Rapport S2.01-2 - K. CHARLERY, C. DARQUES, F. LEFEBVRE, groupe C

Rendu 1 (18/05) :

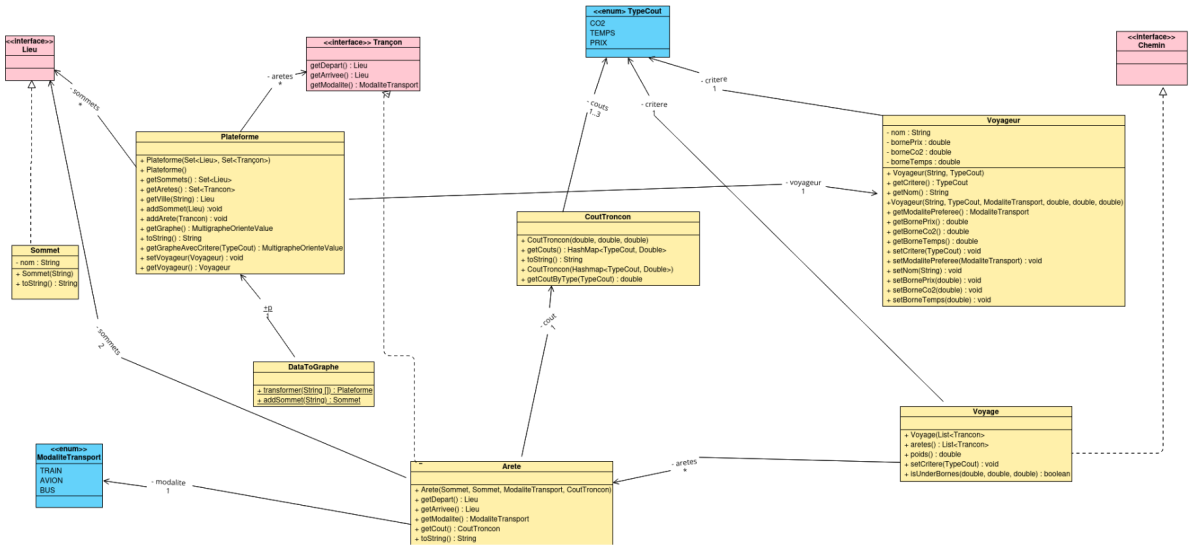
1) **Manière de lancer et utiliser l'application (position des ressources nécessaires, commande de lancement et ses options éventuelles. . .)**

Le programme ne peut fonctionner que si les librairies nécessaires sont présentes. Celles-ci se retrouvent dans le dossier **lib** du dossier.
Le code source, c'est-à-dire l'ensemble des classes et des programmes sont dans le dossier **src**.
Les tests réalisés sur nos classes se trouvent dans le dossier **test**.
Pour lancer le programme depuis la racine du dossier C1, il faut utiliser ces deux commandes :

```
javac -d bin -cp bin/*:lib/* src/*
java -cp bin:lib/* src/Main.java
```

L'application est utilisable par un utilisateur unique à chaque utilisation. Ici, la classe Main qui représente le programme principal utilise des données déjà entrées en tant qu'exemple et il n'y a pas d'informations à entrer par l'utilisateur.
Le fichier backup.java n'a pas d'utilité autre qu'une sauvegarde pour nous.

2) **Un diagramme UML de vos classes, accompagné d'une réflexion sur les mécanismes objets vus en cours que vous avez mis en œuvre et leurs intérêts le cas échéant. Si jamais une restructuration a été nécessaire pour passer à la version suivante, vous devez justifier pourquoi (et ce qui n'avait pas été pris en compte)**



Lien vers le diagramme UML de la version 1

Il y a essentiellement des mécanismes **d'encapsulation** dans ce diagramme, cependant **pas d'héritage**. L'encapsulation nous permet ici d'utiliser au mieux les objets que nous créons, ce qui est crucial car ils sont tous plus ou moins interdépendants. Par exemple, un objet Arete a besoin de Lieu, de ModaliteTransport, de Tronçon, CoutTroncon et de Voyage pour exister. Tous les objets ont des liens entre eux pour que le programme soit logique et fluide à comprendre.

Les interfaces dont nous disposons sont toutes **implémentées** par une classe.

Nous utilisons à plusieurs reprises des **surcharges** de méthodes pour permettre plus de liberté et de facilité d'utilisation s'il y a un manque de paramètres donnés. Par exemple, dans la classe Plateforme, on a deux constructeurs dont un sans paramètre, qui crée une Plateforme vide à partir d'aucune donnée. L'autre crée une Plateforme selon les paramètres donnés et attribue des valeurs neutralisantes si besoin.

Il faudra sûrement modifier certains constructeurs pour permettre d'utiliser le **chaînage**, ce qui n'a pas été le cas dans cette première version.

3) **une analyse technique et critique de l'implémentation des fonctionnalités demandées (cf Section 3.1)**

Nous avons créé plusieurs classes selon les fonctionnalités demandées :
a) La classe **Voyageur** représente l'utilisateur de l'application
Elle possède 2 attributs :

- **nom** le nom de l'utilisateur de type String
- **critere** le critère à optimiser dans la comparaison des voyages de type TypeCout (classe enum)

b) La classe **Plateforme** représente le réseau de transport constitué de :

- villes, représentées par l'attribut **sommets**
- et de tronçons reliant ces villes, représentés par l'attribut **aretes**

c) La classe **Arete** implémente l'interface **Tronçon** pour représenter les tronçons en tant qu'objets caractérisés par :

- une ville de départ représentée par **sDep** de type Sommet
- une ville d'arrivée représentée par **sArr** de type Sommet
- une modalité de transport représentée par **modalite** de type ModaliteTransport (classe enum)
- un coût représenté par **cout** de type CoutTroncon

d) Un voyage étant donc constitué d'un ensemble de tronçons, la classe **Voyage** implémente l'interface **Chemin** et possède les attributs :

- **aretes** l'ensemble des objets Tronçon du Voyage (c'est-à-dire les portions du chemin parcouru d'une ville à une autre lors du voyage)
- **critere** de type TypeCout qui représente le critère optimisé lors du voyage

e) L'énumération **ModaliteTransport** représente les différentes modalités de transport possibles : TRAIN, AVION, BUS
f) L'énumération **TypeCout** représente les différentes manières d'évaluer un tronçon lors d'un voyage : CO2, TEMPS, PRIX. Elle représente aussi le critère d'optimisation choisie par le voyageur
g) La classe **CoutTroncon** représente le coût d'un tronçon par critère d'optimisation et a comme attribut :

- **couts** qui est une table associant le critère (cf TypeCout) et le coût par critère de chaque tronçon

h) La classe **Sommet** implémente l'interface Lieu et sert à modéliser les villes avec l'attribut **nom** de type String qui représente le nom du sommet du réseau
i) La classe **DataToGraphe** permet de transformer les données du réseau de transport de tableau de chaînes de caractères en Plateforme. Elle a deux attributs statiques :

- **p** qui représente la plateforme (cf Plateforme) utilisée pour modéliser le réseau/graphe
- **map** qui est une liste associative de sommets et de leur nom qui permet d'ajouter un sommet à la plateforme créée

L'intérêt que ces attributs soient statiques est que nous n'avons pas à instancier un objet de DataToGraphe quand on utilise la méthode transformer() qui permet de passer de données en chaînes de caractères à un objet Plateforme, et pour que cet objet soit accessible par les autres méthodes.

4) une analyse quantitative/qualitative des tests que vous avez réalisés, en rapport aux notions vues en cours

La majorité des tests réalisés ne sont que des **tests unitaires** pour vérifier le bon fonctionnement des méthodes de chaque classe avec des données initialisées dans chaque classe de test.
Les tests ne sont pas très longs et concernent essentiellement des vérifications d'affichage correct comme pour afficher les bonnes arêtes des graphes selon un critère choisi par exemple. Nous avons aussi pu tester l'intégration des données nécessaires au lancement du programme (sous forme de tableau de chaînes de caractères) et cela permet de savoir que la "première" étape du programme ne pose pas problème.
Nous avons utilisé les annotations de JUnit5 telles que **@BeforeEach** et **@Test** pour spécifier l'utilité de chaque morceau de code et qu'il y ait une certaine automatisation dans les tests dans l'initialisation des données.
Les tests ont été testés avec des valeurs neutralisantes, tel que la classe de test de Voyageur qui teste les bornes par défaut de l'utilisateur.

Nous avons essayé au mieux de n'utiliser qu'une seule assertion par test quand cela était pertinent, comme dans la classe de test Graphe qui n'affiche qu'un seul résultat par test.

Rendu 2 :

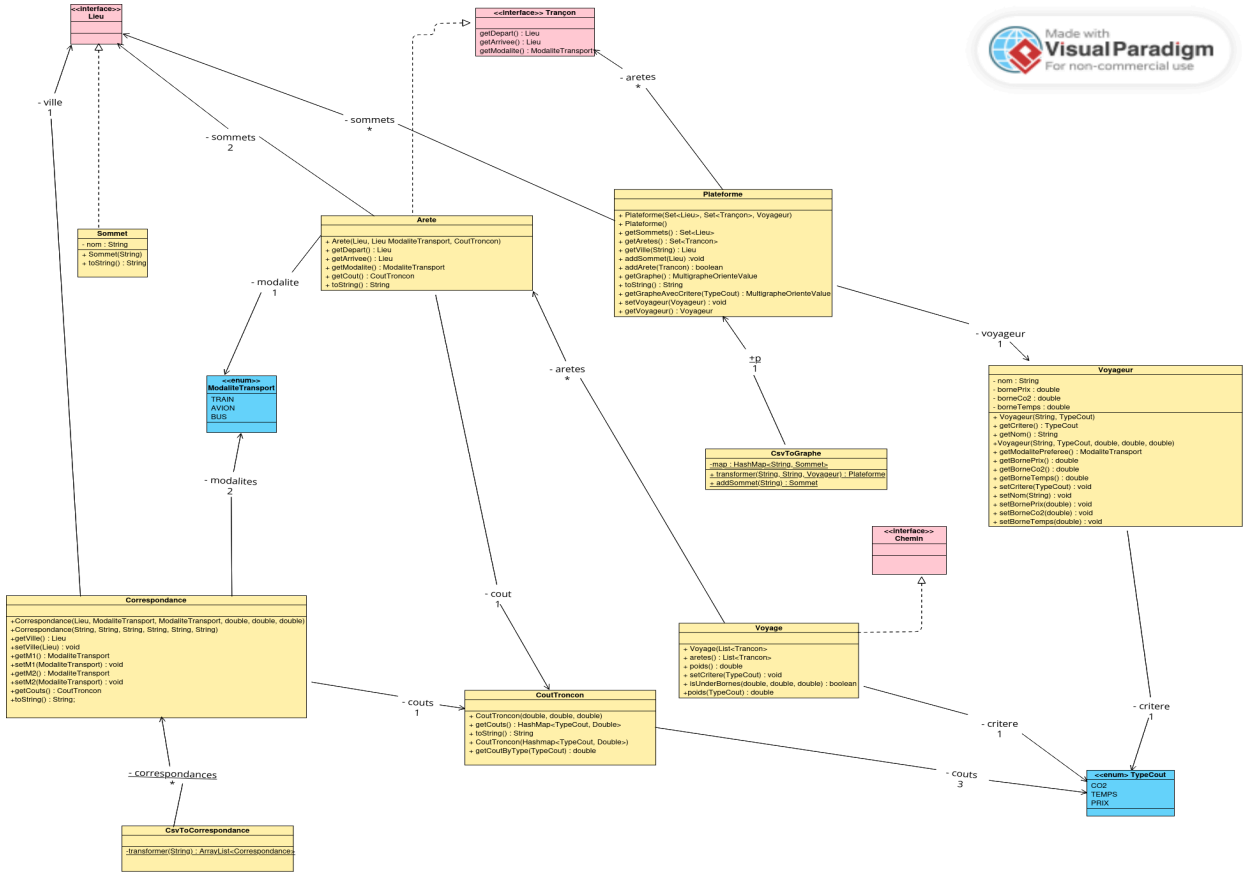
5) **Manière de lancer et utiliser l'application (position des ressources nécessaires, commande de lancement et ses options éventuelles. . .) ;**

Pour cette version, ce sont les mêmes commandes qui sont utilisées pour lancer et utiliser l'application, c'est-à-dire les suivantes :

```
javac -d bin -cp bin/*:lib/* src/*
java -cp bin:lib/* src/Main.java
```

Un dossier **res** a été créé afin de stocker les fichiers CSV utilisables dans le programme, c'est-à-dire le fichier *Data.csv* qui donne les informations sur les arêtes, villes qui les composent, leurs modalités et leurs coûts, et *Correspondance.csv* qui donne les informations sur les correspondances entre villes du réseau de transport, et le coût de ces correspondances.

6) **un diagramme UML de vos classes, accompagné d'une réflexion sur les mécanismes objets vus en cours que vous avez mis en œuvre et leurs intérêts le cas échéant. Si jamais une restructuration a été nécessaire pour passer à la version suivante, vous devez justifier pourquoi (et ce qui n'avait pas été pris en compte) ;**



Lien vers le diagramme UML de la version 2

Ce deuxième diagramme UML a subi quelques changements par rapport à sa première version.
En effet, les classes Correspondances et CsvToCorrespondance sont apparues, afin de gérer les correspondances dans les voyages des utilisateurs.
CsvToCorrespondance **encapsule** Correspondance, qui elle-même encapsule Sommet. Correspondance utilise la **surcharge** pour différents appels de ses constructeurs (un entièrement paramétré et un deuxième où tous les paramètres sont sous forme de chaînes de caractères).
La classe DataToGraphe est devenue CsvToGraphe, étant donné que nous avons modifié la manière d'initialiser les données : au lieu de les écrire en dur dans la méthode **statique** transformer(), elles viennent à présent d'un fichier CSV qui est lu et trié par le programme.
Quelques signatures de méthode ont également été modifiées telles que le constructeur de Voyageur et de Plateforme. En effet, corriger les incohérences de la version 1 ont impliqué des modifications de méthodes, et des restructurations des signatures.
La classe Voyageur utilise le **chaînage** dans ses constructeurs : le premier constructeur est entièrement paramétré, le deuxième appelle le premier en n'ayant que deux paramètres sur cinq.

7) **une analyse technique et critique de l'implémentation des fonctionnalités demandées (cf Section 3.1)**

Nous avons ajouté une classe **Correspondance** afin de modéliser les changements de modalités de transport entre les villes d'un voyage. Une correspondance est donc représentée par :

- une **ville** de type Lieu
- deux modalités de transport **m1** et **m2** (une pour chaque ville du tronçon du trajet)

Nous avions à l'origine ajouté 3 attributs à la classe : 3 doubles pour le temps, le prix et les émissions de CO2. Au lieu de mettre les 3, nous les avons regroupé dans :

- **couts**, un objet CoutTronçon qui représente le temps en minutes du trajet, son prix et la quantité de CO2 émise au cours du trajet du tronçon

Elle contient les accesseurs et mutateurs classiques pour ses attributs.

Aussi, nous avons créé une classe **CsvToCorrespondance** qui permet de créer des correspondances à partir d'un fichier CSV regroupant les informations sur les correspondances possibles. Elle a comme attribut :

- **correspondances**, une ArrayList d'objets Correspondance et retourne cette liste après l'avoir transformée, c'est-à-dire après l'avoir remplie des données issues du fichier CSV.

La classe contient une méthode **transformer()** qui permet de renvoyer la liste des correspondances sous leur forme d'objet, à partir du fichier Correspondance.csv qui

transforme les données textuelles en objet Correspondance comme expliqué ci-dessus.

La classe **CsvToGraphe** utilise deux attributs :

- **p**, une Plateforme qui représente le réseau sur lequel l'utilisateur voyage
- **map**, une liste associative qui associe un nom de ville à un sommet du graphe

Elle contient également une méthode **transformer()**, qui permet de transformer les données fournies dans un fichier CSV grâce à la **gestion de flux** textuels. Elle renvoie alors un objet Plateforme qui représente le réseau de transport dans lequel on prend en compte les correspondances nécessaires au voyage de l'utilisateur.

La classe Algorithme, dont nous n'avions pas parlé en version 1 car il s'agissait uniquement d'une classe permettant de tester un scénario, permet à présent un affichage correct du réseau grâce à la méthode **afficher()**. À partir d'une liste des chemins les plus courts/optimisés pour l'utilisateur, la méthode affiche le numéro du chemin et les sommets parcourus avec leurs modalités ainsi que le coût total pour le critère choisi par l'utilisateur. Cette méthode n'affiche pas les villes quand la modalité reste la même, et n'affiche que les correspondances : il n'y a pas d'intérêt à indiquer la ville traversée à l'utilisateur s'il n'y change pas de modalité de transport.

8) **une analyse quantitative/qualitative des tests que vous avez réalisés, en rapport aux notions vues en cours**

Les nouvelles classes de test créées sont CorrespondanceTest, CsvToGrapheTest, et nous avons également modifié GrapheTest.
Les tests pour les correspondances sont encore une fois essentiellement **unitaires** pour tester les méthodes tostring(), l'accesseur des coûts de la correspondance triés par type… Nous avons également testé le constructeur ne prenant que des chaînes de caractères en paramètres afin de s'assurer qu'il fonctionnait.

Le test de CsvToGraphe consiste en beaucoup d'initialisation car il s'agit d'un **test d'intégration** : on vérifie si tous les scénarios d'exécution seraient pris en charge une fois le programme lancé (valeurs vides ou non renseignées, par exemple). Quelques ajouts ont été faits spécialement pour ces tests : on tri le résultat avant de l'afficher et le comparer avec ce qu'on attend, car la liste n'étant pas ordonnée le test peut s'afficher faux alors qu'il fonctionne, mais les valeurs ne sont juste pas dans le même ordre.

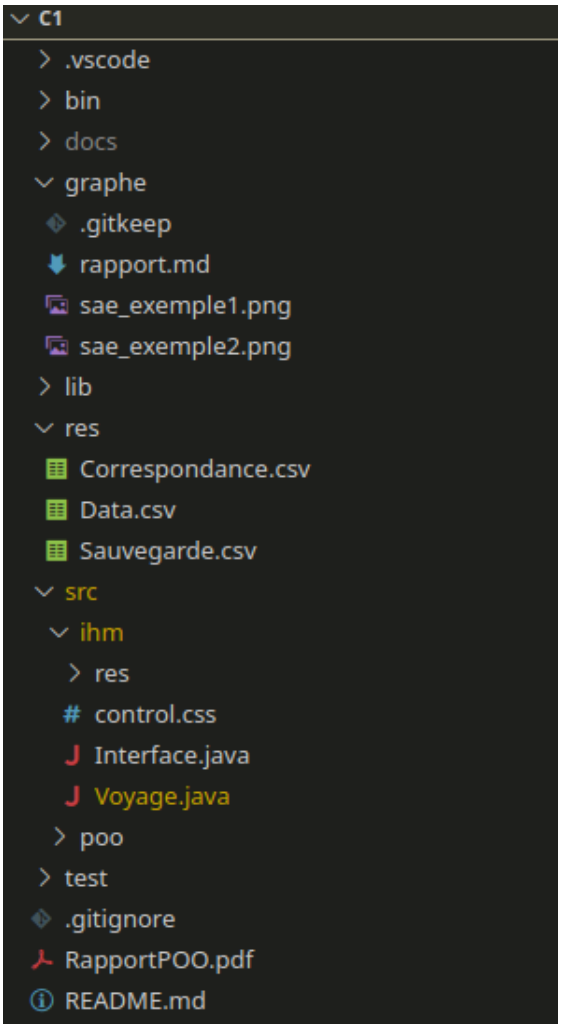
Quant à GrapheTest, il s'agit d'un test d'un scénario qui permet de vérifier l'affichage des graphes. Nous avons donc utilisé des **tests unitaires** pour tester notre fonction d'affichage.

Bien sûr, les mêmes conventions d'écriture de JUnit5 ont été utilisées.

Rendu 3 :

9) **Manière de lancer et utiliser l'application (position des ressources nécessaires, commande de lancement et ses options éventuelles. . .) ;**

La hiérarchie et structuration des ressources a été modifiée pour plus de clarté. Nous avons donc un dossier spécifique à chaque matière prise en compte dans cette SAé :



- le dossier **graphe** reste à la racine et contient le rapport de Graphes et ses images ressources
- un dossier **ihm** a été créé dans le répertoire ***src*** et contient les fichiers permettant de lancer l'application en JavaFX ainsi qu'un sous-dossier **res** qui contient des icônes ressources pour la mise en page de l'application
- un dossier **poo** a été créé dans le répertoire ***src*** et contient toutes les classes déjà écrites précédemment afin de mieux s'y retrouver

Aussi, dans le répertoire ***res***, nous avons créé un fichier Sauvegarde.csv qui contient les données sauvegardées des trajets effectués par l'utilisateur (cf. Q11. Sauvegarde ci-dessous).

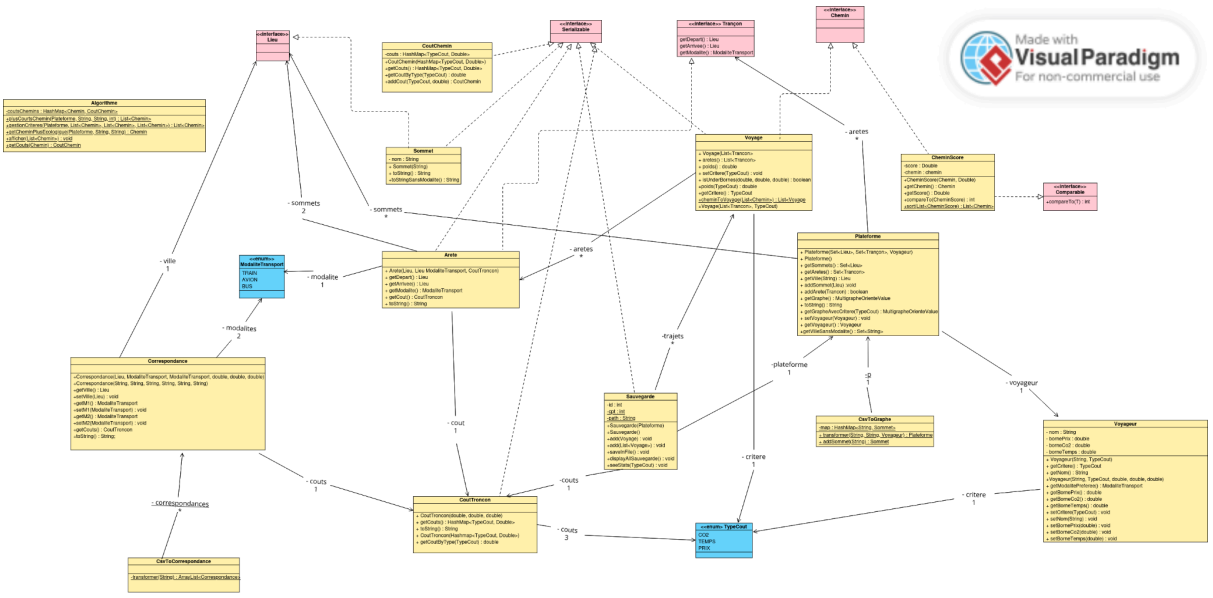
Pour lancer l'application avec son interface graphique, il suffit d'ouvrir le fichier `src/ihm/Interface.java` et de lancer le programme en cliquant sur le bouton "run java".

/!\ Dû à un problème de .gitignore qui n'ignore pas le dossier .vscode, il sera peut-être nécessaire de modifier le fichier launch.json de ce dossier pour indiquer les chemins vers le SDK de JavaFX.

Certaines fonctionnalités n'ont pas pu être implémentées par manque de temps, alors un scénario d'utilisation d'exemple a été codé dans le programme principal qui se lance grâce aux commandes :

```
javac -d bin -cp bin:lib/* src/*
java -cp bin:lib/* src/Main.java
```

10) **Un diagramme UML de vos classes, accompagné d'une réflexion sur les mécanismes objets vus en cours que vous avez mis en œuvre et leurs intérêts le cas échéant. Si jamais une restructuration a été nécessaire pour passer à la version suivante, vous devez justifier pourquoi (et ce qui n'avait pas été pris en compte) ;**



[Lien vers le diagramme UML de la version 3]

Jusqu'alors, la classe Algorithme n'était pas affichée dans l'UML car elle ne servait que de classe de scénario d'utilisation. Ses méthodes sont devenues nécessaires au programme et elle a donc été ajoutée au diagramme, même si elle n'est pas reliée aux autres classes.

Nous avons dû faire quelques changements de structuration suite à l'implémentation d'interfaces dans certaines classes comme Serializable.

11) **une analyse technique et critique de l'implémentation des fonctionnalités demandées (cf Section 3.1)**

Plusieurs ajouts et modifications ont été implémentés dans cette dernière version du programme.

Dans la classe Algorithme :

- une des consignes de la version 3 était de permettre une sélection multi-critères dans les trajets de l'utilisateur. Pour cela, nous avons ajouté une méthode **gestionCriteres()** qui renvoie une liste de Chemin avec tous les critères souhaités
- la méthode **plusCourtsChemins()** a été modifiée pour permettre la gestion multi-critère des trajets en créant une liste de Chemin qui prend en compte les trois critères
- la méthode **getCheminPlusEcologique()** a également été ajoutée pour retourner le chemin avec le moins de CO2 parmi ceux proposés à l'utilisateur

- la méthode **getCouts()** a été ajoutée aussi pour voir les critères du chemin passé en paramètre
- la méthode statique **afficher()** permet d'afficher les informations relatives aux chemins possibles

Dans la classe Voyageur :

- l'attribut **nom** a été supprimé car il a été jugé inutile puisqu'il n'était jamais pertinent dans le code, voire pas utilisé
- l'attribut **criteres** a été modifié de TypeCout à List<TypeCout> afin de pouvoir en ajouter plusieurs au même trajet, en rapport avec la gestion multi-critères
- un constructeur a été ajouté avec du **chaînage** pour utiliser autrement l'instanciation de Voyageur

Dans la classe Sommet :

- nous avons ajouté la méthode **toStringSansModalite()** pour permettre un autre type d'affichage moins complet

Dans la classe Plateforme :

- dans la même idée que Sommet, la méthode **getVilleSansModalite()** a été ajoutée pour ne récupérer que la ville d'une instance

Dans la classe Voyage :

- la méthode **getCritere()** n'avait pas été implémentée et elle s'est avérée nécessaire, d'où sa création
- nous avons ajouté un constructeur qui prend en compte un critère pour créer un Voyage avec un paramètre TypeCout
- la méthode statique **cheminToVoyage()** a été créée pour permettre de transformer les Chemins en Voyage. Cette méthode a été cruciale au reste du développement car l'interface Chemin n'étant pas sérialisable, la sauvegarde n'était pas possible et nous avons dû contourner ce problème en transformant les Chemins générés en Voyage

En parlant de **Sauvegarde**, c'est une classe que nous avons créé afin de pouvoir générer un fichier de sauvegarde des trajets de l'utilisateur.
Elle a comme attributs :

- **trajets**, une liste de Voyage qui représente les trajets de l'utilisateur
- **couts**, un CoutTroncon qui permet d'associer un type de coût à son total en poids
- **plateforme**, la Plateforme qui représente le réseau de transport
- **id** qui est l'identifiant donné à chaque trajet
- **cpt** qui est statique et permet d'incrémenter l'identifiant et a le rôle de numéro automatique
- **path**, lui aussi statique et qui représente le chemin vers le fichier de sauvegarde en chaîne de caractères

Sauvegarde a deux constructeurs, un vide et un qui prend la plateforme en paramètre. Elle a deux méthodes add() pour ajouter à la liste de trajets soit un voyage soit une liste de voyages.
Ses méthodes les plus importantes sont :

- **saveInFile()** qui permet de ***sérialiser*** sous forme textuelle les trajets passés dans son attribut trajets dans un fichier CSV
- **displayAllSauvegarde()** qui affiche le contenu du fichier CSV sous forme textuelle
- **seeStats()** qui prend un critère en paramètre et permet de voir la moyenne de ce critère sur l'ensemble des trajets de l'utilisateur

Nous avons également créé une classe **CheminScore**, qui permet d'associer un total de poids à un trajet, qui a pour attributs :

- **chemin** qui représente le chemin de l'ensemble du trajet
- **score** de type Double qui prend la valeur du coût de l'ensemble du trajet

CheminScore a un constructeur qui prend ses deux attributs en paramètres et les accesseurs de ces attributs. Elle comporte aussi :

- une méthode **compareTo()** qui compare l'objet CheminScore a un autre, grâce à l'implémentation de ***Comparable***
- une méthode **sort()** qui trie une liste de Chemin passée en paramètre selon la méthode compareTo().

Cette troisième version a également nécessité l'implémentation de ***Serializable*** dans Sommet, Arete, CoutTroncon et Voyage afin de pouvoir sérialiser les trajets et utiliser convenablement la classe Sauvegarde.

12) Une analyse quantitative/qualitative des tests que vous avez réalisés, en rapport aux notions vues en cours

Il n'y a pas eu de classe test pour la classe Sauvegarde car le code ayant déjà été écrit pour le reste du programme, il suffisait de compiler et exécuter celui-ci pour voir si la classe Sauvegarde fonctionnait ou non. Nous n'avons pas jugé nécessaire d'en créer une distincte car tout se testait au fur et à mesure des corrections apportées aux méthodes de la classe.
Les tests de classe Voyage ont été modifiés pour vérifier que les modifications de la classe fonctionnaient correctement.
Par raison de temps limité, pas d'autres tests n'ont été effectués autrement que directement dans la classe principale qui utilisait directement les méthodes implémentées.