Conservatoire National des Arts et Métiers
École SITI - Département Informatique

Année 2016–2017

UE NFP 136 (VARI 2)

Sujet du projet 2016-2017 : Le Fil d'Ariane

On raconte que le héros grec Thésée, désirant mettre fin aux sacrifices humains qu'exigeait le roi Minos tous les 9 ans, décida d'aller lui-même abattre le terrible monstre mi-homme mi-bête qui était la cause de ces sacrifices : le Minotaure. Pour cela, il pénétra dans le labyrinthe qui abritait ce dernier, et que le roi Minos, bien décidé à s'assurer que cette engeance terrifiante ne s'échappe jamais, avait chargé l'architecte Dédale, père du tristement célèbre Icare, de construire. Mais Ariane, fille de Minos, s'était éprise de Thésée, et lui proposa de l'attendre à l'entrée du labyrinthe en tenant une pelote de fil (ce dernier restera célèbre comme le *Fil d'Ariane*), que son bien-aimé déroulerait au fur et à mesure de son cheminement, pour pouvoir retrouver à coup sûr la sortie du labyrinthe après avoir occis son abominable proie.

Et si le fil s'était cassé ? Voici la situation : Thésée a rempli sa part du marché, et le Minotaure n'est plus. Mais le voilà coincé au beau milieu de ce labyrinthe infernal. C'est donc à vous de jouer : saurez-vous guider Thésée vers sa bien-aimée, en l'aidant à atteindre la sortie du labyrinthe ?

Le but de ce projet est d'implémenter en JAVA un algorithme permettant de retrouver la sortie d'un labyrinthe (où se trouve Ariane) à partir d'un point de départ donné (où se trouve Thésée), si la chose est possible. Dans le cas où la chose est impossible, l'algorithme devra être capable de le déterminer.

Le labyrinthe considéré sera décrit par le biais d'un fichier texte qui vous sera fourni. Ce document a pour objectif de vous guider pas à pas dans les différentes étapes de la réalisation de ce projet, et est à lire attentivement.

Présentation du projet

On considère donc dans ce projet un labyrinthe de forme rectangulaire (ou grille), que l'on modélisera à l'aide d'une matrice (représentant en fait un graphe). Ainsi, rechercher le chemin jusqu'à la sortie dans le labyrinthe reviendra à rechercher un chemin vers le sommet correspondant dans le graphe associé, ce qui peut se faire à l'aide d'algorithmes de parcours vus en cours.

Les sommets seront les cases de la matrice, et il existera une arête entre deux cases (sommets) voisines si elles ne sont pas séparées par un mur du labyrinthe : autrement dit, si on peut se déplacer librement de l'une vers l'autre. Deux cases sont considérées comme voisines si elles sont sur la même

ligne et sur 2 colonnes consécutives, ou bien sur la même colonne et sur 2 lignes consécutives. Les murs du labyrinthe, quant à eux, ne peuvent être situés qu'à la frontière entre 2 cases, et chaque case peut donc être entourée d'au plus 4 murs : un au Nord, un au Sud, un à l'Ouest et un à l'Est.

Dans un premier temps, on récupérera les informations contenues dans le fichier texte décrivant le labyrinthe considéré, et on les stockera dans une structure adaptée : une matrice dont chaque élément contient les informations associées à la case correspondante, à savoir les numéros de ligne et de colonne, les murs entourant la case, etc. Puis, on explorera le graphe représenté par cette matrice à l'aide d'un parcours en profondeur (ce qui nécessite l'utilisation d'une pile), et enfin on affichera à l'écran le labyrinthe et la solution associée (si elle existe).

Étape 1 : la classe `Case`

On se propose dans un premier temps d'écrire une classe `Case`, qui n'est destinée qu'à contenir les informations relatives à une case du labyrinthe. En d'autres termes, chaque objet de cette classe contiendra uniquement un certain nombre de variables d'instance (appelées aussi *attributs*), les accesseurs en lecture/écriture associés qui seront nécessaires, et un/des constructeur(s). Aucune autre méthode ne sera a priori nécessaire. On rappelle par ailleurs que le bon usage en JAVA est de déclarer ces 8 attributs comme `private`.

Voici les 8 attributs qui seront utiles. Les 2 premiers sont des informations liées à la position de la case dans le labyrinthe :

1. Une variable entière représentant **le numéro de la ligne** où se trouve la case, et compris entre 0 et le nombre total de lignes - 1.

2. Une variable entière représentant **le numéro de la colonne** où se trouve la case, et compris entre 0 et le nombre total de colonnes - 1.

Les valeurs de ces 2 attributs seront fixées dès la création de la case, et donc dans le constructeur, dont voici le prototype :

```
public Case(int, int);
```

Les 4 attributs suivants représenteront les murs entourant la case :

- Une variable booléenne, initialisée à *faux*, qui vaudra *vrai* si la case est bordée par un mur au Nord (en haut), et *faux* sinon.

- Trois autres variables booléennes, initialisées à *faux*, et jouant des rôles similaires, pour indiquer si la case est bordée (ou non) par un mur au Sud (en bas), à l'Ouest (à gauche) ou à l'Est (à droite), respectivement.

Enfin, les 2 derniers attributs seront utiles lors du cheminement au sein du labyrinthe, pour garder trace d'informations liées à cette exploration :

1. Une variable de type `Case`, initialisée à `null`, qui représentera la case qui a précédé cette case au cours de l'exploration, dans le cas où cette case a effectivement déjà été explorée (la variable vaudra `null` sinon).

2. Une variable booléenne, initialisée à *faux*, qui vaudra *vrai* si la case est sur le chemin reliant la case de départ à la sortie, et *faux* sinon.

Les valeurs des attributs autres que les deux premiers devront bien sûr être initialisées, mais elles pourront être modifiées (au cours de l'exploration pour les 2 derniers attributs, et lors de la lecture du fichier texte décrivant le labyrinthe considéré pour les 4 autres). Il faut donc prévoir d'ajouter à la classe les accesseurs en lecture et en écriture qui seront nécessaires.

Grâce à cette classe, le labyrinthe pourra être représenté à l'aide d'un tableau à deux dimensions (matrice) d'éléments de type `Case`.

Étape 2 : la classe `PileCases`

L'objectif de cette classe est de définir une pile (structure LIFO) capable de stocker, non pas des entiers, mais des objets de la classe `Case`.

Cette classe est donc identique à la classe `Pile` codée en TP (séance 6), à ceci près que les objets manipulés seront de type `Case`, et non `int`.

Étape 3 : la classe `Labyrinthe`

Il reste à définir une classe représentant un labyrinthe, sous la forme d'une grille rectangulaire : c'est la classe `Labyrinthe`.

Chacun des objets de cette classe possèdera donc un attribut propre : un tableau à 2 dimensions (matrice) composé d'objets de type `Case`. On définira aussi 2 autres variables d'instance, de type `Case` : l'une sera associée à la case de départ, l'autre à la sortie du labyrinthe.

C'est également à l'intérieur de cette classe que seront implémentées les 3 principales méthodes du projet : une méthode construisant une matrice de `Case` représentant le labyrinthe à partir des informations lues dans le fichier texte (**Méthode 1**), une méthode effectuant un parcours en profondeur de la matrice ainsi construite (**Méthode 2**), et une méthode affichant le labyrinthe (et sa solution) à l'écran (**Méthode 3**). Ces méthodes sont décrites en détails dans la suite de cette section. En outre, un constructeur sera nécessaire.

Méthode 1 : construire la matrice à partir du fichier texte

Les deux premières lignes de chaque fichier de données (fichier texte) contiennent, respectivement, le nombre de lignes et le nombre de colonnes du labyrinthe (grille), deux informations qui sont à lire et à utiliser.

La troisième ligne contient la description des murs horizontaux situés au Nord (en haut) du labyrinthe. Plus précisément, chaque mur potentiel est précédé d'un caractère '+', et le caractère qui le suit est soit ' ' (s'il n'y a pas de mur), soit '-' (s'il y a un mur). Voici un exemple d'une telle ligne :

```
+-+-+-+-+-+
```

Ce labyrinthe possède donc 5 colonnes, soit 5 cases par ligne, et chacune des 5 cases de sa première ligne est bordée par un mur horizontal au Nord.

Les lignes suivantes fonctionnent par paires, et suivent toutes le même schéma : la première ligne de chaque paire décrit une ligne de la grille (labyrinthe), et la seconde a le même format que la troisième ligne décrite ci-dessus (si ce n'est que, cette fois-ci, chaque mur horizontal apparaissant sur cette ligne peut border à la fois une case au Nord et une case au Sud).

Plus précisément, la première ligne de chaque paire contient la description des cases de la ligne de la grille associée : toutes les cases sont représentées par le caractère '.', sauf la case de départ (où se trouve Thésée), qui est associée au caractère 'T', et la case de sortie (où se trouve Ariane), qui est associée au caractère 'A'. Si 2 cases consécutives ne sont pas séparées par un mur vertical, alors seul un espace ' ' les séparera dans le fichier texte, et dans le cas contraire elles seront séparées par le caractère '|', signifiant que la case à gauche est bordée par un mur à droite, et que la case à droite est bordée par un mur à gauche. Voici un exemple d'une telle paire de lignes :

```
|. . .|T .|
+-+-+ + +-+
```

Ainsi, sur la ligne du labyrinthe associée à cette paire de lignes du fichier texte, les première, deuxième et cinquième cases sont bordées par un mur au Sud (ce qui n'est pas le cas des troisième et quatrième cases), et les 3 cases situées aux mêmes endroits sur la ligne du dessous seront bordées par un mur au Nord. Par ailleurs, les première, deuxième et troisième cases ne sont séparées par aucun mur, tout comme les quatrième et cinquième cases, alors que les troisième et quatrième cases sont séparées par un mur vertical. Il est à noter que la quatrième case est la case de départ.

La fonction de cette méthode est donc de récupérer, puis de stocker dans la matrice d'éléments (objets) de type `Case`, les informations contenues dans le fichier texte en tenant compte du format décrit ci-dessus. Il est à noter que, si on souhaite ouvrir (à l'aide d'un éditeur de texte) les fichiers de données pour vérifier visuellement la structure des labyrinthes stockés dans ces

fichiers, il peut être nécessaire d'utiliser une police de caractères dans laquelle tous les caractères occupent la même largeur (une telle police est appelée *monospaced font* en anglais), par exemple `Courier` ou `Courier New`.

Pour lire des données dans un fichier, on peut utiliser une variable de la classe `BufferedReader`. Voici comment initialiser une variable `lecteur` de cette classe, pour accéder au contenu du fichier de nom "toto.txt" :

```
BufferedReader lecteur = new BufferedReader(new FileReader("toto.txt"));
```

En appelant la méthode `readLine` (qui renvoie une valeur de type `String`) sur cette variable, on lira le contenu de la ligne suivante du fichier de nom "toto.txt" (la lecture d'un fichier étant séquentielle). Ainsi, dans le cas d'un fichier décrivant un labyrinthe ayant 10 lignes et 25 colonnes, après le premier appel à cette méthode (qui lit la première ligne du fichier), la variable `ligne` contiendra la valeur `"10"`. Après un deuxième appel à cette méthode (qui lit la deuxième ligne du fichier), la variable `ligne` contiendra la valeur `"25"` :

```
String ligne = lecteur.readLine(); //ligne contient "10"
ligne = lecteur.readLine(); //ligne contient "25"
```

La construction d'une variable de type `BufferedReader` est susceptible de générer une exception (si le fichier de nom "toto.txt" n'existe pas, par exemple), et il serait donc préférable de gérer toutes ces instructions dans un bloc `try{...} catch(Exception e){...}`. Enfin, après utilisation, cette variable `lecteur` doit être fermée, à l'aide de l'instruction `lecteur.close();`.

Pour lire le fichier "toto.txt", il est également possible d'utiliser la classe `Scanner`, de la façon suivante :

```
Scanner lecteur = new Scanner(new File("toto.txt"));
```

On peut alors lire chaque ligne à l'aide de la méthode `nextLine()` :

```
String ligne = lecteur.nextLine(); //premier appel : ligne="10"
ligne = lecteur.nextLine(); //deuxième appel : ligne="25"
```

Pour stocker les informations lues sur chaque ligne dans les cases de la matrice correspondantes, on vous rappelle l'existence de :

- la méthode de classe `parseInt()` de la classe `Integer`, qui permet de convertir en entier (variable de type `int`) l'objet de la classe `String` passé en paramètre (par exemple, "2" sera converti en l'entier 2),
- la méthode `charAt()` de la classe `String`, qui renvoie le caractère (variable de type `char`) qui se trouve, dans la variable de type `String` sur laquelle la méthode est appelée, à l'indice (entier) passé en paramètre.

(Pour plus d'informations sur ces méthodes, se référer à l'énoncé du TP 3.)

Méthode 2 : parcours en profondeur du labyrinthe

Cette méthode est destinée à effectuer un parcours en profondeur du graphe associé à la matrice construite à l'aide de la méthode précédente, à partir de la case de départ, et en utilisant une pile (objet de la classe `PileCases`).

On rappelle ici le principe d'un tel parcours :

1. Définir une pile vide ;
2. Empiler la case de départ (qui sera son propre prédécesseur) ;
3. Tant que la pile n'est pas vide faire :
   - Rechercher, parmi les (au plus 4) cases voisines de la case c qui se trouve sur le dessus de la pile, une case c' non encore explorée telle qu'il n'y a pas de mur entre c et c' ;
   - Si une telle case c' existe, alors empiler c' et définir c comme étant le prédécesseur de c' au cours de l'exploration ;
   - Sinon, dépiler la case c ;
   - Si on a atteint la sortie, alors STOP.

Il est à noter que cette méthode utilise les méthodes et attributs définis dans les classes `Case` et `PileCases`. Par ailleurs, si, en sortant de la boucle tant que, on n'a pas encore atteint la sortie, alors c'est qu'il est impossible d'atteindre la sortie à partir de la case de départ considérée.

Enfin, on aura besoin (dans la méthode suivante) de connaître les cases qui se trouvent sur le chemin entre la case de départ et la sortie (s'il existe). Cela peut se faire dans cette méthode, ou dans une méthode séparée. Une fois qu'on a déterminé les prédécesseurs des cases grâce au parcours, il suffit de partir de la sortie, de se déplacer vers son prédécesseur, puis vers le prédécesseur de ce prédécesseur, etc., jusqu'à atteindre la case de départ.

Méthode 3 : affichage du labyrinthe et de sa solution

La dernière méthode de cette classe devra afficher à l'écran le labyrinthe considéré, en faisant ou non apparaître la solution. Elle prendra en paramètre un booléen, qui vaudra *vrai* si la solution doit apparaître, et *faux* sinon.

Dans le cas où l'on ne souhaite pas faire apparaître la solution, l'affichage obtenu aura exactement le même format que le fichier texte lu. Dans le cas où l'on souhaite la faire apparaître, l'affichage obtenu sera presque le même que précédemment, à ceci près que les cases se trouvant sur le chemin reliant la case de départ à la sortie apparaîtront avec le symbole '*' et non '.'.

Touches finales

Pour compléter le projet, il reste à écrire le (court) programme principal. Ce dernier récupérera tout d'abord un paramètre au moment de l'appel du programme : le nom du fichier texte dans lequel est stocké le labyrinthe.

Il faudra ensuite (cf Étape 3) construire la matrice de cases associée (à l'aide de la Méthode 1), et effectuer un parcours en profondeur du graphe représenté par cette matrice (à l'aide de la Méthode 2), en partant de la case de départ (celle où se trouve Thésée). Enfin, on affichera le labyrinthe (à l'aide de la Méthode 3) deux fois : une première fois sans faire apparaître la solution (chemin allant de Thésée à Ariane), puis une seconde fois en faisant apparaître la solution (à l'aide de symboles '*').

S'il est impossible d'atteindre la sortie (position d'Ariane) à partir de la case de départ considérée (position de Thésée), le programme se contentera d'afficher le message suivant : "Sortie impossible à atteindre".

Ainsi, voici ce qu'afficherait le programme pour le labyrinthe suivant :

```
+-+-+-+
|. . .|
+ +-+ +
|A|. T|
+ +-+-+

Voici la solution :

+-+-+-+
|* * *|
+ +-+ +
|A|. T|
+ +-+-+
```

Le projet est à rendre par e-mail, à l'adresse : `cedric.bentz@cnam.fr`. Lors du rendu, il faudra bien évidemment fournir le code source commenté de toutes vos classes (fichiers `.java`), et leur bytecode (fichiers `.class`).