Projet PPO

Rapport final

Elèves

Enseignants

2022/2023

Masson Kévin
Roquand Tom

B. Carré
B. Cazaux
J-B. Quilliot
F. Hoogstoel

Sommaire

Introduction

Le projet suivant consiste à développer une application « EasySki » afin de permettre aux utilisateurs de trouver le chemin le plus rapide entre deux points donnés de la station, le tout via différents moyens de déplacement : par les pistes, les remontées ou les navettes. Cette application devra permettre de calculer le temps de trajet de deux différentes manières : soit le temps absolu (ou le temps idéal), soit le temps réel (qui lui prend en compte l'attente aux remontées, le temps que le bus arrive, … bref tous les éléments aléatoires qui rallongent le trajet.

Cette application sera développée sous Java et utilisera comme algorithme de plus court chemin la méthode de Dijkstra. Au cours de ce premier rapport, nous allons étudier les classes, les différentes méthodes et structures de données qui seront nécessaires à la réalisation et au bon fonctionnement de cette application.

Partie 1 : Analyse et conception

1- Analyse générale et justification des classes nécessaires

Les éléments clés de ce projet sont donc les différents points de repère sur la carte, ainsi que tous les itinéraires qui existent entre les différents points. Ce sont ces deux éléments qui vont définir les deux classes principales. Ces deux classes seront reliées à une classe station qui reliera le tout pour permettre une meilleure visibilité concernant les points et les transitions. Ainsi depuis la classe Station nous aurons accès à la liste des points et des transitions.

Aussi pour répondre au problème principal qui est de trouver le plus court chemin, une classe Dijkstra sera créé pour étudier l'itinéraire et trouver le PCC entre deux points d'une station. Tous ces éléments seront gérés et appelés dans un package application, qui servira à interagir avec l'utilisateur dans la console puis dans un package ihm (interface homme-machine) qui traitera toute la partie graphique.

La classe Point

La classe point correspond à un lieu sur la zone étudiée.

Données membres : numéro, nom et altitude associée

Sous-classe PointCoord

En regardant le fichier XML, on s'est rendu compte que certains points avaient des coordonnées associées, celles-ci ne seront pas exploitées dans la recherche de plus court chemin, on pourra donc appliquer toutes les fonctions de Point sur PointCoord.

Données membres : coordonnées x et y réelles.

La classe Transition

Une transition correspond donc à un chemin entre deux points donnés.

Données membres : numéro, nom, point de départ, point d'arrivée

On peut imaginer le cas où l'on se retrouve avec deux transitions identiques. On choisira alors aléatoirement celle que l'on sélectionne.

Sous-classe Descente

Les descentes sont les pistes de ski

Données membres : niveau de difficulté, temps moyen de descente/100m

Sous-classe Remontée

Correspond aux remontées mécaniques

Données membres : type, durée contrôle/installation, temps de montée/100m

Sous-classe Navettes

Données membres : type, durée trajet

La classe Station

Elle va simplement représenter la tête de la structure de la station de ski faite des transitions et des points. Elle va également définir si on étudie le temps de façon réelle ou absolue.

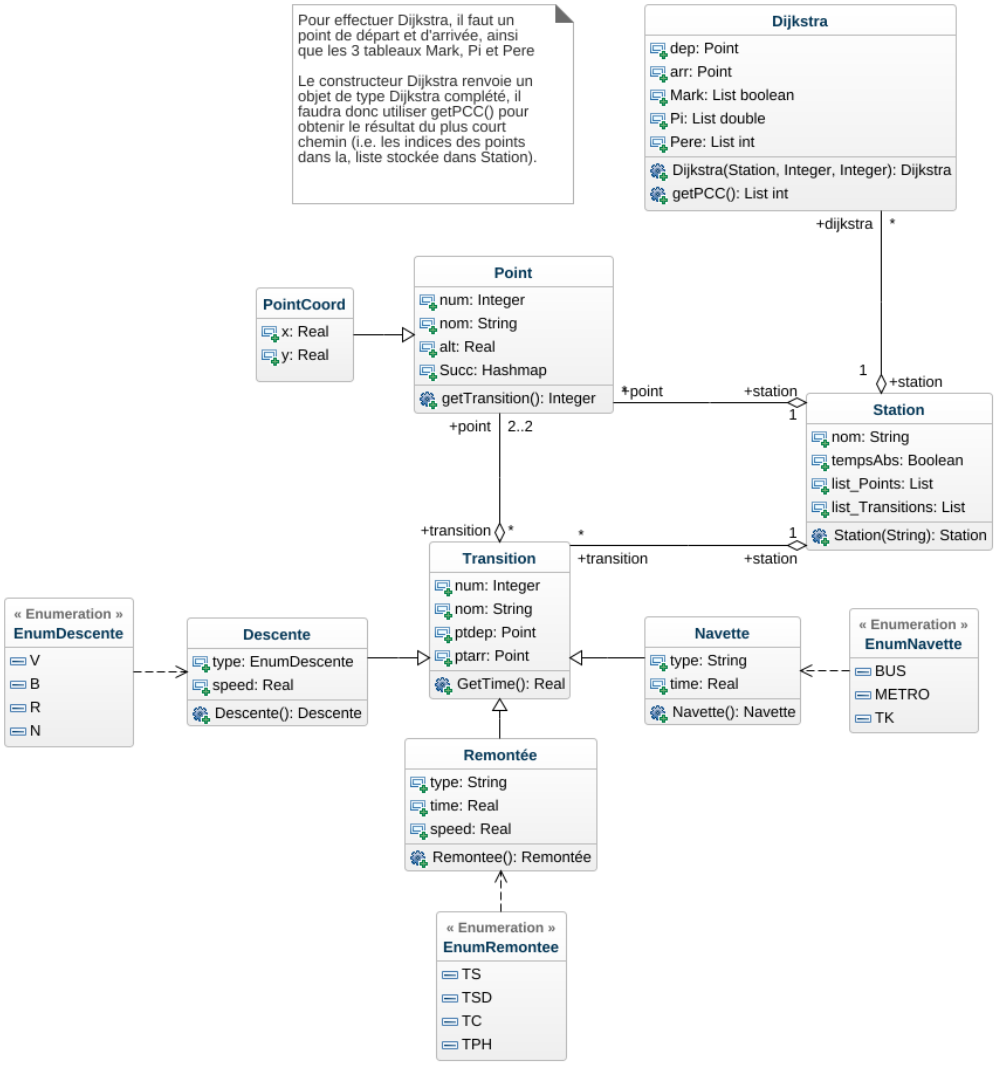
Données membres : nom, type_temps, liste des Points, liste des Transitions

La classe Dijkstra

Cette classe va nous permettre d'utiliser Dijkstra. Cette méthode étant extérieure à nos données, créer une classe nous permettra de gérer les tableaux Pi, Succ et Mark de manière globale pour utiliser les différentes fonctions dont nous avons besoin pour trouver notre PCC.

Données membres : Station, point départ, point arrivée, tableaux Pi, Mark, Succ

2- Diagramme UML



Rq : Pour un souci de lisibilité, nous ne précisons que les constructeurs et les méthodes les plus importantes.

3- Choix de la structure de données

Afin de stocker tous les trajets possibles, nous allons utiliser un hashmap nommé Succ contenant donc comme indiqué tous les successeurs de chaque points et la transition faisant la correspondance. On va donc appliquer l'algorithme dans la classe Dijkstra sur deux points de la station. Une fonction getPCC() définie dans Dijkstra nous permettra d'obtenir le plus court chemin pour un objet de type Dijkstra.

En passant par des matrices, il aurait été compliqué de se repérer dans les indices. La méthode est réalisable mais serait trop grosse en mémoire. En termes de complexité, celle-ci serait également de complexité n^2 dû à la dimension de la matrice, tandis qu'en restant sur des listes, on reste sur une n-complexité. Le dernier problème concerne seulement la réalisation et la lecture du programme, le risque étant de se perdre dans les indices.

L'application des structures de données étudiées dans le cadre du module PPO semble être la solution la plus adaptée ici.

Déclaration des principales SD

`List<Point> Lpoint = new ArrayList<>();` → `dans Station`

`List<Transition> Ltransition = new ArrayList<>() ;` → `dans Station`

`Map<Point, Transition> Succ = new HashMap<>();` → `dans Points (un point = une map)`

4- Principales méthodes

Gestion des cas particuliers

On a plusieurs cas particuliers à traiter. Premièrement, si on ne trouve pas de trajet, alors il faudra que le programme lève une exception. Cette exception sert à interrompre le programme pour que l'utilisateur n'attende pas en vain son itinéraire, sachant qu'il n'existe pas (ou que notre méthode ne le trouve pas). Une autre exception sera créée pour traiter le cas où l'utilisateur saisit des points inconnus.

On a également le cas où il y aurait plusieurs plus courts chemins, alors on choisira arbitrairement. Cependant on pourrait imaginer une évolution de ce projet proposant les chemins à l'utilisateur, lui disant combien de chemins sont possibles pour un temps égal… Il sera cependant difficile de mettre en place ces évolutions durant ce projet, pour une question de temps.

Il faudra également penser au problème dans la lecture des données, qui nous donnerait un graphe non-connexe, ainsi cela rejoint le premier problème et aucun chemin n'existerait. On peut donc imaginer la création d'une autre exception, étudiant la connexité d'un graphe (la non-connexité ne posant pas tout le temps problème…).

Méthode Dijkstra (constructeur)

Données :

```
  P_depart, P_arrivee le point de départ respectivement d'arrivée de
type Point
```

Variables locales :

```
  Mark : tableau binaire de taille le nombre de Points
  P : point occurent
```

Résultat :

```
  PI : tableau réel de taille le nombre de Points
       Représentant les temps de transitions
  Pere : tableau d'entiers de taille le nombre de Points
       Associé au père de chaque Point
```

```
{initialisation}
Mark ← FAUX
PI ← +inf
Pere ←0
Pere[P_depart.getID()] ← P_depart.getID()
{Le point de départ est par définition son propre père}
Pi[P_depart.getID()] ← 0
{Le chemin entre le point de départ et le point de départ}

{traitement}
Mark <- FAUX
PI <- +inf
Pere <- 0

Pere[P_depart.getID()] <- P_depart.getID()
{Le point de départ est par définition son propre père}
Pi[P_depart.getID()] <- 0
{Le chemin entre le point de départ et le point de départ}
tant que (Mark[P_arrivee.getID()] == FAUX) :
{tant que le point d'arrivée n'est pas marqué}
  choisir P tq Pi[P] soit min et non marqué
  Mark[ID.P] ← VRAI
  pour tous les successeurs non-marques de P :
    {On attribue les nouvelles valeurs de Pi}
    si (Pi[ID.P] + GetDistance(P, Succ)) < (Pi[Succ.getID()])
      Pi[Succ.getID()] ← Pi[ID.P] + GetDistance(P, Succ)
      Pere[Succ.getID()] ← ID.P
    fsi
  fpour
ftantque
```

Rq : la fonction GetDistance ne sera pas détaillée ici mais va simplement renvoyer le temps de trajet entre un Point et son Successeur.

Méthode getPCC

{initialisation}

```
i ← 0
j ← 0
Parr ← P_arrivee
cost ← PI[P_arrivee.getID()]
```

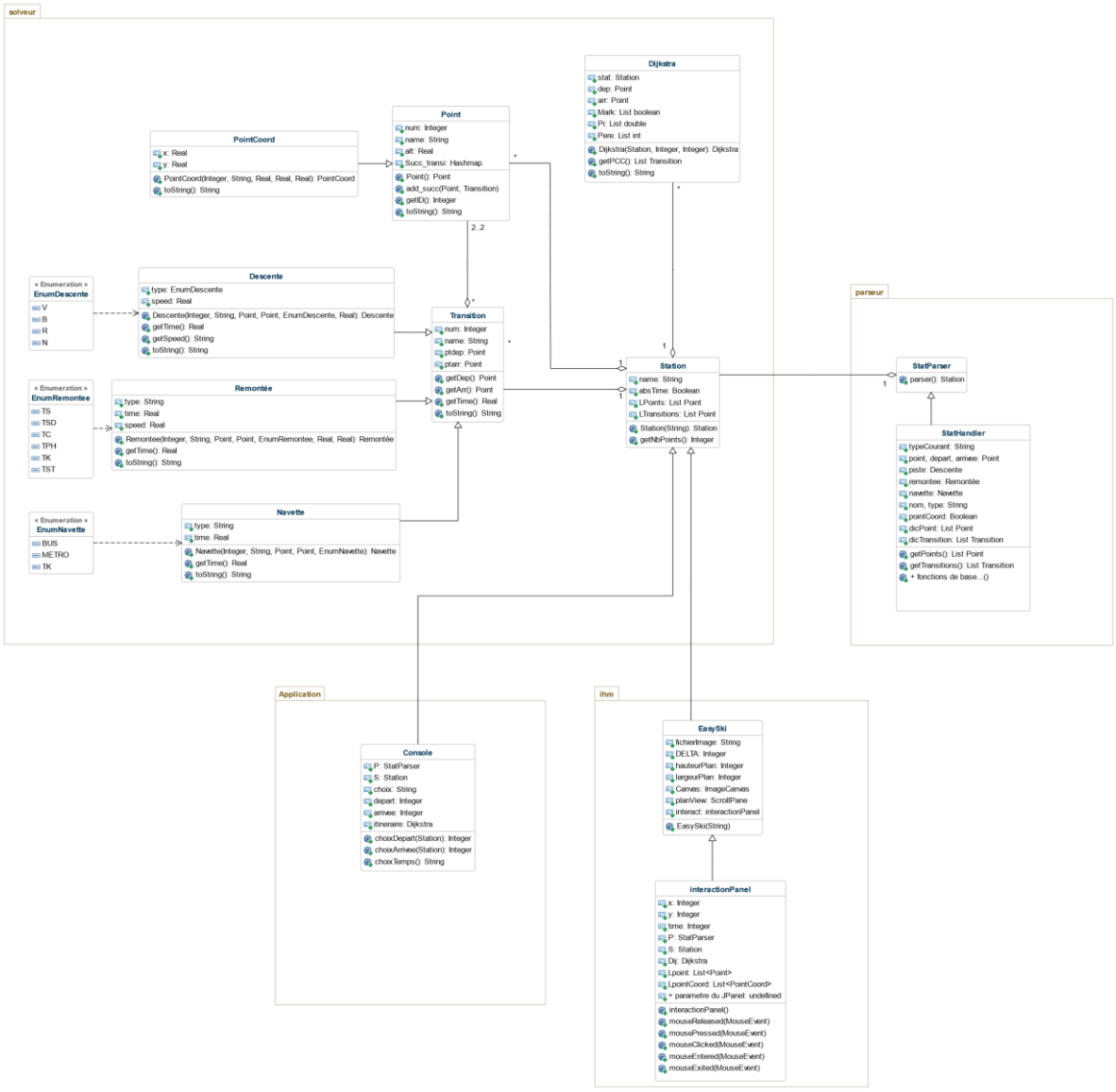
{traitement}

```
tant que (Parr.getID()≠ P_depart.getID()) {On parcourt la liste PI jusqu'à ce qu'on arrive sur le point de départ}
  Pdep ←  Pere[Parr.getID()]
  UnsettledPCC[i] ← getTransition(Pdep.getID(),Parr.getID()) {getTransition récupère le num de la transition entre le point de départ et le point d'arrivé}
  Parr ←  Pdep {Le point d'arrivé devient le point de départ de la boucle suivante}
  i ← i + 1
ftantque

tant que (i ≠ -1) {On inverse la Liste de transition}
  PCC[j] ←  UnsettledPCC[i]
  i ← i-1
  j ← j+1
ftantque
```

Rq : On trouvera la transition la plus rapide s'il en existe plusieurs avec un simple test de comparaison.

Partie 2 : Description de la structure finale



La structure finale de ce projet se différencie de la première partie par l'ajout de 3 nouveaux packages gérant la lecture des données puis la partie interactions programme/utilisateur :

1) Le package Parseur

Ce package est composé de deux classes : StatParser et StatHandler. L'objectif est de pouvoir lire séquentiellement le fichier station.xml. Une fois la lecture effectuée, les données reconnues sont traitées afin de pouvoir les ajouter à nos listes de points et de transitions de la classe Station, ces ajouts sont nécessaires à l'utilisation de Dijkstra.

La classe StatParser

Cette classe correspond au parseur de notre programme, elle utilise un parseur Java pour XML (apache xerces) répondant à l'API SAX. La classe est composée de la méthode parser() retournant une Station. Cette méthode va lire séquentiellement le fichier xml (ici le fichier "src/station/station.xml"), et va se connecter au ContentHandler : notre seconde classe StatHandler, traitant les différentes données reconnues. Une fois le traitement effectué et les données enregistrées dans les variables correspondantes, la méthode va retourner une station S avec les listes Lpoint et Ltransition complétées.

La classe StatParser

StatParser possède les différents attributs présents dans le fichier station.xml : numéro, x, y, altitude, tpsFixe, tpsDenivele, tpsTrajet... De plus, on a la création d'une liste de points et une liste de transitions vides. La classe possède principalement 5 méthodes importantes à détailler :

- startElement() ayant pour utilité de lire les balises ouvrantes du fichier xml : point, navette, piste, remontée.
- characters() lisant le contenue d'une balise ouvrante et va enregistrer sont contenues dans les différentes variables citées précédemment.
- endElement() qui suivant le contenue enregistré va permettre la création d'un Point, PointCoord ou bien d'une Transition, qui va ensuite être ajoutée à la liste (Point, Transition) correspondant.
- getTransitions() retournant la liste de Transition complétée.
- getPoints() retournant la liste de Point complétée

2) Le package Application

L'utilité principale d'un package Application, et ceci est valable pour la plupart des projets de développement informatique, est de pouvoir permettre au(x) développeur(s) de tester continuellement le projet et les modifications qui y sont apportées. Ce package permet ainsi de vérifier le bon fonctionnement de toutes les classes de la partie technique du projet (i.e. le solveur). Aussi cela permet de se rendre compte de certaines erreurs qui auraient pu être omises durant la création de certaines fonctions. On a à l'aboutissement de notre projet la certitude d'avoir une application java qui ne plante pas.

Le package contient dans le cas présent une seule classe Console. La classe doit être le plus simple possible et doit être le reflet de ce que le cahier des charges exige comme résultat.

On va en premier lieu créer un **objet de type StatParser** dans le main afin de lire le fichier xml, puis associer le résultat de la lecture à un objet de type **Station**.

Aussi il contiendra trois fonctions, pour respectivement **saisir le point de départ, le point d'arrivée et le type de temps**. À la suite de l'appel de ces fonctions, on pourra tout simplement créer un objet de type **Dijkstra** afin **d'afficher sur la console le plus court chemin** trouvé par notre algorithme. L'utilisateur pourra calculer autant d'itinéraires qu'il le souhaite. En prenant garde à bien attraper les exceptions qui auraient pu se propager pour une raison ou une autre. De cette manière on pourra constater de la fiabilité de notre programme.

On peut dès à présent tester les exemples du sujet et observer le bon fonctionnement global. Les résultats sont identiques à ceux donnés dans le sujet. Après avoir multiplié les tests nous ne voyons pas de manière de planter notre programme, nos exceptions semblent suffisantes. Alors il ne reste plus qu'à se pencher sur l'interface graphique, avant de pouvoir livrer la version finale de notre application java.

3) Le package ihm

Le package ihm correspond à l'aboutissement de ce projet, correspondant à l'acronyme pour « interface homme-machine », celui-ci va traiter la partie visuelle qui comme son nom l'indique sera utilisée par l'utilisateur final.

Le package est composé du fichier EasySki contenant plusieurs classes nécessaires au fonctionnement de l'interface graphique. Concernant notre utilisation, la plus importante est la classe EasySki appelant les différentes méthodes et classes gérant les interactions avec l'ordinateur : InteractionPanel.

Notre interface graphique doit permettre de sélectionner un point de départ et d'arriver à partir de la souris (en cliquant sur les différentes étoiles rouges correspondants aux points connus) ou alors d'écrire leurs noms dans les cases dédiées. Une fois la sélection des points de départ et arrivée effectuée, nous pouvons sélectionner le type de temps souhaité et ensuite appuyer sur un bouton "go" générant le plus court chemin qui est ensuite affiché dans le chat "resultTextArea".

Nous allons donc voir les 2 principales classes ainsi que les méthodes prépondérantes.

La classe EasySki

Celle-ci possède un paramètre String fichierImage correspondant à l'image que l'on souhaite afficher dans notre interface graphique. La classe va appeler les différentes méthodes et classes en liens avec l'interface graphique : ImageCanvas, ScrollPane, InteractionPanel. Cette dernière classe, est celle que nous avons utilisé et modifié afin d'obtenir le résultat souhaité pour notre projet.

La classe InteractionPanel

Cette classe consiste à toutes les interactions entre l'utilisateur et l'interface Graphique/ la machine. Elle comprend une classe SelectionSubPanel qui permet la création des zones de textes, des boutons et de leurs interactions. Les variables que nous utilisons sont : x, y, list<Point>, list<Transition>, Station S, StatParser P, Dijkstra dij, time et les variables JPanel nécessaires à la création des différentes cellules et boutons.

Dans cette classe, nous allons tout d'abord créer une liste<PointCoord> permettant de pouvoir directement utiliser lorsque l'utilisateur fourni des coordonnées d'un point (Clique ou nom). Ensuite différentes actions sont programmées :

- Clique sur "setDepart" : Les cases vont se réinitialiser et le nom du point sélectionner va s'enregistrer dans la cellule "setDepart".
- Clique sur "setArrivee" : Les cases vont se réinitialiser et le nom du point sélectionner va s'enregistrer dans la cellule "setArrivee".
- Clique sur "tpsReel" : Le temps réel est sélectionné et la variable time va donc prendre la valeur 1 correspondant à l'application du temps réel dans Dijkstra
- Clique sur "go" : Les noms du point de départ et d'arrivé sont enregistrés et leurs indices sont récupérés en trouvant dans un premier temps leurs Point (via une méthode stream + lambda) puis ensuite en utilisant la méthode getID() de la classe Point. Une fois les indices des points récupérés, on applique Dijkstra avec les différents paramètres récoltés et on retourne dans le chat "resultTextArea" le résultat du plus court chemin sous la forme souhaitée (à l'aide de la méthode toString() de la classe Dijkstra).

La méthode mouseReleased permet de pouvoir récupérer les coordonnées x et y à chaque clique souris et aussi de les afficher dans les cellules dédiées, tout en parcourant la liste PointCoord créée précédemment afin de récupérer et afficher le nom correspondant aux coordonnées de l'emplacement cliqué.

Conclusion Globale

Ce projet que nous avons réalisé fut l'occasion de s'exercer et s'améliorer sur de nombreux points. Un premier point sur la réflexion sans programmation que nous avons dû effectuer dans la première partie du projet, faites sur les différentes classes à utiliser, leurs méthodes, variables et exceptions que ces classes peuvent nécessiter. Ce projet permet d'appliquer différentes thématiques vues en cours (Exception, Hiérarchie, Stream, Parseur, Lambda..), mais aussi d'enrichir nos connaissances avec l'interaction entre les packages solveur, parseur, application et l'interface homme/machine. Effectuer ce projet nous fait prendre connaissance de l'étendu des actions que l'on peut effectuer à partir de l'application de notre cours.

Le point important que nous relevons à travers ce projet est l'intérêt d'une bonne réflexion avant d'effectuer le code. Puisque celle-ci peut permettre d'éviter de nombreuses erreurs et de garder une ligne directrice sur le déroulement du projet.

Finalement, nous tenons à remercier nos différents enseignants pour leurs implications dans l'organisation des cours magistraux, travaux pratiques et séances de projet.