DIJKSTRA

INITIALISATION

\*

\* For tous les sommets i de 1 a n

\* Mark(i) <- Faux

\* Cost(i) <- +inf

\* Father(i) <- 0 //sommet inexistant

\* end for

\* Cost(s) <- 0

\* Insert(s, tas)

ITERATION

\*

\* While il existe des sommets non marques

\* x <- ExtractMin(tas)

\* Mark(x) <- true

\* For tous les y successeurs de x

\* If not Mark(y) then

\* Cost(y) <- Min(Cost(y), Cost(x)+W(x,y))

\* If Cost(y) a ete mis a jour then

\* Placer(y, tas)

\* Father(y) <- x

\* end if

\* end if

\* end for

\* end while

Problèmes ouverts :

1 - Co-voiturage :

La résolution de ce problème peut se faire par la minimisation du temps de trajet non-seulement jusqu'à la destination, mais aussi jusqu’à l'autre voiture. En pratique, cela se fait en deux étapes.

Pour la première, nous pensons utiliser deux algorithmes de Dijkstra version A\* modifiés : à chaque itération, le prochain point marqué sera celui minimisant la somme des distances à l’origine, à la destination et à la position actuelle de l’autre voiture. Il faut donc trier les labels des nœuds à parcourir par ordre de distance(origine)+distance(destination)+distance(other).

On doit ainsi remplacer la ligne 28 dans AStarAlgorithm.java par :

|  |
| --- |
| marquage[n.getId()] = new LabelStar(n, false, Double.POSITIVE\_INFINITY, estimation+distanceAutre); |

Cela implique que :

* Les deux occurrences de l’algorithme tournent en parallèle. On utilisera donc des threads.
* La position de chaque voiture est constamment enregistrée et utilisée par le thread concernant l’autre voiture.

Une boucle conditionnelle nous permettra d’arrêter les threads lorsque les voitures se rencontrent.

Une fois que les deux voitures se sont rencontrées, les deux threads se terminent. Les deux chemins parcourus ainsi que le point de rencontre sont enregistrés.

La deuxième étape n’a lieu que dans une situation ou le co-voiturage est utile, c’est-à-dire si le point de rencontre est différent de la destination. Un A\* simple commence alors du point de rencontre vers la destination commune aux deux automobilistes.

Remarque :

Si le passage en sortie de trois chemins et un nœud est trop importante, on peut ne passer que le point de rencontre, à condition d’appliquer à nouveau un algorithme de plus court chemin trois fois (de chaque origine au point de rencontre et du point de rencontre à la destination).

2 - Echange d'un paquet :

Ce problème est similaire à la première étape du premier, à deux points près :

* Les destinations sont différentes.

Cela n’est pas dérangeant, car il suffit alors de lancer chaque thread (similairement à la résolution du problème 1) avec des destinations différentes.

* La rencontre et obligatoire.

Afin d’éviter les solutions de type « non-blabla », on rajoute dans la condition de terminaison de l’algorithme le passage par un nœud commun. Pour cela, une simple variable booléenne *rencontre* permet de vérifier que les deux voitures se sont rencontrées.

3 - Destination de vacances :

Là aussi, le problème est similaire à l’étape 1 du premier problème. Nous pensons utiliser deux threads triant les labels des nœuds à parcourir par ordre de distance(origine)+distance(other).

On peut utiliser A\* en modifiant la condition d’arrêt de l’itération : à la fin de chaque itération, le nœud obtenu (ou les deux chemins y conduisant) est enregistré s’il remplit les conditions imposées, mais l’algorithme continue d’itérer. Il ne s’arrête que lorsque les chemins obtenus ne répondent plus aux critères.

Remarques :

1)

Le critère d’arrêt ne nous semble pas convaincant : certains chemins ne sont potentiellement pas pris en compte, rendant la solution incomplète.

2)

Il vaut mieux, ici, n’enregistrer que les points obtenus, quitte à devoir à nouveau calculer les deux chemins y conduisant : enregistrer tous les couples de chemins menant à chaque point de la solution est énorme !