Rapport de projet

Coordinated motion planning

INF421 | Third Computational Geometry Challenge - CG:SHOP 2021

Josselin Somerville et Olivier Lepel

PRÉSENTATION DU PROBLÈME

On cherche à coordonner le mouvement d'un grand nombre de robots dans un plan discrétisé sous forme de grille : chaque robot a une case de départ et une case d'arrivée. On doit établir de manière algorithmique une série d'instruction pour chacun des robots, de sorte que chaque robot arrive à destination sans heurter les autres et en évitant un certain nombre d'obstacles. Les deux grandeurs à minimiser sont, d'une part, la durée totale du mouvement coordonné (makespan), et, d'autre part, le nombre de pas total effectué par l'ensemble des robots. Il s'agit d'un problème posé dans le cadre de la conférence SoCG 21 ; sa description détaillée se trouve ici. Parmi les exemples d'application d'un tel système, on peut citer le pilotage de robots transportant des colis dans une usine, un entrepôt ou une plateforme logistique.

Nous avons implémenté notre algorithme en Python, en utilisant la bibliothèque cgshop2021_pyutils fournie par les organisateurs du concours. Nous avons également développé à l'aide de l'environnement Processing un outil de visualisation des instances et des solutions que nous avons obtenue, les images de ce rapport sont issues de captures d'écran de cet outil. Les cases de départ des robots y figurent en vert, les cases d'arrivée en rouge et les obstacles en noir. Les cases vert foncé sont à la fois des cases de départ et d'arrivée.

CHOIX DU MODÈLE

Nous avons opté pour un algorithme qui, pour chaque robot, simule une marche aléatoire. À chaque pas, le robot doit choisir entre différents mouvements légaux (les cases pour lesquelles il n'y a pas de collision). Les mouvements qui rapprochent le plus (au sens du plus court chemin disponible) le robot de sa cible sont très favorisés par les probabilités qui leurs sont assignés. On fait ainsi plusieurs tirages par instance et on choisit la meilleure solution.

Le tirage de la case choisie pour le déplacement d'un robot ne dépend pas que de la direction de sa cible mais d'un grand nombre de paramètres. Voici une liste non exhaustive des paramètres entrant en compte dans la prise de décision de la direction à suivre :

Paramètres d et a, de déterminisme et de randomisation

Chaque choix pour le robot (les quatre directions cardinale ou l'attente sur place) modifie la distance du robot à sa cible de -1,0 ou 1. Les probabilités de faire chacun de ces choix sont calculé à partir de ce delta de distance. Le coefficient de déterminisme permet d'accorder un bonus à la (ou les) direction qui permet de réduire la distance de 1. Plus on augmente ce coefficient, moins le comportement des robots est aléatoire (ils tentent d'aller droit au but). Au contraire, le coefficient de randomisation (noté *aleatoire dans le code*) permet d'attribuer une probabilité plus importante de choisir les directions les moins favorisées. Ce dernier coefficient permet notamment de résoudre un certain nombre de blocages : si deux robots se gênent l'un l'autre, la composante aléatoire du processus de prise de décision en écartera un des deux du chemin de l'autre.

Paramètre de répulsion

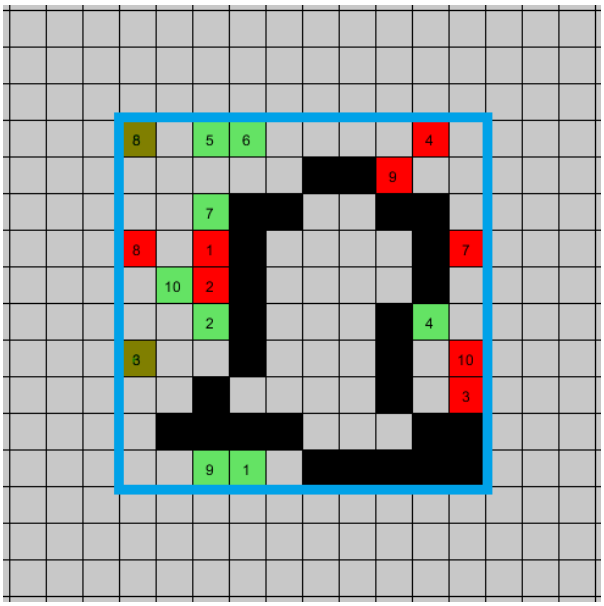
On implémente un effet de répulsion entre les robots en mouvement, selon leur distance les robots se repoussent légèrement. Cela permet de diminuer le nombre de conflit pour l'occupation d'une case, puisque les robots tendent à garder une petite distance entre eux. Cet effet est nettement augmenté dans la phase d'écartement aléatoire des robots. On peut également régler la portée de la répulsion (seuls les robots étant à une distance inférieure à la portée sont considérés)

Paramètre d'effet de groupe [poisson]

Quand un groupe de robot occupe un ensemble de cases contiguës, on peut les encourager (de manière probabiliste) à prendre tous la même direction, comme un banc de poisson.

AJOUT D'UNE MARGE DE TRAVAIL

Les consignes autorisent aux robots de sortir de la grille de départ pendant leur trajet (sans limite de distance) pour ensuite revenir. Nous implémentons cette possibilité en ajoutant un certain nombre de cases vides de *marge* de chaque côté de la grille de base.



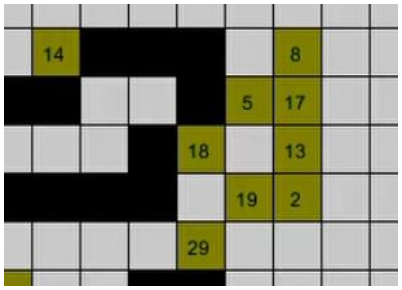
Sur l'exemple ci-dessus (small_000_10x10_20_10), le rajout d'une marge de trois cases de chaque côté de la grille permet de trouver de meilleures solutions, et de manière plus efficace. La grille 10X10 de base est encadrée en bleu. Ainsi, le robot numéro 3 peut rejoindre sa case d'arrivée en contournant le centre de la carte dans le sens trigonométrique (alors qu'il était forcé de le faire dans le sens horaire sinon) et la case d'arrivée du robot 7 n'est plus un goulot d'étranglement.

On a fait le choix pour les robots qui sont arrivés à leur case de destination de ne plus jamais les bouger : il ne se déplacent pas pour laisser passer d'autres. Cela permet de concentrer les ressources pour diriger les autres robots. Cette décision suppose en particulier, pour les instances les plus complexes, que l'on contrôle l'ordre d'arrivée des robots et qu'on le choisisse bien. En effet, certaines cases d'arrivée de robot se trouvent enclavées entre les cases d'arrivées d'autres robots (et éventuellement des obstacles)

La première étape consiste en un parcours en profondeur du terrain à partir de la case d'arrivée de chaque robot. Cela pose la base de l'itinéraire du robot. Ce parcours en profondeur ne permet pas directement d'obtenir l'itinéraire qui sera effectivement pris par le robot : il pourra rencontrer, sur son chemin, d'autres robots en mouvement ou arrivés à leur destination.

PRIORITÉ DE DÉPART

Pour éviter que l'arrivée prématurée de certains robots en empêche définitivement d'autres d'atteindre leur cible, on est amené, sur les instances les plus denses, à former des groupes de priorité pour un départ décalé.



Ici, la case d'arrivée du robot 18 est enclavée : si les robots 5, 13, 19 et 29 arrivent trop tôt à destination, ils risquent de bloquer 18.

Détermination des groupes de départ

Voici un pseudo-code expliquant comment on obtient les groupes de priorité :

```
PasArrivés = tous les robots
DéjaArrivés = []
Groupes = []
Tant qu'il y a des robots dans la liste pasArrivés :
    Groupes.ajouter( [...] )
    Obstacles = [case d'arrivées des robots dans pasArrivés] + [case de départ des
robots dans déjaArrivés] + obstacles de la map
    Pour chaque robot dans pasArrivés:
        On recalcule le parcours en largeur avec les obstacles
        S'il existe un chemin:
            On ajoute le robot à déjaArrivés et on le retire de pasArrivés
            Ajout = Vrai
            On ajoute le robot au dernier groupe
     Si non(Ajout):
         Renvoyer "IMPOSSIBLE"
Renvoyer inverser(Groupes)
```

Clairement, on ne peut pas explorer tous les ordres de départ possibles (il y en a n!). Par conséquent, on procède comme suit : pour chaque robot, on détermine s'il peut aller à sa case d'arrivée si tous les autres robots sont déjà à leur case d'arrivée. Si oui, alors on lui assigne la priorité la plus faible. Ensuite, on recommence en enlevant toutes les cases d'arrivée des robots non prioritaires et en ajoutant toutes leurs cases de départ.

Départ des groupes et anticipation

D'après la façon dont nous avons défini les groupes de priorité, si on déplace les robots dans l'ordre des groupes, chaque robot pourra atteindre sa cible. On peut soit faire partir les groupes les uns après les autres en attendant que les robots du premier groupe soient tous arrivés avant de lancer le deuxième, soit faire partir les groupes sans attendre que le groupe précédent soit complétement arrivé (anticipation du départ). L'anticipation du départ présente un intérêt certain pour l'optimisation du makespan. Cependant, avec l'anticipation, on n'a plus la garantie qu'un robot puisse rejoindre sa case d'arrivée puisque les robots du groupe suivant pourraient le bloquer.

Dans le cas de l'optimisation de la distance, l'anticipation n'a aucun intérêt et risque de faire échouer la simulation. On peut même décider de faire se déplacer les robots un par un quand on optimise la distance. Ainsi, il n'y a plus aucun risque de collision, on peut donc mettre à zéro les paramètres de répulsion et de mouvement aléatoire, ce qui aura pour effet de réduire la distance parcourue

Tri au sein des groupes de priorité et limite du nombre de robots en mouvement

Nous nous sommes rendu compte que sur des instances moyennement denses (galaxy_cluster2_00003_50x50_25_625 par exemple), il y avait au final très peu de groupes de priorité après écartement (ici seulement 2 groupes : le premier contenant 34 robots et le deuxième 591). Ainsi, une fois le premier groupe arrivé, 591 robots se mettent en mouvement pour rentrer dans un carré de côté 50. Inévitablement, il y a des embouteillages.

ROLLBACK

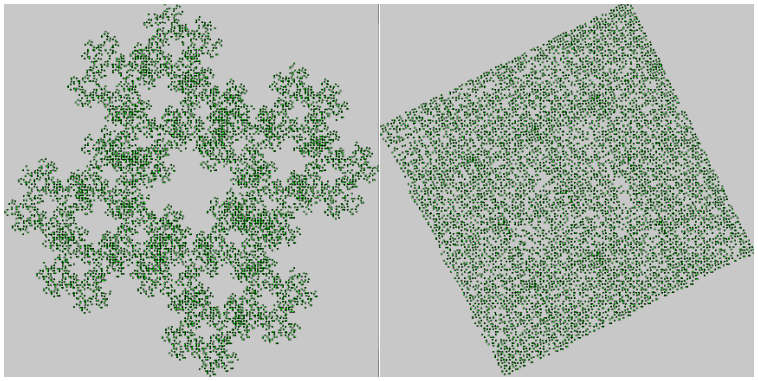
Si, lors de la marche aléatoire, on constate que, pendant un nombre de pas trop long, aucun robot n'arrive à sa destination, on fait un *rollback* c'est-à-dire que l'on revient en arrière et que l'on recommence la simulation au dernier point ou un robot a atteint sa cible. Cela permet d'améliorer sensiblement la qualité des solutions, puisqu'on écarte les tirages pour lesquels on a été le moins chanceux.

ÉCARTEMENT DES ROBOTS

Pour les instances comportant les plus fortes densités de robot, les robots se gênent entre eux. Une solution à ce problème est, dans un premier temps, de les éloigner du centre de la carte puis de procéder comme habituellement.

Première approche : l'écartement déterministe

Dans un premier temps nous avons exploré une méthode déterministe. L'idée est de procéder de manière récursive : à chaque itération, nous découpons la "zone" en 4 rectangles de tailles égales. Chaque rectangle se dirige ensuite dans une direction donnée. On obtient ainsi des figures "fractales" comme on peut le voir ci-dessous :



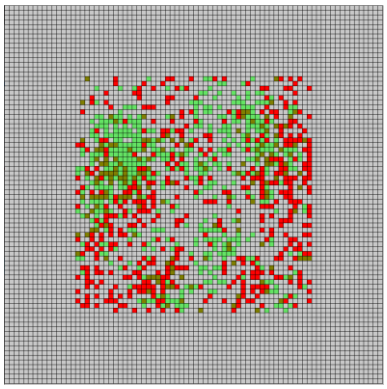
Rapidement, nous avons observé des problèmes. Tout d'abord il était assez difficile de trouver des déplacements qui assuraient qu'il n'y ait aucune collision. Ensuite, comme on peut le voir sur les figures ci-dessus, les robots se dispersent mais ne libèrent pas vraiment le centre et restent condensés en certaines zones. Enfin à cause de la structure récursive, beaucoup de robots se retrouvent à faire des aller-retours inutiles, augmentant grandement la distance parcourue.

Deuxième approche : l'écartement probabiliste

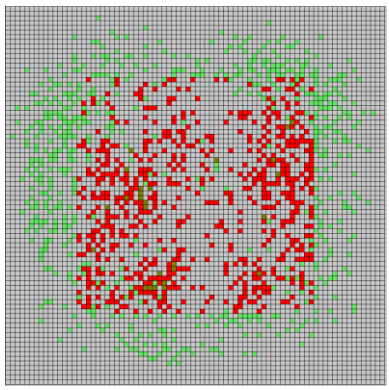
Pour résoudre les problèmes de la méthode déterministe, nous nous sommes tournés vers une méthode probabiliste. L'écartement se réalise en 2 temps : on souhaite tout d'abord que les robots libèrent la zone de départ (carré défini dans le fichier : 50x50 dans les figures ci-dessous). Une fois les robots sortis de la zone centrale, on cherche à les écarter pour créer le plus de place entre eux.

Pour réaliser ces deux étapes on met en œuvre un nouveau concept : le potentiel. Lors de son déplacement aléatoire, un robot va fortement privilégier le déplacement qui maximise son potentiel (le potentiel étant défini par une valeur en chaque case). Ainsi, on peut simplement assigner la distance d'une case au centre de la carte comme potentiel ce qui aura pour effet de pousser les robots à s'éloigner du centre de la carte. En dehors de la zone centrale, on applique un potentiel très élevé, constant, pour que les robots cessent de s'éloigner sans pour autant rentrer à nouveau dans la zone centrale.

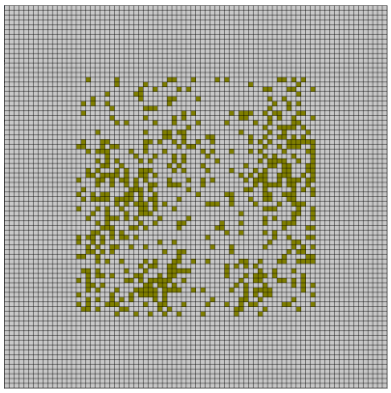
Situation initiale



Position après écartement des robots



Situation finale



L'approche probabiliste rajoute un coût en calcul non négligeable car il faut calculer au moins un parcours en largeur par robot.

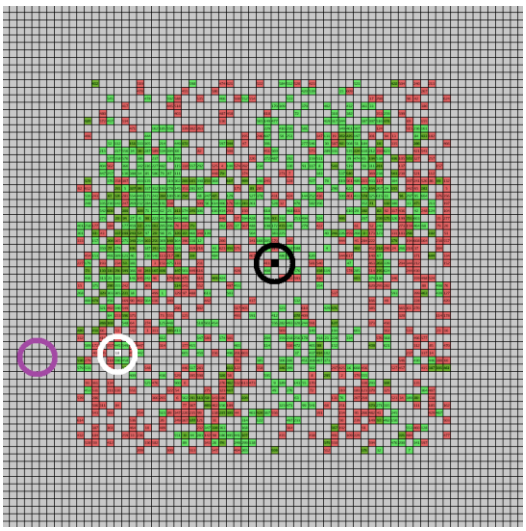
Améliorations du processus d'écartement

Déplacement simultané du premier groupe

Après avoir observé un grand nombre d'écartements, nous avons réalisé que très rapidement la zone centrale devient moins dense et ainsi qu'un nombre limité de robots pourraient circuler sans problème. À partir de ce constat, nous avons décidé que dès le début, les robots du premier groupe de priorité commenceraient à aller à leur cibles, pendant que les autres robots s'écartent. Bien sûr, une fois le premier groupe arrivé, les suivants enchaînent tant que l'on continue l'écartement. Pour que les robots prioritaires puissent circuler plus facilement, leur cases d'arrivée sont considérées comme des obstacles par les autres robots. Par la suite, nous avons implémenté des "cases à éviter" (voir p. x). On a ainsi pu rajouter que les chemins reliant les robots prioritaires à leurs cases d'arrivées étaient à éviter par les autres pour que les robots prioritaires atteignent le plus vite leur case d'arrivée. Contrairement aux obstacles, les cases à éviter peuvent être traversées par les robots ce qui assure que les robots ne seront jamais bloqués et pourront bien se disperser comme voulu. (On peut par exemple imaginer le cas d'un robot prioritaire qui part du coin haut droit de la zone centrale et qui doit aller dans le coin bas droite. Si on utilisait des obstacles, il bloquerait tous les robots voulants "s'écarter à droite", mais avec le système de cases à éviter, ce n'est pas le cas.)

Écartement dans la direction de la cible

Ensuite, nous nous sommes rendu compte que pour certains robots que seulement imposer au robot de s'éloigner du centre, n'était pas forcément le choix le plus judicieux. En effet, comme on peut le voir dans l'exemple ci-contre, le robot (en noir) se trouve relativement au centre. Pourtant, sa case d'arrivée se trouve très à gauche. On court donc le risque que le robot s'éloigne vers la droite et doivent ensuite reparcourir toute la zone centrale. Ceci ajoute bien évidemment de la distance, du makespan mais surtout des embouteillages. Ainsi, en plus du potentiel, nous avons remis un calcul de parcours en largeur pour que le robot se dirige dans la bonne direction. Cependant la cible de ce parcours n'est pas la cible réelle du robot (en blanc) mais sa cible décalée hors de la zone centrale (en violet)



ÉVITEMENT DES CASES D'ARRIVÉE DES AUTRES ROBOTS

Au sein d'un même groupe de priorité, on fait en sorte que les robots évitent de passer par les cases d'arrivée des autres. Appuyons nous sur l'illustration ci-dessus pour expliquer le processus. La première colonne montre le résultat du parcours en largeur classique depuis la case d'arrivée (0) et le chemin favorisé pour le robot en rouge. Dans la deuxième colonne, on a rajouté dans la case encadrée en bleu une case d'arrivée d'un autre robot. Le parcours en largeur est modifié pour dissuader le robot de passer par cette case. Cependant (troisième colonne, deux cases bleues à éviter), si le détour est trop coûteux, le robot garde la possibilité de passer par une case bleue. En effet, interdire la case (comme on le fait dans la quatrième colonne en la notant comme obstacle) serait trop pénalisant.

NETTOYAGE DE SOLUTIONS ET SUPPRESSION D'ALLER-RETOURS

Une fois que l'on a obtenu une solution valide par la marche aléatoire, il est possible d'implémenter des optimisations supplémentaires en post-traitement. On décrit au paragraphe suivant l'optimisation présente dans notre code, qui permet un petit gain de makespan et un gain très significatif en distance totale parcourue.

Appelons boucle le comportement d'un robot qui, au cours de son trajet, passe deux fois par la même case en visitant d'autres cases entre temps. Toutes les boucles ne sont pas inutiles : certaines permettent de libérer temporairement des cases pour céder le passage à d'autres robots qui en ont besoin. On peut cependant détecter relativement facilement certaines boucles inutiles et les éliminer. On observe le changement d'occupant de chaque case au cours du déroulement de la solution. Si la case se vide, on garde en mémoire le robot qui s'y trouvait. Si on détecte une séquence pour laquelle la case reste vide entre deux passages d'un même robot, on sait que l'on peut supprimer tous les déplacements dudit robot entre les deux visites (il reste alors sur place).

PROBLÈMES RENCONTRÉS

Un premier problème a été la rapidité des calculs. En effet, nous avons choisi le langage Python pour sa simplicité ; le coût de ce choix a été un ralentissement important (comparé à un langage comme Java). Pour un fichier comme la galaxie 50x50, nous mettions près de 10 minutes à faire un essai. Les essais n'étant pas toujours fructueux, on peut prendre des heures à trouver une solution correcte.

Un second problème a été la quantité de mémoire utilisée. En effet, chaque robot stocke un objet *Map* qui est basiquement un tableau d'objets *Case*. Ainsi, si on prend l'exemple de la galaxie 100x100 avec 8000 robots, une fois des marges de 50 rajoutées de chaque côté, on se retrouve avec 8000 tableaux de 200 x 200. Chaque case étant composée de ses coordonnées, d'un dictionnaire de ses parents, de ses fils, de son adresse mémoire et autres, on atteint des tailles en dizaines de gigaoctets. Malheureusement, ni nos ordinateurs, ni Google Colab (avec 25Go de RAM) n'ont pu faire tourner notre algorithme sur de telles instances.

PISTES D'AMÉLIORATIONS

Nous n'avons pas eu le temps d'implémenter toutes nos idées ; voici donc une liste non exhaustive d'améliorations qui pourraient être apportées à l'algorithme :

Suppression de plus de cycles inutiles

Dans notre nettoyage des solutions nous n'avons implémenté qu'un nettoyage des cycles ou un robot passait 2 fois sur la même case sans qu'un autre ne passe dessus entre temps. En réalité cette condition n'est pas nécessaire mais seulement simplificatrice. En effet si l'on détecte que le robot 1 se trouve à l'étape 4 en la case (0,0) et y revient à l'étape 10, ce n'est pas parce que le robot 2 passe sur cette même case à l'étape 8 qu'on ne peut supprimer ce cycle. En effet si le robot 2 a lui-même fait un cycle entre 2 étapes telle que l'étape 8 soit comprise entre le début et la fin du cycle, alors on peut supprimer les cycles des robots 1 et 2 (à condition qu'il n'y ait pas de robot qui passe sur la case de départ du cycle du robot 2). On peut alors récursivement ajouter tous les robots impliqués dans les cycles entremêlés. Si on n'a plus aucun robot à ajouter alors on peut supprimer tous les cycles et si un des robots impliqués n'effectue pas un cycle, on ne peut pas les supprimer.

Déplacements groupés

Au final, nous avons peu exploité la stratégie de déplacer les robots en blocs puis de petit à petit diviser de façon récursive ces blocs. Nous avons essayé pour l'écartement déterministe mais nous avons décidé d'abandonner cette idée. On pourrait imaginer qu'au lieu d'écarter les robots de façon aléatoire, on les regroupe par case d'arrivée et que l'on forme des blocs se déplaçant vers la zone d'arrivée, ce bloc se diviserait lui-même ensuite et ainsi de suite. Ceci permettrait notamment d'éviter des allers-retours inutiles et donc d'améliorer la distance.

Meilleure gestion des priorités

Une des problématiques qui nous ont beaucoup occupé était de déterminer les ordres de priorité. Nous avons opté pour une méthode nous garantissant que le robot puisse atteindre sa case d'arrivée. Le problème étant que nous découpons la résolution d'une instance en segments ou les robots sont soit sur leur case de départ (après écartement) soit sur leur case d'arrivée. Ainsi pour beaucoup de problèmes, si l'écartement n'est pas parfait, il n'existe pas d'ordre de priorité convenable. Une piste de recherche pourrait donc être d'introduire des positions intermédiaires ce qui permettrait de résoudre le fait qu'aucune solution n'existe dans certains cas, tout en réduisant le nombre de groupes. Réduire le nombre de groupes de priorité permettrait de réduire le makespan car on fait partir un groupe après l'autre, une alternative serait d'explorer plus en détails le fait de faire se déplacer plusieurs groupes de priorités ensemble (nous avions notamment exploré l'idée de faire bouger tous robots en même temps sauf que robots seuls les robots prioritaires avaient le droit d'atteindre leur case d'arrivée. Les robots non prioritaires avaient un potentiel centré en leur case d'arrivée qui les repoussait sur un faible rayon. Ainsi ils se dirigeaient dans la bonne direction, sans pour autant bloquer leur case d'arrivée. Cette piste fut malheureusement sans succès). Enfin pour pallier les problèmes de priorité, on pourrait imaginer que les robots non prioritaires puissent aller sur leur case d'arrivée mais qu'une fois arrivés, ils puissent encore bouger pour laisser leur place à d'autres.