte du jeu (un graphe dont les sommets sont des villes et les arêtes sont les routes disponibles) ;
- les joueurs, chacun possédant un stock de wagons, des cartes wagon, et des objectifs à remplir ;
- les actions possibles à chaque tour : piocher des cartes wagon, capturer une route, ou éventuellement choisir de nouveaux objectif (option désactivée pour le joueur automatique dans notre simplification) ;
- les règles du jeu, notamment les contraintes de pioche, la capture de routes grises (par n'importe quelle couleur), et la limitation à deux actions maximum par tour ;
- la mise à jour du plateau à chaque action, avec vérification de la fin de partie.

Le système doit également permettre d'attribuer les cartes d'objectifs et de wagon en début de partie, de déterminer les coups possibles à chaque tour et de les exécuter, tout en assurant le suivi du score.

En plus de cette modélisation du jeu pour un joueur humain, le projet intègre une logique de joueur automatique (aléatoire ou intelligent) permettant de tester plus rapidement les mécaniques. Enfin, le calcul des scores repose sur des structures de graphe : composantes connexes pour la vérification des objectifs, et algorithme de parcours pour identifier le plus long chemin.

Ce projet a donc pour but de fournir une simulation fidèle, évolutive et testable du jeu, en mettant l'accent sur une organisation modulaire et une structuration orientée objet claire.

1.2 Domaines d'utilisation

Le programme développé a vocation à être utilisé dans plusieurs contextes pédagogiques ou expérimentaux.

- **Aide au joueur humain** : en simulant la partie, le programme peut rappeler les objectifs en cours, les routes encore disponibles, ou avertir lorsqu'un objectif

devient difficile voir impossible à atteindre. Des fonctionnalités supplémentaires pourront également conseiller des actions possibles à un joueur réel.

— **Test de stratégie** : en comparant les performances de joueurs automatisés avec des comportements différents (aléatoire, optimisé, prudent, etc.), le programme permet d'analyser l'efficacité de différentes stratégies sur un même plateau.

— **Expérimentation d'un joueur automatique** : une première version aléatoire du joueur permet de tester les règles du jeu. À terme, l'intégration d'un joueur intelligent ouvre la voie à des travaux sur l'optimisation, la recherche de chemins, et l'intelligence artificielle rudimentaire.

— **Application pédagogique** : le projet illustre de nombreux concepts de l'enseignement en informatique : modélisation par classes, graphe orienté, algorithmes de parcours, règles de gestion, tests unitaires, documentation, et structuration en modules.

1.3 Limites et simplifications

Afin de rendre le projet plus accessible et réalisable dans un cadre pédagogique limité, plusieurs simplifications ont été mises en place :

— **Pas de sélection des objectifs** : chaque joueur reçoit trois cartes itinéraire qu'il est contraint de garder. L'action de tirer de nouveaux objectifs en cours de partie a été retirée pour les joueurs automatiques.

— **Trois joueurs fixes** : le nombre de joueurs est limité à trois pour éviter la complexité du multijoueur complet tout en conservant la richesse d'interaction entre joueurs.

— **Pas d'interface graphique complète** : une visualisation simple du graphe du plateau est proposée via `matplotlib`, mais aucune interface interactive n'est encore amorcée.

Ces choix permettent de concentrer l'effort sur la mécanique de base du jeu, tout en préparant le terrain pour une extension ultérieure pour la fin du projet.

Chapitre 2

Description du programme

2.1 Organisation générale

Le déroulement d'une partie suit les grandes étapes classiques du jeu *Les Aventuriers du Rail*, adaptées ici dans une version simplifiée et pilotée par un contrôleur central (la classe `Table`).

Mise en place

Au lancement du programme soit une partie est chargée (à l'aide d'une sauvegarde sous fichier Json), soit elle est initialisée :

- Le plateau de jeu est initialisé (chargement des villes et routes à partir d'une structure pré-définie en début de programme).
- Trois joueurs sont créés et reçoivent chacun un nombre initial de wagons (45), trois cartes objectif, et quatre cartes wagon.
- Les cartes wagon visibles sont tirées et affichées (cinq cartes).

Déroulement d'un tour de jeu

Chaque tour, le joueur actif a le choix entre deux actions principales :

- **Piocher des cartes wagon** :
  - Il peut piocher deux cartes, visibles ou cachées.
  - Si la première carte est une locomotive visible, le tour se termine immédiatement.
  - Si la deuxième carte sélectionnée est une locomotive visible, les règles interdisant ce choix, un nouveau choix lui est demandé.
- **Capturer une route** :
  - Il sélectionne une route disponible.
  - Il doit disposer du bon nombre de cartes wagon de la couleur correspondante (ou Locomotives/jokers).
  - S'il possède les cartes requises et assez de wagons, il prend la route, sinon on lui demande de refaire un autre choix.
  - Son score est mis à jour en fonction de la longueur de la route.
- **Piocher une carte objectif** (non accessible au joueur auto) :
  - Il pioche 3 cartes de la pioche correspondante.
  - Il choisit de garder une à 3 cartes et repose les autres le cas échéant.

Après son action, le tour passe au joueur suivant. La partie continue ainsi jusqu'à ce qu'un joueur ait deux wagons ou moins, déclenchant le dernier tour.

Schéma général du fonctionnement

Un schéma synthétique peut représenter cette logique :

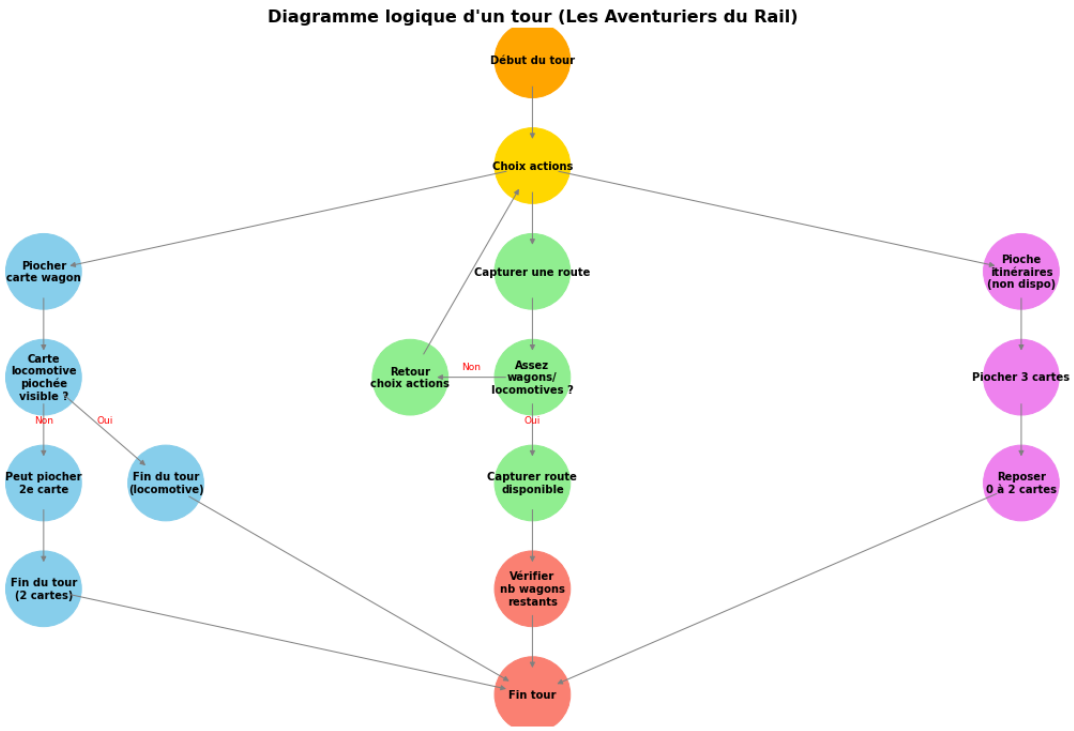

FIGURE 2.1 – Schéma du déroulement d'un tour de jeu

Ce schéma illustre les choix possibles à chaque tour et les principales conditions vérifiées avant de poursuivre.

2.2 Diagramme de classes

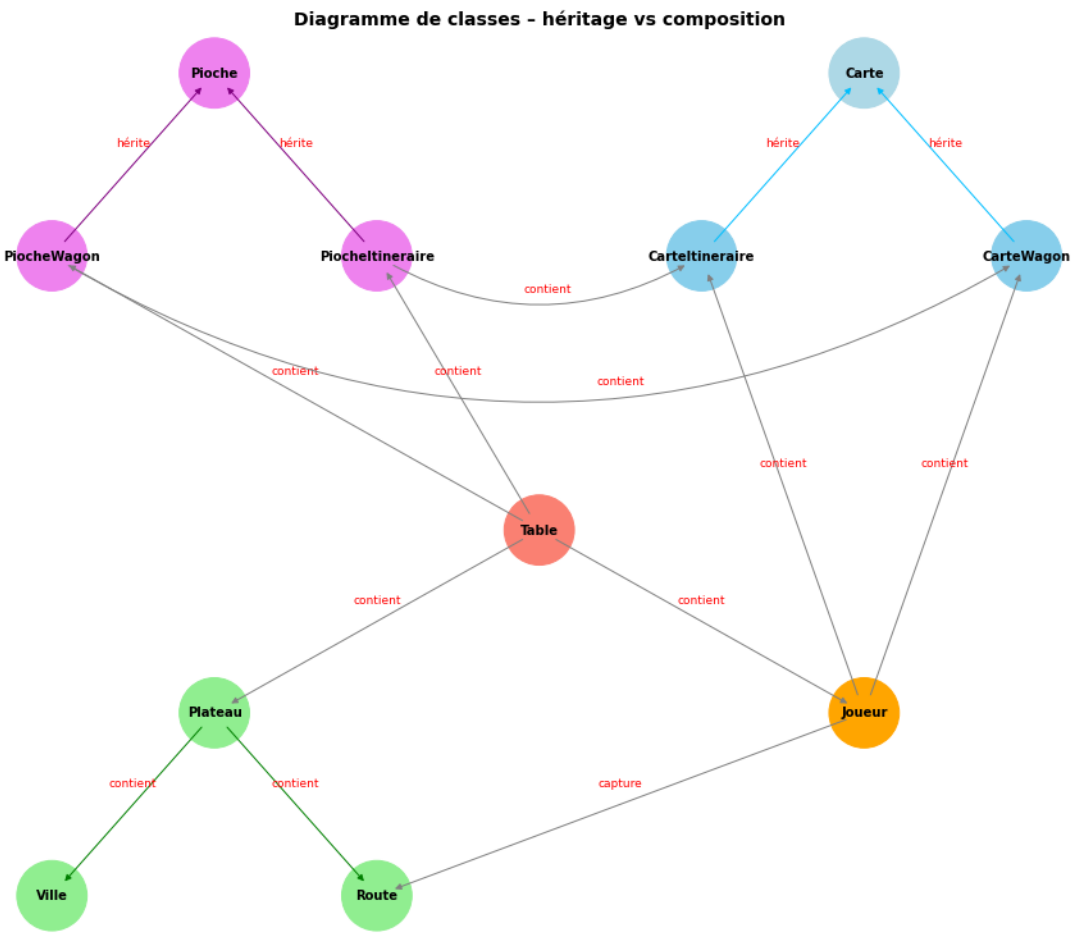

FIGURE 2.2 – Diagramme de classe

2.3 Classes principales

Le projet repose sur une modélisation orientée objet fidèle à la structure du jeu Les Aventuriers du Rail. Les principales classes implémentées sont les suivantes :

Classe `Table`

Cette classe coordonne la partie. Elle contient l'ensemble des objets nécessaires au déroulement du jeu : les joueurs, le plateau, les pioches (cartes wagon et cartes itinéraire), ainsi que la gestion des tours. C'est le point d'entrée principal de l'exécution du jeu.

— Initialise les composants de la partie (joueurs, pioches, plateau, objectifs).
— Gère le déroulement du jeu tour par tour.
— Implémente les règles de pioche, de capture de route, et les contrôles de validité.

Classe `Joueur`

Un joueur est caractérisé par :
— Un nom, un nombre de wagons disponibles.

— Une main de cartes wagon et une main de cartes objectif.
— Un score évolutif.

Le joueur interagit avec la table en piochant, en capturant des routes, et en remplissant ses objectifs. Des méthodes de jeu automatique seront ajoutées par la suite.

Classe `Plateau`

Cette classe représente la carte du jeu, sous forme d'un graphe via `networkx`. Elle contient :

— La liste des villes (sommets du graphe).
— La liste des routes entre villes (arêtes, avec couleur et longueur).
— Une méthode d'affichage graphique du plateau à l'aide de `matplotlib`.

Elle est utilisée pour déterminer les routes disponibles et afficher l'état du jeu.

Classes `Carte`, `CarteWagon`, `CarteItineraire`

La classe `Carte` est une classe abstraite commune aux cartes du jeu, utilisée comme base pour organiser les classes `CarteWagon` et `CarteItineraire`. Elle permet de regrouper certains comportements communs et de faciliter la représentation du jeu.

— `CarteWagon` hérite de `Carte` et représente une carte de couleur (ou locomotive). Elle contient une information sur la couleur, utilisée pour valider la capture d'une route.
— `CarteItineraire` hérite aussi de `Carte` et représente une carte objectif : elle contient deux villes et une valeur en points. L'objectif est réussi si les deux villes sont connectées à la fin de la partie.

Classes `Pioche`, `PiocheWagon`, `PiocheItineraire`

La classe `Pioche` est une classe générique pour toute pile de cartes tirables. Elle gère l'instanciation des cartes et leur mélange. Deux sous-classes spécialisent ce comportement :

— `PiocheWagon` contient les cartes wagon, gère une pioche visible (5 cartes affichées), et applique les règles spécifiques du jeu (ex : une locomotive visible en premier interdit un deuxième tirage).
— `PiocheItineraire` contient les cartes objectif. Lors d'une pioche, le joueur reçoit trois cartes et doit en conserver au moins une (hors initialisation).

La séparation de ces classes facilite la gestion de règles différentes pour chaque type de carte tout en mutualisant la logique commune dans la classe parente `Pioche`.

Classe `Ville`

Une ville est un nœud du graphe du plateau généralement définie par son nom et ses coordonnées graphiques. Elle est utilisée pour placer les routes sur le graphe et pour calculer les connexions.

Classe `Route`

Une route est une liaison entre deux villes. Elle est caractérisée par :

— Sa longueur (nombre de wagons nécessaires).

— Sa couleur (ou gris pour n'importe quelle couleur).
— Le joueur qui l'a capturée (ou `None` si encore disponible).

Elle est représentée par une arête dans le graphe du plateau.

Classe `JoueurAuto`

Cette classe hérite de `Joueur` et permet de simuler un joueur automatique. L'objectif est de valider le bon déroulement du jeu sans intervention humaine. Sa méthode principale, `jouer_automatiquement`, intercepte les appels à `input` pour simuler les réponses d'un joueur.

Les décisions sont prises selon une stratégie aléatoire mais cohérente :
— Choix pondéré entre piocher ou capturer une route, en fonction du nombre de cartes disponibles.
— Sélection automatique d'une carte visible ou cachée pour la pioche.
— Tentative de capture d'une route parmi les routes disponibles.

Ce joueur automatique constitue une première étape vers une intelligence plus élaborée. Il a notamment permis d'exécuter de longues parties de test sans action manuelle.

Un exemple de code :

```
def jouer_automatiquement(self, table):
    original_input = builtins.input
    builtins.input = self.repond_automatiquement
    try:
        table.jouer_tour(self)
    finally:
        builtins.input = original_input
```

2.4 Méthodes importantes

Certaines méthodes du projet jouent un rôle central dans le déroulement du jeu. Nous présentons ici celles qui participent directement à l'exécution des actions du joueur et à la mise à jour du plateau.

`jouer_tour(joueur)` — classe `Table`

Cette méthode coordonne un tour de jeu pour le joueur donné. Elle appelle selon les cas les fonctions de pioche ou de capture de route, puis actualise le plateau. Elle gère les règles suivantes :
— Choix entre piocher ou capturer une route.
— Interdiction de piocher une locomotive en deuxième carte.
— Passage au joueur suivant.

`piocher_cartes_wagon(joueur)` — classe `Table`

Permet au joueur de piocher une ou deux cartes wagon. Elle intègre les règles du jeu :
— Le joueur peut choisir parmi les cartes visibles ou tirer à l'aveugle.
— Si la première carte est une locomotive visible, le joueur ne peut pas piocher une deuxième carte.

— Si le joueur choisit une locomotive en deuxième carte, il est contraint de refaire un choix.

`capturer_route(joueur)` — classe `Table`

Cette méthode permet à un joueur de capturer une route, c'est-à-dire :
— Vérifier qu'il possède les cartes nécessaires.
— Vérifier qu'il lui reste assez de wagons.
— Retirer les cartes utilisées et décrémenter le stock de wagons.
— Mettre à jour le plateau : la route devient indisponible et est associée au joueur.
— Ajouter les points correspondant à la longueur de la route.

`plus_long_chemin(joueur)` — classe `Table`

Calcule la longueur du plus long chemin continu capturé par un joueur. Cette méthode repose sur une exploration récursive du sous-graphe constitué uniquement des routes prises par ce joueur, représentées dans `networkx`. L'algorithme visite les routes une à une, en construisant récursivement tous les chemins possibles sans repasser par les mêmes arêtes.

Cette méthode est utile :
— Pour attribuer le bonus de 10 points du « plus long chemin » en fin de partie.
— Pour évaluer la progression stratégique du joueur automatique ou humain.
— Pour illustrer l'utilisation d'une fonction récursive dans le cœur du moteur de jeu.

Méthodes à venir

D'autres méthodes clés seront développées dans la suite du projet :
— Gestion du joueur automatique.
— Ajustements fonctionnels de certaines classes

Chapitre 3

Figures imposées

Le projet respecte plusieurs des figures imposées, réparties en deux catégories : les six figures communes à tous les sujets, et trois figures supplémentaires sélectionnées par le binôme.

Nous listons ci-dessous ces figures, en distinguant celles qui sont clairement implémentées, celles qui restent à travailler, et celles qui ne sont pas pertinentes dans le cadre de notre projet.

1. Figures imposées communes à tous les sujets

— **Création d'au moins quatre types d'objets avec variables d'instance** – *Implémentée*
Le projet comprend les classes suivantes : `Table`, `Joueur`, `Plateau`, `CarteWagon`, `CarteItineraire`, `Ville`, `Route`. Chacune dispose de plusieurs variables d'instance décrivant leur état propre.

— **Structuration du code en plusieurs modules**

  — Le code principal est actuellement regroupé dans un fichier unique : (`LesAventuriersDuRail.`
  — Le module (`Lancement.py`) permet d'entrer les paramètres de lancement du jeu
  — Le module de test unittest est implémenté dans (`Tests.py`)
  — Un module dédié au diagramme de classes (`Diag_classe.py`) ainsi qu'un autre dédié au graph logique (`graph_logique.py` facilitent la création d'images pour la rédaction du rapport.
  — Des fichiers `.tex` permettent de gérer la rédaction du rapport et de suivre l'état du projet.

— **Héritage / composition entre au moins trois types**
Le code repose principalement sur la **composition** (ex. : `Table` contient un `Plateau`, des `Joueur`, des `Pioches` contenant elle-même, des `Cartes`. Ces deux dernierent sont garante d'une notion d'héritage minimaliste dans le code.

— **Documentation et commentaires du code** – *En cours*
Des docstrings explicatifs sont en cours d'ajout pour chaque classe et méthode. Le fichier `README.md` ainsi que le GitHub permet une vue globale du projet et de son avancement/fonctionnement. Une documentation complète de ces derniers est prévue pour la livraison finale.

— **Tests unitaires (au moins 4 méthodes, 2 cas chacune)** – *À développer*

Les tests unitaires sont à formaliser. Les fonctions clés (`piocher_cartes_wagon`, `capturer_route`, etc.) ont été testées via le module `Tests.py`. Ce point est décrit dans un chapitre dédié du rapport.

— **Stockage de données (fichier ou BDD)** – *Implémentée via fichier JSON*
Nous avons intégré une fonctionnalité complète de sauvegarde et de chargement de partie. Celle-ci repose sur l'utilisation d'un fichier au format `JSON`, ce qui permet une sérialisation simple, lisible et portable des données de la partie. La méthode `sauvegarder` construit un dictionnaire contenant l'état des joueurs (nom, couleur, cartes en main, wagons restants, routes capturées, objectifs), ainsi que l'indice du joueur actif. Ce dictionnaire est ensuite exporté vers un fichier nommé `sauvegarde.json`.


```
{
    "joueurs": [
        {"nom": "Alice",
         "couleur": "red",
         "wagons_restants":40
         "cartes_wagon": ["yellow","pink",],
         "cartes_defi": [{"ville_depart": "New York",
                          "ville_arrivee": "Dallas",
                          "points": 11} ],
         "routes_capturees": [{"ville1": "New York",
                               "ville2": "Pittsburgh",
                               "couleur": "green",
                               "longueur": 2}] },
        {"nom": "Bob",
         "couleur": "blue",
         "wagons_restants": 43,
         "cartes_wagon": ["white","blue"],
         "cartes_defi": [{"ville_depart": "Denver",
                          "ville_arrivee": "Pittsburgh",
                          "points": 12}],
         "routes_capturees": [{"ville1": "New York",
                               "ville2": "Washington",
                               "couleur": "black",
                               "longueur": 2}] }
                ],
    "joueur_actuel": 0
}
```


FIGURE 3.1 – Fichier JSON

Lors du chargement avec `charger_partie`, ce fichier est lu et interprété pour reconstituer l'intégralité de l'état du jeu : les objets joueurs sont recréés, les cartes sont restaurées sous forme d'instances, et les routes capturées sont réaffectées au bon joueur sur le plateau. Le système assure également le rétablissement du joueur actuel. Ce mécanisme de persistance de données n'était pas imposé dans les objectifs pédagogiques, mais son ajout apporte une réelle valeur pratique à l'application. Il ouvre la voie à d'autres extensions possibles, comme l'historique de parties, les sauvegardes multiples ou un tableau de scores.

2. Figures supplémentaires choisies pour le sujet

— **Algorithme d'optimisation** – *À venir*
Un joueur intelligent (objectif 4) est prévu. Il devra construire un sous-graphe reliant ses objectifs en minimisant le nombre d'arêtes ou de couleurs utilisées. Cette tâche est liée à un problème d'arbre de Steiner et constitue un bon candidat pour l'optimisation. L'algorithme de Dijkstra est aussi une piste envisagée pour connaitre le chemin le plus intéressant pour l'aide au jeu ou pour le joueur intelligent.

— **Fonction récursive** – *Implémentée*
Une fonction récursive est utilisée dans la méthode `plus_long_chemin` pour explorer toutes les routes capturées par un joueur. Elle permet de parcourir de manière exhaustive et récursive les chemins possibles dans le graphe, en calculant la longueur maximale atteignable sans revisiter les routes déjà explorées. Cette approche récursive est adaptée au problème car elle simplifie l'exploration combinatoire des arêtes capturées.
— **Exploration de graphe avec bibliothèque dédiée** – *Implémentée*
Le projet utilise `networkx` pour représenter et manipuler le graphe du plateau. Cela inclut la détection des routes disponibles, la suppression d'arêtes, l'évaluation des composantes connexes. Cette figure est donc clairement plus qu'abordée.

3. Figures non pertinentes dans le cadre de ce projet

Les figures suivantes ne sont pas jugées pertinentes dans notre cas :
— **Calcul vectoriel** : non applicable, car le projet ne traite pas de données vectorielles ou numériques en masse.
— **Design patterns type**

Chapitre 4

Tests et résultats

4.1 Méthodes testées manuellement

Au début du projet, plusieurs fonctionnalités ont été testées de manière manuelle en exécutant le programme dans des scénarios interactifs. Cela a permis de valider empiriquement les comportements attendus du jeu, comme la gestion des choix du joueur, la pioche, la capture de route ou encore l'affichage graphique du plateau.

- `jouer_tour(joueur)` : vérification que chaque tour force un joueur à effectuer une action avant de passer la main.
- `piocher_cartes_wagon(joueur)` : contrôle des règles de pioche selon que la carte visible est une locomotive ou non.
- `capturer_route(joueur)` : vérification de la condition de possession de cartes et de wagons suffisants.
- `afficher_plateau_graphique()` : vérification visuelle de l'affichage correct des routes, des villes et des wagons.

Ces tests manuels ont permis de stabiliser les fondations du projet avant de formaliser les tests automatisés.

4.2 Tests unitaires automatisés

De nombreux tests ont été ajoutés depuis la version initiale du rapport. La suite `Tests.py` couvre désormais :

- Le comportement du joueur automatique (`JoueurAuto`) ;
- La capture de route dans tous les cas (succès, échec, conditions limites) ;
- Les cartes itinéraires, y compris les cas limites (garder toutes, aucune, etc.) ;
- La validité des cartes pour une route grise ou colorée (`verifier_cartes_wagon`).

Voici un aperçu des tests déjà implémentés :

Tests sur la pioche des cartes wagon

- Vérification que la pioche visible ne contient pas trois locomotives (`verifier_visibles_sans_3_l`
- Vérification que le joueur pioche correctement deux cartes normales.
- Vérification que le joueur ne peut pas piocher une locomotive en deuxième carte.
- Test du mélange automatique de la défausse lorsque la pioche est vide.

Tests sur la pioche des cartes objectif

— **Tests unitaires (au moins 4 méthodes, 2 cas chacune)** – *Implémentée*
Huit classes de test ont été mises en place avec `unittest`. Chaque méthode du cœur du jeu (pioche, capture, validation, score) est testée dans plusieurs cas de figure, avec simulation des entrées joueur via `patch`.
— Tests des cas en cours de partie, où le joueur peut choisir librement combien de cartes il garde (minimum 1).

Tests sur la capture de routes

— Cas réussi : le joueur capture une route pour laquelle il possède les cartes nécessaires et assez de wagons.
— Cas d'échec : tentative de capture sans cartes suffisantes.
— Cas d'échec : tentative de capture sans wagons suffisants.

Ces tests permettent de s'assurer que les règles fondamentales du jeu sont correctement appliquées par le moteur. Les interactions complexes du joueur sont testées à l'aide de `unittest.mock.patch` pour simuler les entrées clavier.

Conclusion : Une suite de tests unitaires solide a été développée dans le fichier `Tests.py`, couvrant l'ensemble des règles fondamentales du jeu : pioche, capture, vérification d'objectifs, sauvegarde, plus long chemin et calcul des scores. Les tests simulent les actions des joueurs à l'aide de `mock`, garantissant un comportement conforme des fonctions clés du moteur de jeu.

Chapitre 5

Avancement des objectifs

Ce projet s'appuie sur quatre objectifs principaux décrits dans le cahier des charges. Ce chapitre fait le point sur l'état d'avancement de chacun d'eux, en mettant en évidence les éléments réalisés, partiellement réalisés ou non encore commencés.

Objectif 1 – Aide au jeu pour un humain

- **Carte du jeu modélisée comme un graphe** : *Réalisé* (avec `networkx`).
- **Gestion des routes prises** : *Réalisé*, avec mise à jour du graphe.
- **Pioche de cartes wagon** : *Fonctionnelle*, avec cartes visibles/cachées.
- **Main des joueurs (wagons et objectifs)** : *Initialisée automatiquement.*
- **Stocks de wagons** : *Réduits au fil de la partie.*
- **Liste des coups possibles** : *Partiellement réalisé*, non encore structurée explicitement.
- **Mise à jour du plateau après chaque coup** : *Opérationnelle.*
- **Fin de partie détectée** : *Non implémentée.*
- **Défausse en cas de trop de locomotives visibles** : *Non implémentée.*

Objectif 2 – Joueur aléatoire

- **Liste des coups valides** : *Partiellement réalisée*, les actions sont autorisées selon les règles.
- **Sélection aléatoire d'un coup** : *Implémentée via* `JoueurAuto`.
- **Exécution automatique d'un tour** : *Fonctionnelle.*

Objectif 3 – Calcul des points

- **Points pour les routes prises** : *Implémentés dans* `capturer_route`.
- **Vérification des objectifs réalisés (composantes connexes)** : *Partiellement fait.*
- **Calcul du plus long chemin (bonus)** : *Implémenté via méthode récursive et validé par tests.*

Objectif 4 – Joueur intelligent (IA tactique)

- **Arbre de routes pour relier les objectifs** : *Non implémenté.*
- **Réaction aux routes prises par d'autres joueurs** : *Non implémenté.*
- **Équilibrage des couleurs** : *Non implémenté.*
- **Comportement prudent / secret** : *Non implémenté.*

Objectif 4 bis – Aide à l'humain (interface)

- **Affichage graphique du plateau** : *Réalisé via* `matplotlib` (rectangles pour les routes, couleurs dynamiques, noms de villes).
- **Interface utilisateur interactive (IHM)** : *Une interface graphique complète a été développée en parallèle du moteur de jeu. Elle permet de visualiser les cartes, les routes, les objectifs et les actions du joueur de manière interactive.*
- **Indication des objectifs atteints** : *Prévue mais non encore intégrée.*
- **Alertes sur les routes critiques** : *Non encore développées.*
- **Évaluation de la faisabilité d'un objectif (Dijkstra)** : *Envisagée dans l'IA à venir.*

Tests et validations

- **Plateau USA implémenté** : *Utilisé comme terrain principal.*
- **Plateaux simplifiés (arbre, graphe complet, toutes routes grises)** : *À créer pour tests ciblés.*
- **Mode de test avec 1 seul joueur** : *Possible et utilisé dans les tests unitaires.*

Synthèse

L'objectif 1 est totalement atteint. Les objectifs 2 et 3 sont largement réalisés, avec des validations formelles. L'objectif 4 (IA tactique) reste un axe de développement pour une version plus ambitieuse du jeu. L'infrastructure actuelle (graphes, affichage, tests) forme une base robuste pour une poursuite de projet ou une industrialisation. Enfin, une IHM a été développée pour compléter le moteur textuel. Elle repose sur les classes existantes et offre une expérience utilisateur enrichie, ce qui témoigne de la modularité de notre code. Son intégration avec le cœur du moteur est en cours de finalisation pour le rendu final.

Chapitre 6

Perspectives et améliorations

Plusieurs fonctionnalités prévues dans le projet sont en cours de développement ou à finaliser dans la suite du travail. Certaines améliorations à court terme sont aussi envisagées pour renforcer la jouabilité et la clarté du programme.

Amélioration des tests unitaires

Une première suite de tests unitaires a été développée dans le fichier `Tests.py`, couvrant notamment la pioche des cartes wagon et la sélection des cartes objectif. Ces tests utilisent `unittest` avec des entrées simulées pour valider les règles de pioche, y compris la contrainte sur les locomotives.

Dans les prochaines étapes, il est prévu de :

— Étendre les tests à la capture de routes et à la vérification du respect des règles de prise de carte.
— Tester le calcul des scores (routes, objectifs, bonus de route la plus longue).
— Couvrir les cas de fin de partie et la validité des choix du joueur automatique.

Perfectionnement de l'affichage du plateau

L'affichage graphique du plateau a été considérablement amélioré : les routes sont maintenant représentées par des rectangles alignés (un par wagon), orientés entre les villes. Les routes capturées sont remplies avec la couleur du joueur. Les routes doubles sont décalées latéralement.

Des ajustements restent à faire pour améliorer :

— la lisibilité des noms de ville (ajout de fond blanc, repositionnement dynamique) ;
— la différenciation visuelle des routes capturées vs. disponibles ;
— l'échelle et le centrage du graphe pour éviter l'encombrement des zones denses.

Implémentation des joueurs aléatoires

Le projet intègre un joueur automatique fonctionnel, de type aléatoire. Il est capable de choisir des actions valides à chaque tour : pioche de cartes, capture de routes, ou sélection d'objectifs selon les règles du jeu. Cette automatisation permet de simuler des parties sans intervention humaine, et de valider les règles du jeu de manière plus fluide.

Ce joueur automatique constitue une première étape vers une intelligence artificielle plus stratégique. Il a également permis de tester l'intégration des règles complexes (pioches interdites, priorités d'action) dans un cadre automatisé.

Interface graphique et aide au joueur (IHM)

L'interface actuelle repose sur des impressions terminal et un affichage graphique avec `matplotlib`. Une extension vers une IHM plus interactive est prévue pour la dernière phase du projet.

Plusieurs pistes sont envisagées :

— afficher dynamiquement l'état du joueur (cartes, wagons, objectifs) ;
— permettre de cliquer sur les routes ou cartes plutôt que taper des entrées ;
— intégrer une forme d'aide au joueur humain (alerte si un objectif devient impossible, rappel des objectifs restants).

Une base est posée via l'affichage graphique, et l'évolution vers une IHM complète pourrait être amorcée avec des bibliothèques comme `Tkinter` ou `PyQt5`.

Conclusion

Ce projet nous a permis de mettre en œuvre, dans un cadre concret et progressif, les compétences acquises en programmation orientée objet, en structuration de projet Python et en manipulation de structures de données comme les graphes.

L'ensemble des mécanismes fondamentaux du jeu *Les Aventuriers du Rail* a été implémenté : modélisation du plateau sous forme de graphe, gestion des pioches, interaction joueur–plateau, affichage graphique évolutif, et calcul des scores pour les routes capturées. La sauvegarde et le chargement de parties, réalisés via un fichier `JSON`, renforcent la robustesse de l'application et permettent de reprendre une partie à tout moment.

Le projet intègre également un joueur automatique fonctionnel, capable de simuler des tours complets selon les règles du jeu, ainsi qu'une suite de tests unitaires exhaustive garantissant la validité des règles fondamentales. Cela nous a permis de valider non seulement le fonctionnement du moteur, mais aussi sa stabilité dans des situations variées.

Nous avons été confrontés à des choix de conception concrets, à des arbitrages techniques (affichage graphique, gestion de l'état), et à la nécessité d'une structuration modulaire pour favoriser l'évolutivité. Ce projet nous a aussi permis de renforcer notre collaboration et notre capacité à gérer une application complexe à plusieurs composantes.

Enfin, les bases posées permettent d'envisager plusieurs perspectives enrichissantes : développement d'un joueur stratégique, interface interactive (IHM), ou encore amélioration de l'aide au joueur. Ces extensions pourront être explorées dans une seconde phase, à visée pédagogique ou ludique.

Annexes

Github

Projet GitHub associé : `https://github.com/LancelotRC/Projet_Info_Lesaventuriersdurail`

Toutes les informations, fichiers sources et documents de travail sont accessibles via ce lien.

README *(rédigé en anglais)*

Le fichier `README.md`, présent à la racine du dépôt GitHub, décrit la structure du projet, les fonctionnalités principales, les instructions d'installation et d'utilisation, ainsi que les pistes d'amélioration. Il est à jour avec l'avancement présenté dans ce rapport.