Le labyrinthe du Minotaure

MO David

Sujet du problème

Persée s'est perdu dans le Labyrinthe du Minotaure. Il faut l'aider à en sortir.

On dispose en entrée de la carte du labyrinthe, constitué d'un tableau à deux dimensions (N étant la dimension du labyrinthe, qui est carré) :
boolean[][] labyrinthe;

La première dimension indique les colonnes, la deuxième correspond aux lignes.
On a donc: N = labyrinthe.length;

Pour chaque position du labyrinthe, true indique la présence d'un mur, false indique un passage.
Au début, Persée se situe en bas à gauche du labyrinthe, sur un passage.
Donc labyrinthe[0][N-1] = false;

La sortie du labyrinthe se trouve en haut à droite, donc labyrinthe[N-1][0] = false;

Persée se déplace horizontalement et verticalement, mais pas en diagonale.

On dispose également d'une classe Point définissant chaque point de passage de l'itinéraire :

```
class Point
{
   public x;
   public y;
}
```

L'objectif est donc de construire la méthode :
List<Point> echappeToi(boolean[][] labyrinthe);

Cette méthode retourne la liste des points par lesquels doit passer Persée pour sortir du labyrinthe, en passant par le chemin le plus court.

MO David
05/12/2016

Logique pour la résolution du problème

Algorithme de parcours en largeur ou BFS (Breadth First Search).

Cet algorithme permet le parcours du labyrinthe de la manière suivante :

Nous allons commencer par le point de départ puis l'algorithme va vérifier les 4 cotés adjacents (haut, droite, bas, gauche) ensuite si un point est un chemin disponible et non un mur, il va ajouter la position du point et sauvegarder son point origine ou " mère " dans la file. Ce point origine permettra de retrouver le chemin le plus court après avoir trouvé la sortie.

Après avoir vérifié les 4 cotés adjacents, on va marquer la position actuelle pour savoir qu'on est déjà passé par là puis on avance au point suivant dans la file, tester les 4 cotés, marquer la position actuelle puis avancer dans la file après avoir fini et cela en boucle jusqu'à avoir trouvé une sortie ou que la file soit vide ou dans ce cas-là on a trouvé la sortie ou il n'y a plus de chemin disponible.

Les étapes rapidement :

1. Mettre le nœud dans la file.
2. Retirer le nœud pour l'analyser.
3. Mettre les côtés proches qui sont des chemins disponibles, sauvegarder leur origine dans la file et marquer la position actuelle.
4. Tant que la file n'est pas vide on reprend à partir de l'étape 2.

Avantages :

- Cet algorithme ne sera jamais bloqué à explorer infiniment des chemins inutiles.
- S'il y a une solution, il sera trouvé.
- S'il y a plus d'une solution possible, l'algorithme trouvera le chemin le plus court.

Désavantages :

- Le principal problème est qu'il peut être très consommateur de mémoire, comme chaque niveau de l'arbre doit être sauvegardé pour générer le niveau suivant, la quantité de mémoire est proportionnelle au nombre de nœud dans la queue. L'utilisation de mémoire est étroitement liée à l'espace, et dans le cas d'un espace trop large il se peut épuiser la mémoire d'un ordinateur classique en quelques minutes.
- Si la solution est loin du départ, cela prendra beaucoup de temps.

MO David
05/12/2016

Explication du code

A noter que j'ai remplacé x et y du sujet par col et row pour plus être explicite

La classe point.

```
6     public static class Point { // la classe point
7         int col;
8         int row;
9         Point origine;
10
11        public Point(int col, int row, Point origine) { // constructeur
12            this.col = col;
13            this.row = row;
14            this.origine = origine;
15        }
16
17        public Point getOrigine() { // récupérer le point d'origine
18            return this.origine;
19        }
20        public String toString() {  //  afficher l'instance de la classe
21            return "Col Row = " + col + " " + row;
22        }
23    }
```

Création de la file et début de la méthode avec les déclarations de variable

```
25    public static Queue<Point> queue = new LinkedList<Point>(); // création de la file
26
27    static List<Point> echappeToi(boolean[][] labyrinthe) {
28        List<Point> solucePoints = new ArrayList<Point>(); // création de la liste solution
29        int N = labyrinthe.length;
30        int[][] maze = new int[N][N];
31        int i = 0;
32        int j = 0;
```

Transformation du tableau booléen en tableau d'entier

```
33        //recuperation du tableau booleen en int
34        while (j < N)
35        {
36            i = 0;
37            while (i < N)
38            {
39                if (labyrinthe[j][i])
40                    maze[j][i] = 1;
41                else
42                    maze[j][i] = 0;
43                i++;
44            }
45            j++;
46        }
```

MO David
05/12/2016

Explication de la boucle

Création de la première requête qui a pour départ en bas à gauche. La boucle continue tant qu'il y a des requêtes dans la file.

```
47          //création de la premiere requete
48          queue.add(new Point(0, N - 1, null));
49          //tant qu'on a des requetes
50          while (!queue.isEmpty())
```

La boucle continue tant que la file n'est pas vide. Le premier élément devient "points" et est effacé, ensuite on va vérifier si on a trouvé la solution avec ce "points" et on créer la liste de points solutions et on renvoie la liste. Autrement on vérifie avec l'ordre suivant : haut, droite, bas, gauche. Si une case voisine est un chemin disponible, on marque la case actuelle et on créer puis ajoute le point dans la queue.

```
50          while (!queue.isEmpty())
51          {
52              Point points = queue.remove(); // on va vérifier l'element suivant de la queue
53
54              if (points.col == N - 1 && points.row == 0) { // Solution trouvé
55                  System.out.println("Le chemin a été trouvé.");
56                  while (points.getOrigine() != null) // on construit a partir de la dest
57                  {
58                      solucePoints.add(points); // on cree la liste de points solution
59                      points = points.getOrigine(); // points précédent
60                  }
61                  return solucePoints;
62              }
63              if (chemin(points.col, points.row - 1, maze, N)) { // On vérifie en haut
64                  maze[points.row][points.col] = -1; // Visité
65                  Point nextP = new Point(points.col, points.row - 1, points); // création du point en haut
66                  queue.add(nextP); // ajout du point dans la file
67              }
68
69              if (chemin (points.col + 1, points.row, maze, N)) { // On vérifie a droite
70                  maze[points.row][points.col] = -1; // Visité
71                  Point nextP = new Point(points.col + 1, points.row, points); // création du point a droite
72                  queue.add(nextP); // ajout du point dans la file
73              }
74
75              if (chemin (points.col, points.row + 1, maze, N)) { // On vérifie en bas
76                  maze[points.row][points.col] = -1; // Visité
77                  Point nextP = new Point(points.col, points.row + 1, points); // création du point en bas
78                  queue.add(nextP); // ajout du point dans la file
79              }
80
81              if (chemin (points.col - 1, points.row, maze, N)) { // On vérifie a gauche
82                  maze[points.row][points.col] = -1;// Visité
83                  Point nextP = new Point(points.col - 1, points.row, points); // création du points a gauche
84                  queue.add(nextP); // ajout du point dans la file
85              }
86          }
87          return null; // aucun chemin disponible
88      }
```

La méthode qui renvoie vrai ou faux pour savoir si un chemin est disponible soit 0 ou 1 et vérifie aussi qu'on est bien dans le tableau.

```
89      public static boolean chemin(int col, int row, int maze[][], int N) {
90          if ((col >= 0 && col < N) && (row >= 0 && row < N) && (maze[row][col] == 0)) { // on vérifie qu'on est toujours dans le tableau
91              return true;                                                               // et la disponibilité de chemin
92          }
93          return false;
94      }
```

MO David
05/12/2016

Le main avec un tableau écrit en dur. On appelle la méthode echappeToi qui est sauvegardé dans points. On vérifie si on a reçu une liste ou null, si on a une liste on imprime les points pour le chemin le plus court, autrement on affiche qu'il n'y a pas de solution.

```
95      public static void main(String[] args) {
96          boolean[][] labyrinthe = new boolean[][] {
97              {false,true,false,false,false},
98              {false,true,false,true,true},
99              {false,true,false,false,true},
100             {false,false,true,false,true},
101             {false,false,false,false,true},
102         };
103
104         List<Point> points = echappeToi(labyrinthe);
105         if (points != null)
106         {
107             for (int i = 0; i < points.size(); i++)
108             {
109                 System.out.println(points.get(i)); // utilise la méthode toString pour afficher
110             }
111             System.out.println("Ceci est le chemin le plus court disponible");
112         }
113         else
114             System.out.println("Il n'y a pas de solution");// pas de solution
115     }
116 }
```

MO David
05/12/2016

Le labyrinthe du Minotaure

MO David