Polytech Lille - IS4 - Projet de PPO

2022-2023

`Voir le Wiki : Spécialité IS > IS4 > Modules d'enseignement > PPO > Projet`

Préambule

Lire le sujet entièrement pour vous faire une idée d'ensemble du projet et de sa progression. Mise à part la phase d'analyse/conception initiale (coeur du projet), le travail qui suivra pourra judicieusement être réparti entre les membres du projet, au moins en ce qui concerne la compréhension des technologies employées telles que XML et interfaces graphiques. Des ressources sont fournies (fichier de données, portions de code) et à charger à partir du Wiki (chapitre "Ressources").

Objectif

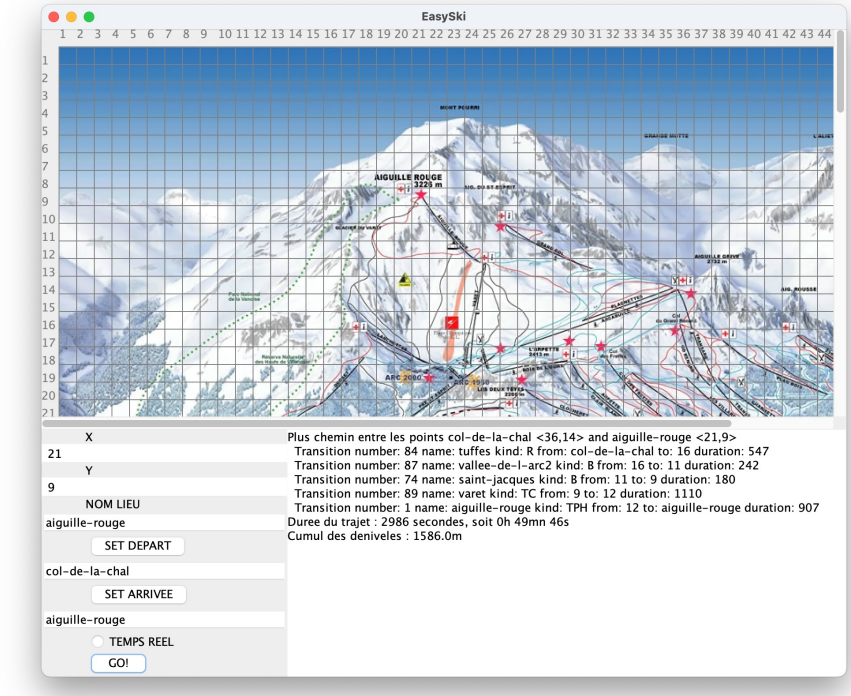

Figure 1 : EasySki

Une station de ski (par exemple "Les Arcs" en Savoie, comme sur la figure précédente) souhaite offrir à ses pratiquants une application "EasySki" leur indiquant l'itinéraire le plus rapide pour se rendre entre des points de la station en empruntant les divers moyens de déplacement : pistes de ski, remontées mécaniques, navettes, ... selon deux modes :

— “absolu” : indépendamment de la fréquentation en cours
— “temps réel” : tenant compte la fréquentation de la station à l'instant de la consultation, notamment des temps d'attente aux remontées mécaniques ou autres modes de déplacement (navettes, ...) obtenus en direct.

1 Analyse/Conception du noyau du logiciel

1.1 Représentation de la station et résolution

Il est facile de se persuader que le plan de la station peut être représenté par un graphe orienté dont les sommets sont les points identifiés et les arcs sont les transitions possibles avec leurs caractéristiques. Il suffira alors d'appliquer l'algorithme de Dijkstra sur ce graphe pour calculer le chemin le plus rapide entre deux points.

Points et transitions
Les points de la station et les transitions sont décrits comme suit.
Pour les points :
— numéro unique (`int`)
— nom éventuel (`String` (sinon = numéro)
— altitude (`double`)
Les transitions entre deux points ont toutes :
— un numéro unique (`int`)
— un nom éventuel (`String`, sinon = numéro)
— un point de départ et un point d'arrivée
et elles peuvent être de diverses natures avec leurs propres caractéristiques :
— descentes (pistes de ski)
  — type (niveau de difficulté) : verte (“V”), bleu (“B”), rouge (“R”), noire (“N”)
  — temps moyen en secondes pour descendre 100 m de dénivelé.
— remontées mécaniques
  — type : téléski (“TK”), télésiège (“TS”), télésiège débrayable (“TSD”), télécabine (“TC”), téléphérique (“TPH”), ...
  — une durée fixe en secondes correspondant à la phase de contrôle et d'installation de l'utilisateur
  — temps moyen en secondes pour monter 100 m de dénivelé.
— navettes
  — type : “bus”, “métro”, ...
  — durée du trajet en secondes.
A titre d'exemples, voir les données du fichier XML `ressources/station/station.xml` qui sera exploité par la suite dans la phase de développement du projet (les coordonnées x, y des points ne seront utiles que pour l'interface graphique en toute fin du projet).

Règles de calcul des durées des transitions
— la durée de descente d'une piste est proportionnelle à son dénivelé.
— pour les remontées mécaniques, la durée est composée :
  — d'un temps d'attente dans la queue obtenu en ”temps réel” ou 0 en mode ”absolu”
  — de sa partie fixe correspondant à la phase de contrôle et d'installation de l'utilisateur
  — et de la durée de remontée à proprement parler, proportionnelle à son dénivelé.
— pour les navettes la durée est composée :
  — d'un temps d'attente dans la queue obtenu en mode “temps réel” ou 0 en mode “absolu”
  — de leur durée de trajet.

A des fins de simulation, dans le mode “temps réel”, les temps d'attente dans les queues (remontées mécaniques, navettes, ...) seront obtenus par une méthode aléatoire. Ne pas se préoccuper de sa formulation précise à ce niveau d'analyse/conception.

1.2 Rapport d'analyse/conception à rendre Lundi 28/11 à 19h

Fournir un rapport d'analyse/conception à votre tuteur (environ 5 pages numérotées, hors pages de garde) qui devra contenir, en respectant la numérotation suivante :

1. Analyse générale et justification des classes nécessaires avec leurs relations munies de rôles, attributs et principales méthodes pour résoudre le problème sur la base d'un schéma UML commenté.
2. Choix et justification des SD (collections Java) appropriées pour représenter les associations n-aires, et leur déclaration dans les classes concernées.
3. Formulation (commentée) en "pseudo-Java" des principales méthodes : essentiellement la méthode de résolution Dijkstra et les principales méthodes requises dans les classes concernées.

Remarque : il s'agit ici de présenter l'analyse/conception de l'essentiel du noyau du logiciel. Il ne doit donc pas être question à ce niveau de quelconque `"main"`, ni d'entrées/sorties utilisateur.

2 Programmation et tests du noyau du logiciel

- Créer sous Eclipse un projet `ski-login` en remplaçant `login` par le nom de login linux de l'un des membres du binôme.
- Y programmer les classes résultant de votre analyse/conception dans un package `solveur`.
- Dans un package `applications`, vérifier le bon fonctionnement en programmant une classe principale `applications.TestNoyau` (`main`) :
  - construire en dur une "mini-station" suffisante à des fins de tests
  - vous inspirer pour cela des données du fichier : `ressources/station/station.xml`
  - effectuer les essais nécessaires du solveur Dijkstra par appel en dur à la méthode de résolution
  - il est bien évident qu'en premier lieu, le mode "absolu" doit être utilisé à des fins de stabilité des tests
  - il ne s'agit pas ici de faire un beau menu interactif, mais seulement de vérifier en dur que votre solveur fonctionne bien, notamment sur les exemples tests ci-dessous.

Exemples tests
Les coordonnées `<x,y>` des points de départ et d'arrivée sont renseignées pour exploiter ces mêmes tests dans l'interface graphique à programmer dans la partie IHM en final, ne pas vous en préoccuper ici.

```
Fastest path between col-de-la-chal <36,14> and aiguille-rouge <21,9> :
  Transition number: 84 name: tuffes kind: R from: col-de-la-chal to: 16
  Transition number: 87 name: vallee-de-l-arc2 kind: B from: 16 to: 11
  Transition number: 74 name: saint-jacques kind: B from: 11 to: 9
  Transition number: 89 name: varet kind: TC from: 9 to: 12
  Transition number: 1 name: aiguille-rouge kind: TPH from: 12 to: aiguille-rouge
Total time : 2986 seconds (0h 49mn 46s)
Cumulative altitude difference: 1586.0m
Fastest path between arc1800 <42,24> and villaroger <5,28> :
  Transition number: 16 name: carreley kind: TS from: arc1800 to: 31
  Transition number: 24 name: clocheret kind: TS from: 31 to: 17
  Transition number: 57 name: muguet kind: N from: 17 to: pre-saint-esprit
  Transition number: 93 name: villaroger1 kind: B from: pre-saint-esprit to: 2
  Transition number: 5 name: aiguille-rouge4 kind: N from: 2 to: villaroger
Total time : 3591 seconds (0h 59mn 51s)
Cumulative altitude difference: 1835.0m
```

```
Fastest path between col-de-la-chal <36,14> and villaroger <5,28> :
  Transition number: 84 name: tuffes kind: R from: col-de-la-chal to: 16
  Transition number: 87 name: vallee-de-l-arc2 kind: B from: 16 to: 11
  Transition number: 88 name: vallee-de-l-arc kind: R from: 11 to: 10
  Transition number: 17 name: cascade kind: B from: 10 to: pre-saint-esprit
  Transition number: 93 name: villaroger1 kind: B from: pre-saint-esprit to: 2
  Transition number: 5 name: aiguille-rouge4 kind: N from: 2 to: villaroger
Total time : 2799 seconds (0h 46mn 39s)
Cumulative altitude difference: 1400.0m
```

Indications :

- Dans le code de `applications.TestNoyau` prendre soin d'isoler chaque essai par un commentaire justificatif.
- Comme dit en fin de la partie 1.1, dans le mode "temps réel", les temps d'attente dans les queues (remontées mécaniques, navettes, ...) seront simulés par une méthode aléatoire. Pour programmer cela utiliser la méthode `java.lang.Math.random()` sur un intervalle de temps `[0 .. MAXWAITING]`, considérant que `MAXWAITING` (constante à définir, par exemple 30 minutes) est le temps d'attente maximum dans les queues garanti par la station.

3 Station en XML

Le format textuel standard de fichiers XML (`https://www.w3.org/XML`) va nous permettre de décrire une station de ski (points, transitions), par exemple `ressources/station/station.xml` dont voici un extrait :

```
<docbase>
<point>
   <numero>1</numero>
   <nom>villaroger</nom>
   <altitude>1200</altitude>
   <x>5</x>
   <y>28</y>
</point>
<point>
   <numero>2</numero>
   <nom>2</nom>
   <altitude>1425</altitude>
</point>
...
</docbase>
```

Le fichier est délimité par `<docbase> ... </docbase>`. Les éléments délimités par `<point> ... </point>` décrivent les points de la station avec leurs informations (numéro, nom, altitude) délimités par leur couple de balise respectif. Il en est de même pour les autres éléments descriptifs de la station.

Nous allons utiliser un parseur Java pour XML (apache xerces) répondant à l'API SAX (Simple API for XML, `http://www.saxproject.org`). Ce parseur lit séquentiellement un fichier XML en entrée et à chaque construction reconnue (début de document, fin de document, début d'élément, fin d'élément, contenu d'un élément, ...) renvoie sur une méthode spécifiée par l'interface `ContentHandler`. Il suffit alors de programmer une classe implémentant cette interface pour effectuer les traitements voulus. Dans notre cas nous aurons besoin d'implanter les méthodes `startElement, endElement, characters` et `endDocument` (les autres ne fourniront pas de traitement spécifique et seront implantées avec un corps vide). `startElement et endElement` sont ap-

pelées quand une balise ouvrante (respectivement fermante) est reconnue. La méthode `characters` permet d'en récupérer le contenu texte s'il existe.

A titre d'exemple le fichier `ressources/parseur/PointHandler.java` contient le source d'un `ContentHandler` (dans un package `parseur`) qui ne fait qu'afficher les points de la station avec leurs informations.

Le code suivant montre comment indiquer au parseur le `ContentHandler` à utiliser et le fichier XML à traiter :

```
// Le parseur SAX
XMLReader reader =
  XMLReaderFactory.createXMLReader(‘‘org.apache.xerces.parsers.SAXParser’’);
// Creation d’un flux XML sur le fichier
InputSource input = new InputSource(new FileInputStream(‘‘station/station.xml’’));
// Indiquer le ContentHandler au parseur
reader.setContentHandler(new PointHandler());
// Lancement du parseur
reader.parse(input);
```

Cette portion de code est utilisée dans l'exemple de programme `ressources/parseur/PointParser.java` permettant de tester ce `ContentHandler` sur un fichier passé en paramètre du `main` (par exemple `station/station.xml`).

Pour essayer :

— intégrer ce code à votre projet Eclipse (dans un package `parseur`)
— ajouter la bibliothèque `/usr/local/xerces/xerces.jar` au `CLASSPATH` du projet :
  — faire bouton droit sur le projet / Properties
  — sélectionner "Java Build Path" puis l'onglet "Libraries"
  — faire "Add External Jars..." et sélectionner `/usr/local/xerces/xerces.jar`
— exécuter :
  — bouton droit sur la classe `parseur.PointParser` et "Run as / Run Configurations ..."
  — onglet "Arguments" : saisir le nom du fichier `station/station.xml`
  — Apply et Run.

Travail à faire

Dans le package `applications`, programmer une version complète de l'application (nommée `Console`) en mode texte dans un terminal, qui :

— charge une station à partir de sa description XML (`ressources/station/station.xml`)
— offre un menu à l'utilisateur pour :
  — choisir des points de départ et d'arrivée
  — choisir le mode de calcul (en temps réel ou non)
— affiche le chemin le plus rapide correspondant à ses choix : transitions à emprunter, durée, cumul des dénivelés (à la façon des exemples tests présentés en partie 2).

Pour cela il vous sera nécessaire de programmer dans le package `parseur` un handler SAX, soit `parseur.StationHandler`, en vous inspirant du handler `parseur.PointHandler` fourni en exemple. Bien comprendre que :

— contrairement à `parseur.PointHandler` qui ne fait que traiter les points de la station, `parseur.StationHandler` doit traiter toutes les entités descriptives (points mais aussi transitions)
— les actions de `parseur.PointHandler` ne faisaient que tracer (afficher) les informations reconnues alors que `parseur.StationHandler` doit permettre de construire une station à partir de ces informations.

L'application doit être "solide", c'est à dire sans "plantage" suite à d'éventuelles erreurs de saisie, par exemple :

— erreurs de paramétrage lors du lancement (paramètres insuffisants ou fichiers inacessibles)
— points de départ et d'arrivée inconnus

— erreur sur la sélection de mode de calcul (temps réel ou non)
— ...

4 Interface Graphique (IHM)

Au final on souhaite offrir une interface graphique d'accès à l'application telle que celle montrée en début du sujet (Figure 1).

Elle doit permettre de choisir des points de départ et d'arrivée, de choisir un mode de calcul (en temps réel ou non) et en conséquence de renseigner à l'utilisateur (partie inférieure droite de la Figure 1) le chemin le plus rapide à emprunter avec sa durée et le cumul des dénivelés correspondants. La sélection des points de départ et d'arrivée doit pouvoir se faire :

— soit en saisissant leur nom ou leurs coordonnées dans des champs textes correspondants
— soit en cliquant sur les points connus du plan représentés par des étoiles rouges.

Pour cela un prototype `ressources/ihm/EasySkiProto.java` est fourni (dans un package `ihm`), comme montré dans la Figure 2 ci-dessous avec sa structure que vous retouverez commentée dans son code.

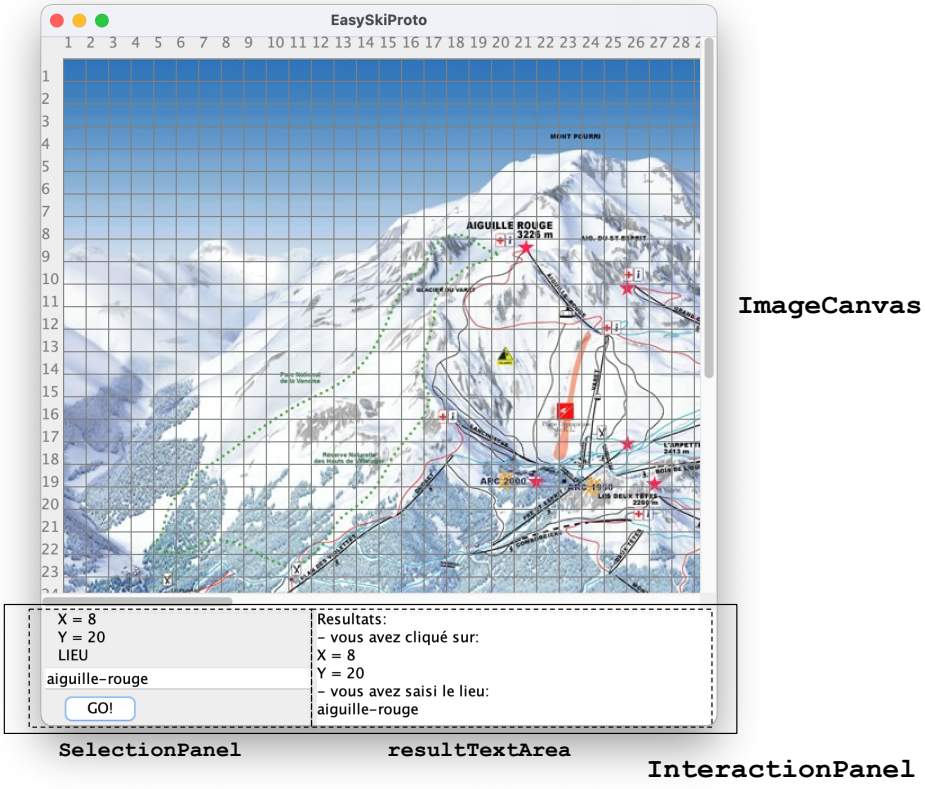

Figure 2 : EasySkiProto

Il permet :

— de cliquer sur une image du plan munie d'une grille de repérage pour récupérer les coordonnées d'un lieu
— ou de saisir directement le nom d'un lieu
— bouton "GO !" : les saisies sont alors reportées dans une zone de texte résultat (`resultTextArea` en bas à droite).

Pour l'essayer :

— l'intégrer dans votre projet Eclipse
— l'exécuter en ajoutant le fichier `ressources/station/station.jpg` en paramètre du main.

Travail à faire

Programmer une interface graphique de votre application `ihm.EasySki` (telle que celle de la Figure 1) en complétant le code de `ihm.EasySkiProto`.

5 Compte-rendu de fin de projet

A rendre à votre tuteur :

Rapport

5 à 7 pages numérotées (hors page de garde et table des matières). Il doit comporter, en respectant la numérotation suivante :

1. Diagramme de classes UML final et expliqué.
2. Schéma de structuration de l'application (répertoires (packages) et fichiers).
3. Indications d'utilisation.
4. Bilan et conclusion, en précisant ce qui a été fait / n'a pas été fait ou ne fonctionne pas, les améliorations possibles et un bilan personnel.

Archive du projet

Votre projet doit fonctionner sur l'environnement standard des machines de TP sous Linux.
Fournir à votre tuteur une archive (`‘‘.zip’’, .‘‘tar’’ ou ‘‘.jar’’`) contenant les sources commentés et structurés en répertoires (packages).