Projet d'informatique 421 : planification de mouvements de robots

Stanislas DOZIAS, Xavier DURAND

1 Introduction du problème

Ce projet d'informatique a pour but d'optimiser le mouvement de robots sur une grille où se trouvent des obstacles, c'est-à-dire des cases interdites pour les robots. Chaque robot a une position de départ et une d'arrivée (appelée **target**), et le but est de trouver un parcours valide pour chaque robot afin qu'il parte de sa position de départ et finisse sur sa position d'arrivée.

Un parcours valide est un parcours où les robots ne vont d'un temps t à un temps $t+1$ que sur une des cases voisines (nord/sud/est/ouest) libres (sans obstacle) ou restent sur place, tout en évitant toute collision. Nous en différencions 2 types : soit un robot arrive sur la case que quitte un autre robot mais pas dans la même direction : nous appellerons cela un **rentre_dedans** ; soit 2 robots veulent aller sur une même case ou échangent leur case : nous appellerons cela un **bump**. Cette différence nous sera importante par la suite

On peut alors imaginer deux types d'optimisation de ce problème : une portant sur le temps (l'ensemble des robots doit arriver le plus vite possible à sa target), ou sur la distance (on veut minimiser la somme des distances parcourues par les robots). Nous proposons tout d'abord un algorithme qui privilégie plutôt le temps, puis nous exposerons un modification possible pour plutôt optimiser la distance.

2 Présentation de la méthode utilisée

Nous avons adopté une approche locale, c'est-à-dire que nous calculons les chemins les plus courts (grâce à un parcours en largeur) sans se soucier des autres robots. Puis quand les robots se déplacent et provoquent des collisions, on les résout en forçant certains mouvements ou en désignant des **targets intermédiaires**, c'est-à-dire des cases à atteindre avant d'atteindre la target finale. Un robot peut en avoir plusieurs et les stocke comme une pile, ne s'intéressant à un instant t qu'à celle qui est sur le dessus de la pile.

Pour résoudre les différentes collisions, nous avons besoin de deux fonctions : **bump** et **rentre_dedans** (une pour chaque type). Ces fonctions ont pour but de résoudre localement une collision entre deux robots (ou du moins la résoudre partiellement). Cela va au pire en recréer une au même temps où à un temps ultérieur, afin d'être sûr de ne jamais devoir revenir dans le passé. De fait, une fois à un temps t, notre algorithme ne se soucis pas de ce qu'il s'est passé aux temps antérieurs, sauf les assignations de targets intermédiaires, qui est la seule "mémoire" de notre algorithme.

L'algorithme est alors très simple :

- On fait un parcours de graphe (algorithme A*) pour trouver un chemin optimal des robots à leur cible, sans se soucier des autres robots (seulement des obstacles)
- On initialise le temps à $t = 0$
- Tant que tous les robots ne sont pas arrivés :
  - Tant qu'il y a des bump ou des rentres-dedans au temps t :
    on les résout avec les fonctions associées
  - $t = t + 1$

3 Pseudo-code

Tout d'abord nous ne détaillerons pas l'algorithme A*, que nous utilisons avec la distance de Manhattan, car c'est un algorithme connu (l'existence de la solution est vérifiée).
Dans cette partie nous nous intéressons aux pseudo-codes de nos fonctions rentre_dedans et bump, qui sont le coeur de notre algorithme. Mais tout d'abord faisons un petit point sur les structures de données que nous utilisons :

- Tout d'abord à chaque robot nous associons une liste de targets, avec initialement une seule target dans chaque liste : la case d'arrivée du robot associée. Ensuite ces listes se comportent comme des piles avec des ajouts/suppressions de target au fur et à mesure de l'algorithme.
- Á chaque robot nous associons aussi un chemin optimale et un chemin parcouru. Le premier chemin est une liste de cases de la position actuelle du robot (non comprise) à sa prochaine target, quant au second il correspond au chemin parcouru précédemment par le robot (position actuelle comprise).

Voici les pseudo-codes des fonctions rentre_dedans et bumb (les lignes en rouge indiquent les changement qu'apportent nos fonctions à nos instances)

```
fonction rentre_dedans :

   """les robots sont nommés robot_arriere et robot_avant (robot_arriere va sur la case où
    est actuellement robot_avant)"""

   robot_arriere recherche un nouveau plus court chemin avec la case où est robot_avant
    comme obstacle

   si cela ne change pas la longueur du chemin :
        ce nouveau chemin remplace le chemin actuel

   sinon, si robot_arriere est plus proche ou à distance égale de sa prochaine target que
    robot_avant :
        on met la case actuelle de robot_arriere au début de son chemin (le faisant ainsi
    patienter un tour)


   sinon, si la prochaine target de robot_avant n est pas sur le chemin de robot_arriere :
        on met la case actuelle de robot_arriere au début de son chemin (le faisant ainsi
    patienter un tour)

   sinon :
      robot_avant cherche une case voisine (la plus proche) au chemin de robot_arriere (on
    considère la case actuelle de robot_arriere comme un obstacle)
      si une telle case existe :
          robot_avant la définie comme target intermédiaire : on ajoute cette case à sa
    pile de target

      si une telle case n existe pas, on parcourt la grille à partir de la place de
    robot_arriere jusqu'à avoir un choix (c est-à-dire deux cases non obstacle sans compter
    celle d où l'on vient) :
            si cela n arrive jamais :
                on met les cases actuelles des deux robots au début de leur chemin (les
    faisant ainsi patienter un tour)
            si un tel choix existe :
                une de ces cases choisie au hasard devient une target intermédiaire de
    robot_avant
```

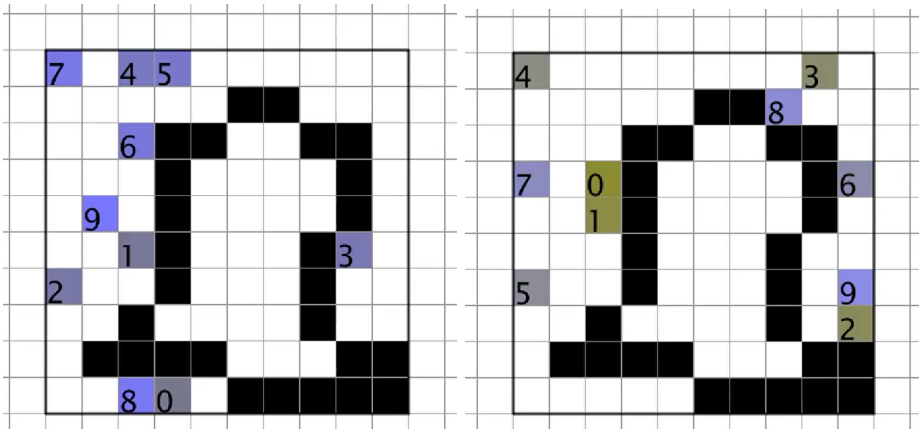
```
fonction bump :

   """les robots sont appelés r1 et r2"""

   si un des robots a un chemin aussi court en évitant le bump :
       ce nouveau chemin remplace son chemin actuel

    sinon :
        chaque robot cherche une case voisine (la plus proche possible) du chemin de l'autre
     à partir de sa case actuelle (un robot peut déjà être en dehors du chemin de l'autre
    robot et la case trouvée est sa position actuelle). Si un choix est possible, la case
    est choisie aléatoirement.

        si aucun des deux ne peut sortir du chemin de l autre :
          on met les cases actuelles des deux robots au début de leur chemin (les faisant
    ainsi patienter un tour)

      si l'un des deux (par exemple r2) ne peut pas sortir du chemin de l'autre :
              r1 définie sa case trouvée comme target intermédiaire (sauf si c'est sa case
    actuelle et dans ce cas on met sa case actuelle au début de son chemin)

      si les deux peuvent sortir du chemin de l autre :
         si il y en a un des deux (par exemple r2) qui est déjà hors du chemin (si les deux
    sont en dehors du chemin de l autre, on choisit pour le rôle de r2 ci-dessous celui qui
    est le plus proche de sa target) :
            si r1 est arrivé :
                r1 définie sa case trouvée comme target intermédiaire (sauf si c'est sa case
     actuelle et dans ce cas on met sa case actuelle au début de son chemin pour le faire
    patienter un tour)

            sinon :
                on met la case actuelle de r2 au début de son chemin (le faisant ainsi
    patienter un tour)

         sinon :
             le robot qui a trouvé une case hors du chemin de l'autre la plus proche la dé
    finie comme target intermédiaire (en cas d'égalité, c'est celui qui est le plus proche
    de sa prochaine target actuelle qui ajoute une target intermédiaire)
```

Quelques précisions s'imposent.

- Quand rentre_dedans et bump définissent une target intermédiaire pour un robot, celui-ci refait un parcours de graphe sans se soucier des autres robots pour atteindre la case, puis il faut de nouveau gérer d'éventuelles collisions, et il y a donc potentiellement besoin de plusieurs appels à ces fonctions pour gérer un problème.
- Nous rajoutons aussi une règle pendant un temps t : si un robot tente d'aller dans une direction, mais qu'une collision le fait changer de direction, alors nous lui interdisons cette direction pendant le temps t. Cela permet d'éviter d'avoir des boucles infinies. Si jamais un robot voit ses quatre directions interdites, il ne bouge plus au temps t et rien ne le fera changer. Nous ne le mettons pas dans le pseudo-code pour plus de clarté. Cependant notre code n'a pas l'air de bien mettre en place cette règle, entraînant parfois des boucles infinies.
- Ce ne sont pas rentre_dedans et bump qui vérifient s'il y a des collisions, c'est une autre fonction qui le fait. D'ailleurs cette détection se fait avec le plus de hasard possible pour pouvoir mieux gérer les cas où aucun robot ne bouge au temps t (notamment à cause de la règle précédente). Si cela arrive, on ne met pas à jour ni le chemin parcouru ni t, nous supprimons seulement la tête des chemins à parcourir, puis on recommence l'analyse de la même configuration (avec potentiellement quelques targets intermédiaires qui se seraient rajoutées), mais avec un ordre de détection des collisions différents, pour pouvoir potentiellement se débloquer. En toute rigueur il faudrait supprimer les potentielles target intermédiaires qui ont été rajoutée par les essais non fructueux, mais nous n'avons pas eu le temps de l'implémenter.

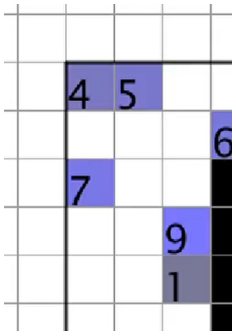
4 Illustration de nos fonctions

Voici des exemples d'application de nos fonctions sur une des grilles fournies (à gauche les positions de départs et à droite les positions d'arrivée).

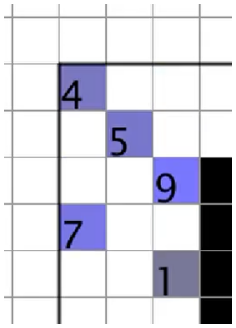


4.1 Exemple de bump

Voici les positions zoomés des robots 4 et 5 à $t = 2$:

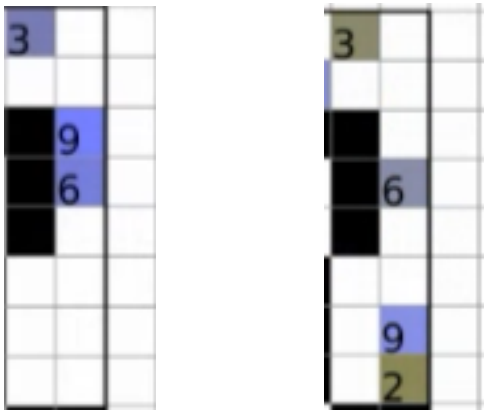


A ce moment là, le 5 veut aller à l'ouest pour ensuite descendre tout droit vers sa target, créant un bump avec 4 qui ne bouge pas. Alors 5 trouve un chemin de même longueur en descendant au sud et en allant à l'ouest plus tard, il le choisi donc et voici les positions au temps 3 :

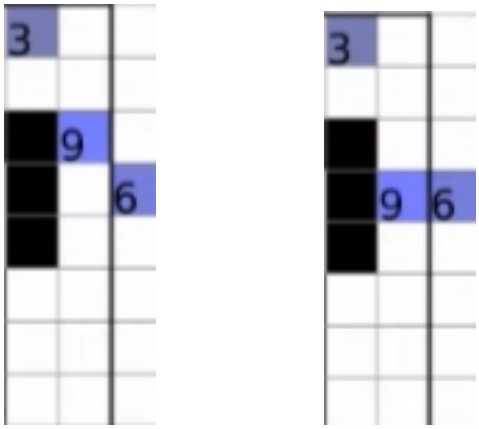


4.2 Exemple d'enchaînement bump/rentre_dedans

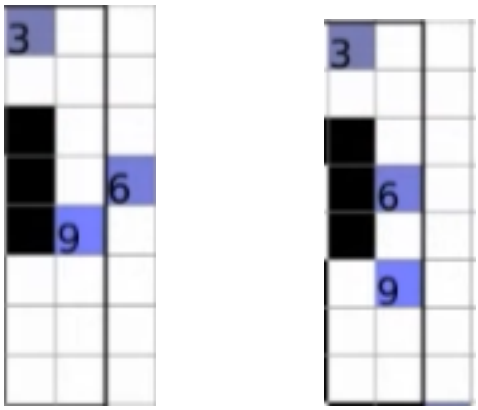
Voici les positions à $t = 14$ (à gauche) et les target (à droite) des robots 9 et 6. Nous allons détailler ce qu'il se passe à partir de ce moment la pour voir comment se comportent les enchainement de bump/rentre_dedans.



À ce moment là, le 9 veut aller vers le bas tandis que 6 veut rester sur place, créant un bump. Comme 9 n'a pas de chemin aussi court en évitant 6, ce dernier cherche la case la plus proche en dehors du chemin de 9, ce qui est la case à sa droite. Cependant, cela crée alors un rentre_dedans, qui est alors réglé par un temps d'attente de 9 et un décalage à l'est de 6. Ensuite, nous avons un bump entre 9 et 6 qui veulent aller sur la même case. Il n'y a pas de chemin aussi court en évitant le bump et les deux robots sont déjà en dehors du chemin de l'autre : c'est celui le plus proche (6) qui attend un tour.



Maintenant nous avons de nouveau un rentre_dedans, mais comme 6 est plus proche de sa prochaine target que 9, il patiente un tour, et puis il n'y a plus de collision.



5 Analyse de l'algorithme

5.1 Avantages/Inconvénients de la méthode

Le problème de cette méthode est de ne traiter que des problèmes entre deux robots, même si la mémoire apportée par les targets intermédiaires permet de gérer des interactions plus complexes. En outre, même si nous sommes convaincus que notre algorithme trouve une stratégie convenable (quand il en existe une) dans le cas de deux robots, il est possible qu'il y ait des cas à plus de trois robots où notre programme peut boucler, surtout dans les cas où beaucoup de robots sont au même endroit. Cependant nous avons essayé de rendre cela le plus rare possible en ajoutant de l'aléatoire à certaines étapes, pour ainsi éviter de boucler. Cependant, des cas pathologiques comme "the_king_94", où la énormément de robots au même endroit fait que l'algorithme boucle, même après de nombreuses tentatives d'ordres de collisions différents. Il arrive aussi que l'on boucle pour d'autres raisons sans pouvoir changer d'ordre de collisions (notament dans le cas de bump dans "galaxy_cluster_00000_20x20_80"

L'avantage de cette méthode est celle commune à tous les algorithme gloutons : l'algorithme générale est relativement simple une fois que l'on a décidé de ce que l'on faisait dans le cas d'un rentre_dedans ou d'un bump, et il n'y a pas besoin d'avoir de vue d'ensemble de la grille entière.

De plus la manière dont les robots sont gérés ressemble beaucoup à ce qu'il se passeraient si les robots étaient conduits par des pilotes tous indépendants les uns des autres, qui prendraient donc constamment le meilleur chemin pour aller à leur target, et en changeant de direction qu'au dernier moment s'il y avait une collision. On a donc ainsi le modèle d'un trafic routier, et comme on s'y attend, s'il y a beaucoup de monde au même endroit, il y a embouteillage.

On pourrait imaginer de différencier nos fonctions suivant que l'on veuille optimiser la distance ou le temps, mais nous n'avons pas réussi à trouver de modifications véritablement concluantes pour l'un ou l'autre.

5.2 Analyse de correction

De part notre boucle qui permet de régler toutes les collisions à un temps t, nous sommes sûr d'avoir des chemins valides quand notre algorithme termine, assurant la correction de l'algorithme. Bien sûr les chemins ne sont pas optimaux ni pour la distance ni pour le temps.

5.3 Analyse temporelle

Nous allons noter n le nombre de robots et c le nombre de cases de la grille de départ.

Le temps d'exécution d'un parcours de graphe (nous utilisons A*, qui est en moyenne plus rapide qu'un simple parcours en largeur) pour trouver le plus cours chemin d'un robot à sa target se fait en $O(c)$ car chaque sommet n'est relié qu'à au plus 4 autres sommets.

Or, de part la potentielle recherche de nouveau chemins à chaque bump/rentre_dedans (ceci étant ce qui est le plus coûteux pendant ces fonctions), nous avons, à l'étape t, une complexité égale à $O(c \times$ (nombre de collisions à l'étape t)), car nous faisons un nombre constant de parcours de graphe à chaque collision. Or une collisions provoque au moins l'abandon d'une case voisine au temps t pour au moins un robot, et un robot ne peut abandonner qu'au plus 4 case à un temps t. Il y a donc $O(n)$ collision à un temps t. Si on note maintenant T le temps que met le dernier robot à arriver à sa target finale, on obtient finalement une complexité de :

$$O(n * c * T)$$

6 Visualisateur à la volée

Comme notre algorithme fonctionne étape par étape, nous avons mis en place une classe pour visualiser l'avancement de notre algorithme en coup par coup sans avoir besoin d'une solution entière pré-compilée.
Pour cela, nous nous sommes appuyer sur la classe MotionView et nous calculons le coup d'après à chaque fois que nous demandons le coup suivant.
Pour ne pas recalculer deux fois le même coup, nous gardons en mémoire chaque coup. Ceci nous permet de ne balader dans les coups précédents et suivants sans en recalculer.
Le lancement de ce visualisateur est le même que pour la classe MotionViewer sauf qu'il n'y a que besoin de spécifier le fichier datasets.