Rapport du Projet de MOGPL: "La balade du robot"

Introduction

Le thème de ce projet porte sur la minimisation du temps pour le transport d'objets par un robot dans le dépôt d'un grand magasin.
Le monde représenté comporte plusieurs contraintes, en particulier le mode de déplacement du robot, que je vais présenter plus en détail.
Le problème est présenté sous forme d'une grille représentant les obstacles mesurant un mètre de longueur.
Chaque case est entouré de rails qui permettent au robot de se déplacer, certaines d'entre elles contenant des obstacles.

Il s'agit donc de trouver le chemin le plus court entre un point A et un point B.

Nous allons donc d'abord donner une formulation du problème en le transformant en un problème dans un graphe orienté. Puis nous coderons un algorithme pour résoudre le problème, pour lequel nous allons faire une analyse de complexité.

Nous effectuerons des essais numériques afin d'évaluer le temps de calcul de l'algorithme, en fonction de la taille de la grille d'une part, et du nombre d'obstacles présents d'autre part.

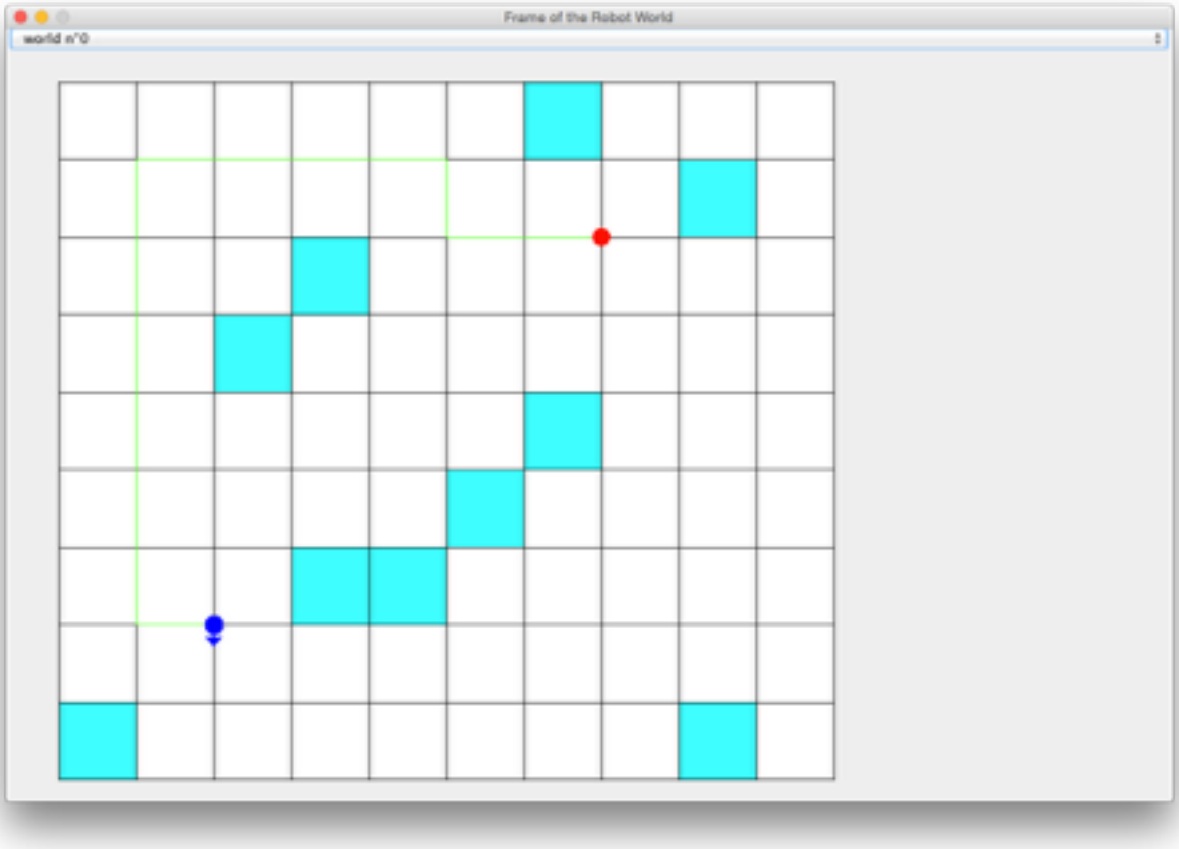
Le projet a été implémenté en Java, j'ai donc utilisé les fonctionnalités objets pour représenter les données du problème.

Question 1:

La transformation du problème sous forme de graphe orienté n'est pas évidente dans le sens où le robot se déplace uniquement sur les rails, et non à l'intérieur des cases.

Nous allons donc formaliser le problème sous forme de graphe, mais il faut donc adapter la formulation du problème en conséquence en implémentant deux représentations différentes pour chaque instance du problème:

- une première matrice pour stocker les coordonnées des obstacles à partir d'un monde fourni en fichier externe;
- une deuxième matrice pour instancier ce même monde, mais du point de vue du robot, c'est-à-dire qu'elle va stocker les coordonnées des intersections des rails accessibles au robot, tout en tenant compte des obstacles précédemment mis en place. Il y a donc une colonne de plus qu'il n'y a de cases dans une ligne, et réciproquement, une ligne de plus qu'il n'y a de cases dans une colonne, ce qui modélise le fait que le robot puisse se déplacer aux extrémités du monde.



Question 2:

Le robot a un mode de déplacement très particulier: pour une action de même coût unitaire d'une seconde, il ne peut se déplacer que dans une seule direction, la direction courante, sur une, deux ou trois cases ou bien tourner à droite ou à gauche.

Etant donné que chaque action a un coût unitaire, nous pouvons donc déterminer le plus court chemin à l'aide de l'algorithme du Parcours en Largeur (BFS). Le temps que met le robot pour aller d'un point A à un point B est donc déterminé par le nombre d'actions effectués par celui-ci.

Pour pouvoir appliquer cet algorithme, il m'a fallu manipuler des données spécifiques pour chaque Point de la grille des rails. J'ai donc décidé de créer une classe Point dans laquelle je stocke pour chaque élément:

- sa clé (0 = case vide, 1 = obstacle, 5 = case de départ, 9 = case d'arrivée),
- ses coordonnées,
- son marquage (s'il a déjà été visité ou pas),
- son orientation (nord, sud, est, ouest) qui représente l'orientation avec laquelle on arrive sur ce Point,
- son parent, qui représente l'élément parent par lequel on arrive sur ce Point,
- son coût;

Mon algorithme BFS est basé sur une file First-In First-Out, initialisé avec le point de départ.
On recherche pour chaque élément ses voisins que l'on stocke dans une liste. Parmi ces voisins, si l'élément n'a pas encore été visité, on l'ajoute dans la file.
On arrête l'algorithme dès que l'on trouve dans la liste des voisins le point d'arrivée, différenciée par sa clé (9).

Pour obtenir la liste des voisins d'un Point, nous distinguons cinq cas:

- les deux cas pour tourner à gauche ou à droite tout en restant sur la même case,
- les trois autres cas pour se déplacer d'une, deux, ou trois cases par rapport à la case courante, selon l'orientation courante.

Pour les cas où le robot se déplace, il est nécessaire de vérifier que le robot ne sorte pas du monde, mais également l'accessibilité de la case où le robot veut se déplacer: si c'est un obstacle, alors il ne doit pas aller au delà, c'est-à-dire qu'il ne doit pas vérifier l'accessibilité des points situés après, en passant « au-dessus » de l'obstacle.

Question 3

Pour une exécution de l'algorithme en fonction de la taille de la grille, pour des grilles de taille N = M = 10, 20, 30, 40, 50, avec un nombre d'obstacles identiques à la dimension et 10 instances par dimension, nous obtenons le tableau des valeurs moyennes suivant:

	M || N || Execution Time
100	10 || 10 || 0 s 3 ms
400	20 || 20 || 0 s 14 ms
900	30 || 30 || 0 s 14 ms
1600	40 || 40 || 0 s 55 ms
2500	50 || 50 || 0 s 159 ms
3600	60 || 60 || 0 s 278 ms
4900	70 || 70 || 0 s 372 ms
6400	80 || 80 || 0 s 808 ms
8100	90 || 90 || 2 s 355 ms
10000	100 || 100 || 5 s 615 ms

Nous avons une complexité quadratique par rapport au nombre de cases, car nous parcourons au pire cas n fois la liste des voisins, qui est au pire de taille n.

Faute de temps, je n'ai pas pu optimiser mon algorithme, mais l'algorithme peut être amélioré en choisissant une table de hachage pour stocker les points déjà visité au lieu d'une liste qui oblige à parcourir un à un les éléments jusqu'à trouver le voisin correspondant.
Cela m'aurait permis d'abaisser la complexité en O(n).

On aurait même pu faire mieux en stockant tous les voisins pour chaque point avant de lancer l'algorithme BFS, de manière à ce que l'on connaisse déjà les voisins de tous les éléments lors du parcours de chaque noeud visité, alors qu'actuellement, la recherche des voisins se fait point par point, en prenant le premier élément de la liste des points déjà visités, et en y appliquant la fonction de recherche du voisinage.

Question 4

Pour une exécution de l'algorithme en fonction du nombre d'obstacles présents, pour des grilles de taille 20x20, avec un nombre d'obstacles de 10, 20, 30, 40, 50 obstacles et 10 instances par nombre d'obstacles, nous obtenons le tableau des valeurs moyennes suivant:

nbObstacles	Execution Time
10	0 s 9 ms
20	0 s 7 ms
30	0 s 5 ms
40	0 s 4 ms
50	0 s 1 ms

La complexité ne change pas, mais la baisse dans le temps d'exécution s'explique par le fait que plus il y d'obstacles, moins il y a de chemins possibles d'un point A vers un point B à rechercher, donc moins l'algorithme met de temps pour trouver le plus court chemin.

Question 5

L'interface graphique ne permet dans l'état d'afficher qu'un seul monde, le premier.
Si l'utilisateur veut définir son propre monde, il faut passer par la console, en modifiant la variable booléenne correspondante présente au début du fichier FrameWorld.java .