Programmation & Algorithmique 2

TP - Séance 4

11 mars 2022

Matière visée : Héritage

1 Robots

Veuillez télécharger sur Moodle l'archive `labyrinth-class-java11.zip` (ou autre selon la version du jdk que vous utilisez). Cette archive contient quelques fichiers `.class`, notamment :

— `Grid.class`
— `Labyrinth.class`
— `Position.class`

Ces classes permettent de représenter un labyrinthe.

Un labyrinthe peut être vu comme une grille rectangulaire de taille $m \times n$ où chaque case est soit un "mur", soit un "couloir empruntable". Un labyrinthe possède une entrée (position initiale lorsqu'on entre dans le labyrinthe) et une sortie (case du labyrinthe qu'il faut atteindre pour sortir du labyrinthe). À noter que l'entrée et la sortie peuvent être n'importe quelle case qui n'est pas un "mur", et ne se trouvent pas forcément sur le bord de la grille.

Comme dans un véritable labyrinthe, il est impossible de savoir où on se trouve exactement dans le labyrinthe. Par rapport à l'endroit où on se trouve, on peut uniquement savoir si les cases adjacentes (en haut, en bas, à gauche ou à droite) sont un mur ou s'il s'agit d'un couloir empruntable. On peut également savoir si la case sur laquelle on se trouve est la sortie.

Le fichier `Labyrinth.html` donne l'interface publique de l'objet `Labyrinth` (explique comment s'en servir). La classe `LabyrinthTest` donne un exemple des interactions possibles avec le labyrinthe. Vous ne devez pas créer de labyrinthe vous-mêmes. Pour obtenir un labyrinthe, il suffit d'invoquer une des méthodes de classe `laby1()`, `laby2()`, `laby3()` ou `laby4()`. Nous garantissons qu'il existe un chemin entre l'entrée et la sortie des labyrinthes créés grâce à ces méthodes. Dans la suite de l'exercice, nous allons construire des robots qui ont des comportements différents afin de trouver la sortie du labyrinthe. Voici les robots que vous devrez implémenter :

— `RobotRandom`, il se déplace de façon aléatoire ;
— `RobotPreferential`, il a un ordre de préférence pour les directions à suivre ;
— `RobotRightHand`, il fait en sorte de toujours longer un mur à main droite ;
— `RobotRecursive`, utilise le "backtracking" pour tracer un chemin (sans cycle) de l'entrée vers la sortie.

Ces robots ont certes des comportements différents, mais ils ont des attributs en commun, tout d'abord un labyrinthe dans lequel ils évoluent, mais également des méthodes en commun, par exemple `findExit()` qui leur demande de trouver la sortie et de retourner le nombre de pas nécessaires.

Il vous est demandé de créer une classe `Robot` qui représentera un robot générique (qui contient ce qui est en commun parmi les robots), et ensuite de créer une sous-classe pour chacun des robots spécifiques.

Les robots devront créer tout ce dont ils ont besoin afin de pouvoir trouver la sortie du labyrinthe en respectant le comportement énoncé. Par exemple, le robot devra peut-être mémoriser une copie de la grille (construite au fur et à mesure de son exploration).

1.1 RobotRandom

Veuillez créer une classe `RobotRandom`. À la création d'un objet `RobotRandom`, on doit donner en paramètre un objet `Labyrinth`. Cette classe doit posséder une méthode `findExit()` dont le rôle sera d'utiliser les différentes méthodes du labyrinthe afin de trouver la sortie de celui-ci.

Voici le comportement du `RobotRandom` : s'il est sur la sortie, il s'arrête et indique qu'il a terminé. Sinon, il se déplace dans le labyrinthe de manière aléatoire, c'est-à-dire qu'à chaque fois qu'il doit se déplacer, il choisit une case sur laquelle il peut aller. Toutes les cases adjacentes à la position courante (qui ne sont pas des murs) ont la même probabilité d'être sélectionnées.

La méthode `findExit()` doit retourner le nombre de déplacements nécessaires pour que le robot trouve la sortie.

Créez une méthode main (dans cette classe ou dans une autre), qui va créer un labyrinthe, instancier un `RobotRandom`, et demander au robot de trouver la sortie. Elle affiche ensuite le nombre de déplacements que le robot a effectué.

Le robot peut créer tout ce dont il a besoin afin de pouvoir trouver la sortie du labyrinthe, ou à des fins de débuggage (par exemple une copie de la grille supposée pour afficher à l'écran la découverte des murs et la progression du robot dans le labyrinthe... Mais à priori, ce stupide robot n'a besoin de rien de particulier.

1.2 RobotPreferential

Lisez la question entière avant d'implémenter. Il est également conseillé d'écrire l'algorithme sur papier avant d'implémenter.

Veuillez créer une classe `RobotPreferential`. À la création d'un objet `RobotPreferential`, on doit donner en paramètre un `Labyrinth`.

Veuillez redéfinir la méthode `findExit()`. La méthodologie utilisée pour trouver la sortie est la suivante :

- Le robot a un ordre de préférence en ce qui concerne la direction du chemin à emprunter. Par défaut, `UP - RIGHT - DOWN - LEFT` ;
- Lorsque le robot est sur une case, il prend la direction `UP` si la case en haut n'a pas encore été visitée. Sinon, il prend la direction `RIGHT` si la case de droite n'a pas encore été visitée, etc ;
- Lorsque toutes les cases qui l'entourent sont des murs ou ont déjà été visitées, il choisit une des cases déjà visitées (dans l'ordre des directions) et marque la case qu'il est sur le point de quitter par un faux mur (qu'il considèrera par la suite comme un vrai mur du labyrinthe).

Notez qu'il serait sympa de laisser à l'utilisateur le choix de la priorité des directions (à prévoir dans un autre constructeur).

1.3 RobotRightHand

Lisez la question entière avant d'implémenter. Il est également conseillé d'écrire l'algorithme sur papier avant d'implémenter.

Veuillez créer une classe `RobotRightHand`. À la création d'un objet `RobotRightHand`, on doit donner en paramètre un labyrinthe.

Veuillez redéfinir la méthode `findExit()`. La méthodologie utilisée pour trouver la sortie est la suivante :

- Le robot mémorise dans quelle direction/orientation il est en train de se déplacer ;
- Le robot se déplace en longeant le mur qui se trouve à sa droite (relativement à sa direction actuelle), c'est-à-dire il fait en sorte de toujours avoir un mur à main droite.

Notez qu'il est possible que le robot ne trouve jamais la sortie ! Le robot peut facilement détecter cette situation s'il passe sur une case où il est déjà passé avec la même orientation.

1.4 RobotRecursive

Lisez la question entière avant d'implémenter. Il est également conseillé d'écrire l'algorithme sur papier avant d'implémenter.

Veuillez créer une classe `RobotRecursive`. Ce robot va utiliser le principe du *backtracking*. À la création d'un objet `RobotRecursive`, on doit donner en paramètre un objet `Labyrinth` ainsi qu'une profondeur maximale autorisée (explications ci-après).

Veuillez redéfinir la méthode `findExit()`. La méthodologie utilisée est d'essayer de construire un chemin (sans cycle) partant de l'entrée, en allant jusqu'à la sortie. Lorsqu'on remarque qu'on est dans un chemin sans issue, on revient en arrière pour prendre une autre branche. Dans son exploration du labyrinthe, le robot va parcourir certaines cases. La *profondeur maximale autorisée* correspond à la longueur maximale du chemin qu'on autorise entre le point d'entrée dans le labyrinthe et la sortie trouvée par le robot.

Veuillez créer une méthode *récursive* `explore(int direction, int depth)`. Cette méthode doit explorer le labyrinthe à la recherche de la sortie, en commençant par aller dans la direction spécifiée par rapport à la position courante. S'il s'agit de la sortie, on retourne la profondeur du chemin courant. Si la profondeur maximale n'est pas encore dépassée, on doit explorer (récursivement) dans chaque direction possible qui ne crée pas un cycle. Si une solution est trouvée, on retourne la profondeur qui permet de trouver la sortie. Si aucune solution n'est trouvée, ou que la profondeur maximale était dépassée, c'est que la direction choisie ne permet pas de trouver la sortie avec la profondeur spécifiée. Comme il est impossible de trouver une solution en commençant par aller dans la direction spécifiée, on replace le robot dans sa position initiale (avant cet appel récursif) et on retourne une valeur qui fasse comprendre qu'il n'y a pas de solution (`-1` par exemple).

Étant donné que `findExit` retourne le nombre de pas, on affichera à l'écran la profondeur atteinte.

Dans les labyrinthes retournées par `Labyrinth.laby1` à `Labyrinth.laby4`, sachez que vous pouvez toujours trouver une solution en moins de 90 pas.

Autres versions proposées :
Actuellement il est uniquement demandé de trouver un chemin ayant une profondeur maximale autorisée.

- Une fois la sortie trouvée, on pourrait demander d'afficher le chemin emprunté. Ceci requiert un marquage particulier dans la copie de la grille (par exemple, indiquer la profondeur actuelle lors du passage dans une case) (peut être fait naïvement ou efficacement).
- Dans la version de base, l'algorithme effectue un marquage/démarquage dans la copie de la grille. S'il n'est pas demandé d'avoir une profondeur maximale autorisée, le marquage peut être fait d'une autre manière de sorte à être plus efficace.