PI project 2: Coordinated Robot Motion Planning

Nous présenterons le cheminement logique qui a amené à créer notre algorithme.

Contents

I Idées de base

Compréhension du sujet

La première chose qui saute aux yeux est qu'un robot doit partir d'une destination a sur un pavage pour se rendre à une destination b. Nous basant sur les cours précédents ainsi que d'un peu d'imagination, nous avons imaginé quelques possibilités : tout d'abord, un algorithme qui regarderait les cases proches pour voir s'il peut avancer dans la direction du Target. Ceci ne pouvait évidemment pas marcher, au vu des obstacles et des autres robots, qui ne sont que des obstacles mobiles.

Il nous fallait donc un algorithme qui nous calculait un chemin réalisable entre la position du robot et le Target, qui prenne en compte les obstacles, qui nous donne le chemin le plus court à réaliser…

Nous avions déjà vu Dijkstra en P1 : ceci nous semblait être un algorithme simple à lancer, assez bien connu, aux performances solides.

Problème des changements de chemins

Une fois avec Dijkstra, chacun de nos robots aurait donc un trajet à entreprendre avant d'arriver au Target. Le problème, c'est que quand les robots se lancent dans leur chemin, ils peuvent bloquer les chemins que les autres robots voulaient prendre. Il faut donc trouver une solution, en fonction de quel type de résultat on veut avoir.

Vérifier la condition de rotation interdite

Nous avons inséré dans la vérification de la classe solution un hashmaps noté hash qui contient toutes les locations des robots au dernier tour, et permet de vérifier en un temps linéaire si un robot est là où était un robot au dernier tour, auquel cas il faut vérifier qu'ils ont bien pris la même direction.

II Classes et programmes utilises

Dijkstra

L'algorithme de Dijkstra sert à résoudre le problème du plus court chemin. Il calcule des plus courts chemins à partir d'une source vers tous les autres sommets dans un graphe orienté pondéré par des réels positifs.

Pour vérifier qu'une case n'est pas occupée ni par un robot ni par un obstacle, si on faisait une vérification obstacle. getX(i) et robots.getX(i)à chaque fois ça multiplierait par 0(nombre obstacle + nbr robot) le calcul de déplacement. Des HashMap `currentRobotPos` ainsi que `hashObstacles` nous permette d'effectuer ces recherches dans un temps 0(1).

Classe Coord

Cette classe permet une manipulation beaucoup plus facile des coordonnes : avec comme seules donnes x et y, elle permet de calculer la prochaine coordonnée du robot en fonction de la byte de la

direction, ce qui enlève tous les switches qu'on aurait dû mettre lors de ces calculs. Elle propose de plus un. ToString () efficace dans le débogage, un. Equals (Coord o), et un hashCode que nous utilisons dans les maps.

Class Trajectoires

Cette classe contient l'essentiel de notre travail.

Cette classe permet d'abstractiser l'idée de feuille de route. Elle prend en entrée l'instance d'entrée input et la position actuelle des robots current qu'elle va modifier. Elle met en place un simple Dijkstra prenant en compte la position des obstacles et la position des robots (ou non, au choix) pour générer une feuille de route personnalisé pour chaque robot, de sa position à la position de sa Target. La roadmap est initialisée avec des feuilles de routes null. On utilise la méthode initRoadMaps pour la remplir avec les premières feuilles de route. Ensuite, on peut demander à obtenir la prochaine direction planifiée (getNextPlannedDir), à faire bouger un robot selon la feuille de route (followRoadMap). Pour le moment, on empêche des mvt quelconques : move reste private. A chaque mouvement réalisé, la classe met à jour les différents HashMaps et roadMaps.

Les attributs de cette classe sont un Boolean[] finished qui nous permet de tracker quel robot a atteint son objectif, `ArrayList<ArrayList<Byte>> roadMaps` qui va conserver l'information des anciens pas de tous les robots mais va aussi garder en mémoire les prochains pas à prendre pour atteindre l'objectif.

L'utilisation de la classe tourne vraiment autour de la génération de ces feuilles de routes, et toutes les manipulations que l'on fait sur les robots se doivent de respecter une feuille de route déjà préétablie. Ainsi, demander un mouvement aléatoire génère en réalité une feuille de route dont le premier mouvement décrit est un mouvement aléatoire, et dont la suite forme, s'il existe, le chemin optimal du robot à sa destination. Seul la méthode followRoadMap doit être utilisé pour effectuer un déplacement. La vérification de la possibilité du déplacement se fait avec isMovePossible, qui regarde si un obstacle ou un autre robot se trouve sur la prochaine case souhaitée. La règle qui interdit aux robots le déplacement rotatoire est implémentée en dehors de cette classe. En effet, les manipulations "globales" de l'ensemble des robots, correspondant au mouvement coordonnée et "correct", pour des règles plus compliquées, est laissée à la discrétion de l'utilisateur. C'est à ce niveau que nous avons estimé que la séparation de la logique d'implémentation de déplacement serait la plus agréable.

On peut également choisir de perturber seulement la position des robots ayant atteint leur position finale, dans une logique de fin de parcours où seul quelques robots se verraient interdire l'accès à leur target, les autres formant une barricade autour de sa destination.

III Première tentative, Problèmes rencontres

Grace a la classe Coord, Trajectories et les différents hashmaps, on sait où sont les autres robots ainsi que les obstacles, chaque robot a un trajet idéal qu'il aimerait bien réaliser, et on peut aussi à n'importe quel moment recalculer sa trajectoire, en prenant en compte les autres robots ou non.

Les résultats obtenus sont différents en fonction de comment on choisit quand recalculer le trajet, comment on réagit à un obstacle etc…

Gros problèmes initiaux

Comme j'ai pu le mentionner précédemment, certaines de ces différentes techniques commencent à avoir beaucoup de mal avec une forte densité ou même simplement un nombre élevé de robots. La technique de distance optimale n'arrive pas aux plus grands exemples en quelques minutes.

Les robots qui se retrouvent bloques

Il peut arriver, lors de certaines simulations, que des robots se trouvent encapsules, comme cela a pu arriver avec Donald Trump(election_109) : en effet, des lettres comme le D se referment complètement, et malheur aux robots qui n'ont pas pu en sortir à temps.

Ceci nous donne une réflexion supplémentaire sur les comportements que les robots devraient avoir les uns par rapport aux autres : ils devraient être au courant des déplacements les uns des autres. Nous abordons dans notre dernière partie ces éléments ci

Prise en compte du fait qu'il faut débloquer les targets

Comment arriver à trouver une solution alors que le target se retrouve bloque ? Nous avons tenté une option : l'aléatoire. Il s'agit donc de faire en sorte que les robots qui ont accédé à leur target se mettent à bouger aléatoirement jusqu'à ce que les robots qui n'avaient pas pu accéder à leur target trouvent un passage.

IV Solution :

Description du programme final MyBestAlgorythm

En cas de blocage (i.e. le robot ne peut pas suivre sa feuille de route optimale), on peut créer des stratégies mêlant plusieurs des concepts que nous avons implémentés :

- Ne pas bouger
- Régénérer une feuille de route (en prenant en compte ou non la présence des autres robots. Par exemple, au début du parcours, on peut imaginer que les robots vont beaucoup bouger, donc

on génère une feuille de route sans les prendre en compte. A la fin, la disposition du tableau sera plus statique, donc il semble mieux de prendre en compte la présence des autres robots cette fois-ci.

- Faire se déplacer aléatoirement d'une case les robots ayant atteint leur destination. Comme déjà évoqué, cela permettra peut-être de dégager le passage aux robots en chemin. C'est de plus une bonne idée si on cherche à optimiser le makespan, étant donné qu'ils seront revenu à leur target en une étape ensuite (sauf si on leur a volé leur case, mais ils ne se trouvent pas très loin et on imagine qu'ils rejoindront leur target bien assez vite).
- Faire se déplacer aléatoirement tous les robots. On peut imaginer que les destinations sont très proches les unes des autres et qu'on se retrouve vite avec un gros tas de robot qu'il faut démêler : on peut alors soit demander un déplacement aléatoire à tous les robots, ou bien carrément plusieurs afin de bien dégager le graphe. Les robots reprennent leur chemin tranquillement ensuite.

Le run () que l'on utilise est un mélange graduel de ces procédés, et on peut en modifier les paramètres en raisonnant sur la forme du graphe afin d'obtenir les solutions les plus satisfaisantes.

Ne pas bloquer les autres

Il manque quand même grandement une conscience à notre robot : ne pas bloquer a jamais les trajets des autres. Et ça, pas aléatoirement : même si aléatoirement, on a théoriquement une chance de trouver la solution optimale, ce serait mieux de la trouver à coup sûr.

Il faut les rendre conscients de leurs semblables, et voici comment nous aurions pu le faire :

-pour chaque robot, à chaque tour, vérifier s'il y a des cases qui, si elles sont bloquées, rendent le trajet du robot impossible. Créer un Hashmaps qui contient ces cases et les relient au robot en question.

-Interdire à tout autre robot de pouvoir aller dans ces cases

-Quand le robot est passé par l'endroit en question, l'enlever du HashMap.

On peut

V Annexes :

Présentation des résultats expérimentaux

Pour buffalo, j'utilise les paramètres suivants dans mes strategySwitch : strategySwitch[0] = 1 ; strategySwitch[1] = 1; strategySwitch[2] = 5; strategySwitch[3] = 8; strategySwitch[4] = 12; Int stratNumRandom = 5; Ce qui me donne le résultat suivant :

```
Makespan optimization (CG:SHOP 2021 contest)

1
Input file: buffalo_000_25x25_20_63.instance.json
Reading JSON input file: buffalo_000_25x25_20_63.instance.json...ok
Reading start and target positions...done
        name: buffalo_000_25x25_20_63
        number of robots: 63
Input instance loaded from file
------------------
Instance of a coordinate robot motion planning problem:
        number of robots=63
        number of obstacles=98
        bounding box= [0, 24]x[0, 24]

Solution computed
Solution to the input instance: buffalo_000_25x25_20_63
        number of steps (makespan): 109
        total distance (total number of robot moves): 3333
Saving solution to Json file: buffalo_000_25x25_20_63_makespan.json ...done (109 steps)
```

Pour le reste :

```
Solution to the input instance: galaxy_cluster_00000_20x20_20_80
        number of steps (makespan): 88
        total distance (total number of robot moves): 2333
Given solution validity : false
Running motion viewer
         input problem: 80 robots
         input solution: 88 time steps



------------------
Solution to the input instance: buffalo_000_25x25_20_63
        number of steps (makespan): 571
        total distance (total number of robot moves): 22873
Running motion viewer
         input problem: 63 robots
         input solution: 571 time steps
Given solution validity : true
```

```
Solution to the input instance: election_109
        number of steps (makespan): 237
        total distance (total number of robot moves): 8624
Running motion viewer
         input problem: 109 robots
         input solution: 237 time steps
Given solution validity : true
```

```
Solution to the input instance: small_000_10x10_20_10
        number of steps (makespan): 31
        total distance (total number of robot moves): 114
Given solution validity : true
Running motion viewer
         input problem: 10 robots
         input solution: 31 time steps

Solution to the input instance: the_king_94
        number of steps (makespan): 95
        total distance (total number of robot moves): 2945
Given solution validity : false
Running motion viewer
         input problem: 94 robots
         input solution: 95 time steps
```

Présentation des temps de calculs

Nous avons utilisé `System.nanoTime() ;` pour le calculer le temps d'opération.

Mon ordinateur est équipe d'un processeur Intel i7 de 9eme génération, une carte graphique Nvidia GTX 1660Ti et Windows 10 pro, c'est un Asus ROG Zephyrus G17.

Pour le buffalo : `542581000 nanosecondes`

`Pour l'élection 109 : 373396200 nanosecondes`

`Pour le Galaxy : 176848900 nanosecondes Pour le Small : 56507899 nanosecondes`

`Pour The_King_94 : 139190300 nanoseconds`