Project of Computer science: Coordinated Robot Motion Planning

Didier Ngatcha, Armel Zebaze

January 2021

1 Introduction

Le but de ce projet proposé dans le cadre du cours INF421 est d'implémenter une solution du probleme connu sous le nom de "coordinated robot planning problem". Le probleme est formulé comme suit :

On dispose de n robots placés dans le plan à des positions connues $S = \{s_1, ..., s_n\}$ et l'on souhaite les deplacer jusqu'a leurs objectifs respectifs $T = \{t_1, ..., t_n\}$. L'on veut élaborer une stratégie pour déplacer chaque robot de son point de départ vers son point d'arrivée en respectant un certain nombre de contraintes. En une unité de temps, chaque robot peut se déplacer de sa position (x, y) vers l'une des positions adjacentes (Nord, Sud, Est ou Ouest) i.e $(x, y+1), (x, y-1), (x+1, y)$ ou $(x-1, y)$ sous reserve que ces positions soient inoccupées ou en voie de l'être. Il peut aussi y avoir des obstacles qui bloquent l'accès à certaines cases. On considèrera une solution valide si elle vérifie les conditions suivantes :

- Un robot se déplace au plus une fois pendant une phase de mouvement (le robot peut rester sur place).
- Les robots ne peuvent pas être dans des cases obstacles.
- Deux robots ne peuvent pas éssayer de rentrer dans la meme cellule pendant la même phase de mouvement.
- Si un robot se déplace vers une case dans une phase de mouvement, soit elle est vide, soit le robot à l'intérieur se déplace dans la même phase de mouvement.

Les robots peuvent se mouvoir hors du plan et le problème peut être résolu de différentes facons selon la fonction à optimiser, ici on a le choix entre : the makespan and the total distance.

2 Description et Analyse de la Solution Algorithmique Proposée

Nous avons opté pour un algorithme global, sachant que la minimisation du makespan fait intervenir des considérations de distance. Nous avons abouti à une solution partielle, car l'algorithme fonctionne pour toutes les instances sans obstacles, mais hélas pour certaines instances avec obstacles, on n'a pas de solution. Notre algorithme est découplé, en ce sens où les robots sont déplacés séparément les uns des autres.

2.1 Optimisation de la distance

Soit $i \in [1, n]$. le robot i est placé d'entrée de jeu au point (x_i, y_i) et il souhaite se rendre au point de coordonnées (s_i, t_i). La distance à vol d'oiseau entre ces deux points vaut : $d[i] = |x_i - s_i| + |y_i - t_i|$.
En absence d'obstacles cette distance est la distance optimale parcourue par un robot de son point de départ au point d'arrivée. En présence d'obstacles cette distance peut être plus grande. On cherche à minimiser la quantité : $Q = \sum_{k=1}^{n} \tilde{d}[k]$ où $\tilde{d}[k]$ représente la distance parcourue par le robot k de son point de départ au point d'arrivée calculée par notre algorithme. Pour minimiser cette distance on peut éssayer de minimiser pour chaque robot la distance qu'il parcourt pour atteindre son objectif.
L'on dispose du nombre n de robots et des coins $x_{min}, x_{max}, y_{min}$ et y_{max} du plan, la stratégie que l'on a adopté pour résoudre ce problème est la suivante : considérer la matrice D de taille (N, M, n) avec $N = x_{max} - x_{min} + marge$ et $M = y_{max} - y_{min} + marge$.
En effet la marge est la pour représenter la possibilité pour les robots de se déplacer hors du plan initial, la prendre nulle ne garanti pas l'existence d'une solution.
Il s'agit en effet d'un hyperparamètre, fixé dans le code, et qui ne donne pas de solution pour toutes les valeurs.
Pour tous $i \in [0, N-1], j \in [0, M-1], k \in [0, n-1]$ On a $D[i][j][k]$ = la distance minimale entre le point d'arrivée du robot numéro k et le point de coordonnées (i, j). On pourra écrire la relation de réccurence suivante :
$D[i][j][k] = 1 + min\{D[i+1][j][k], D[i-1][j][k], D[i][j+1][k], D[i][j-1][k]\}$.
Si on note (i_k, j_k) les coordonnées du point d'arrivée du robot k, $D[i_k][j_k][k] = 0$, on initialisera tous les membres de la matrice à ∞.
Pour calculer toute ces distances on utilisera l'algorithme du parcours en largeur sur le graphe : (V, E) avec $V = \{(i, j) \in [0, M-1][0, N-1] - \{Obstacles\}\}$ et E l'ensemble des arcs entre les points adjacents de V. On peut directement constater que l'algorithme (BFS) laisse la valeur ∞ pour les positions correspondantes a des obstacles, ce qui est tout à fait cohérent avec ce que l'on veut.

Algorithm 1 Initialisation(): Initialisation de D par l'algorithme BFS

Choix de $Marge$
Calcul de $N = x_{max} - x_{min} + Marge$ et $M = y_{max} - y_{min} + Marge$
Initialisation de D à ∞ pour tout (i, j, k)
for k allant de 0 à $n - 1$ **do**
  $D[i_k][j_k][k] = 0$
end for
for k allant de 0 à $n - 1$ **do**
  faire BFS pour $D[.][.][k]$ et (i_k, j_k)
end for

D'autre part on souhaite pouvoir connaitre en temps réèl l'état des cases de notre plan (si elles sont occupées ou pas). Et pour ce faire on a deux objets :

- Une matrice L MxN telle que $L[i][j] = 1 \iff$ la case (i, j) est occupée par un obstacle et nulle partout ailleurs.
- Une matrice P MxN telle que $P[i][j] = 1 \iff$ la case (i, j) est occupée par un obstacle ou un robot et nulle partout ailleurs. Elle représente vraiment l'état du plan en temps réèl.

La matrice L est utile pour l'implémentation du BFS car elle permet de savoir quels sont les voisins d'une case (i, j) qui ne sont pas occupés par des obstacles en $\mathcal{O}(1)$.
L'utilité de P est évidente, c'est un objet qui doit être mis à jour à chaque déplacement de robot.
On utilise une autre structure, un tableau *indices* 1xn qui est initialisé à $[0, 1, 2, ..., n - 1]$ et qui permet de savoir aucours de l'algorithme quel robot n'a pas encore atteint son point d'arrivée. La boucle principale du programme porterait donc sur le booléèn $indices.isEmpty()$, dépendament de sa valeur l'on va effectuer ou non un $ComputeOneStep()$.
On peut des lors faire une remarque, un robot a de base au moins 1 voisin qui n'est pas un obstacle. C'est - à - dire qu'un robot ne peut pas être retenu entre 4 obstacles, autrement il n'en sortirait jamais.
L'algorithme fonctionne comme suit :

- Tant que le tableau *indices* est non vide, on traite les robots simultanément.
- Étant donné un robot k, on peut trouver ses voisins (hors obstacles).À noter que ceux - ci peuvent être occupés par d'autres robots.
- S'il existe on choisit le voisin non occupé qui possède la distance par rapport au point d'arrivée (i_k, j_k) la plus faible et on se deplace vers lui. On met à jour P. Il peut arriver que la case où se situait le robot précedemment soit la seule case disponible avec la plus faible distance par rapport au point d'arrivée. Dans ce cas, le robot se retrouverait à chaque itération à osciller entre deux cases. Pour éviter cela, une fois que le robot

k a choisi la prochaine case voisine où il se déplacera, dans $D[.][.][k]$ on met la valeur de la case correspondant à sa position actuelle à ∞. Vu qu'à chaque itération on choisira la distance la plus faible parmi ses voisins, cela disqualifie naturellement la case précédente en tant que prochaine destination potentielle, évitant ainsi les retours en arrière inutiles ou les passages multiples par la même case. Deux cas se présentent ensuite:

  - Si le robot arrive à son point d'arrivée, alors on met à jour L. ce qui sous - entend qu'un robot arrivé à destination est traité comme un obstacle puisqu'on ne le bougera plus. Ensuite, de nouveau à l'aide du BFS on actualise D de sorte à prendre en compte le nouvel obstacle; on n'actualise alors que les tableaux $D[.][.][k]$ correspondant aux robots pas encore arrivés à destination. Ceci est explicité dans l'algorithme 2.
  - Sinon on continue vers le traitement des robots restants.

- Sinon, i.e si tous les voisins de k sont soit des obstacles, soit des robots (avec au moins un robot parmi, qui peut ou pas être arrive deja à destination). Dans ce cas le robot n'a nulle part où aller. Comme précedemment on réactualise D pour ce robot là. Il reste sur place et attend la prochaine itération.

L'algorithme 4 ci-dessous permet de donner le mouvement d'un robot par étape. $voisin_k$ représente les coordonnées de tous les voisins du robot k à la position (x_k, y_k) qui ne sont pas des obstacles. L'algorithme suivant permet de mettre à jour D comme mentionné plus haut à partir des coordonnées actuelles (x_k, y_k) du robot k.

Algorithm 2 Reinitialisation(): Réinitialisation de D par l'algorithme BFS pour les robots encore en chemin

Initialisation de D à ∞ pour tout (i, j, k) tels que le robot k ne soit pas encore arrivé
for k allant de 0 à $n-1$ **do**
  if $(i_k, j_k) \neq (x_k, y_k)$ **then**
    $D[i_k][j_k][k] = 0$
  end if
end for
for k allant de 0 à $n-1$ **do**
  if $(i_k, j_k) \neq (x_k, y_k)$ **then**
    faire BFS pour $D[.][.][k]$ et (i_k, j_k)
  end if
end for

Algorithm 3 ChoisirDeplacement(*destination*, x_k, y_k): Choisit le mouvement en fonction de la prochaine destination du robot et de sa position actuelle

```
if destination = null then
  Reinitialisation()
  return Immobile
else
  D[x_k][y_k][k] = ∞
  if destination[0] > x_k then
    return Déplacer vers la Droite
  else if destination[0] < x_k then
    return Déplacer vers la Gauche
  else
    if destination[1] > y_k then
      return Déplacer vers le Haut
    else if destination[1] < y_k then
      return Déplacer vers le Bas
    end if
  end if
end if
```

Algorithm 4 Mouvement(k, x_k, y_k, $voisin_k$): Définit le déplacement de chaque robot pour une itération

```
if voisin_k est vide then
  return Immobile
else
  min = ∞
  for (x, y) ∈ voisin_k do
    if min > D[x][y][k] and P[x][y] = 0 then
      min = D[x][y][k]; destination = (x, y)
    end if
  end for
end if
return ChoisirDeplacement(destination, x_k, y_k)
```

Notre algorithme suit la logique des *greedy algorithms* mais on a pas de garanti d'optimalité. Une borne inferieure triviale de la distance totale parcourue par les robots est la distance à vol d'oiseau totale entre depart et arrivées. i.e :

$$Q \geq \sum_{i=1}^{n} |x_i - s_i| + |y_i - t_i| \tag{1}$$

Une information utile serait l'ordre d'approximation de notre solution par rapport à cette valeur (qui ne tient compte ni des obstacles, ni compte de l'externalité

négative des robots les uns sur les autres) i.e le plus petit entier z tel que :

$$Q \leq z \sum_{i=1}^{n} (|x_i - s_i| + |y_i - t_i|) \tag{2}$$

On ne peut rien affirmer avec certitude mais sur les tests que nous avons menés on constate que pour notre solution de z est de l'ordre de 2. D'autre part, en terme de complexité on a :

- Pour le stockage on a $\mathcal{O}(n * M * N)$
- Pour le temps, c'est plus compliqué à indiquer. On a initialement n BFS correspondant à $\mathcal{O}(n * M * N)$. Cependant le nombre d'itérations n'est pas connu à l'avance, et pour notre algorithme, cela ne depend pas de manière polynomiale des entrées de l'algorithme.

2.2 Optimisation du Makespan

Il s'agit ici de minimiser le temps mis pour que tous les robots arrivent à destination i.e le nombre d'itérations au bout duquel $indices = []$. Ici aussi on possède une borne inférieure triviale. En effet, en absence d'obstacles, si on note ξ ce nombre d'itérations, on a :

$$\xi \geq max_{i \in [1 \ldots n]} (|x_i - s_i| + |y_i - t_i|) \tag{3}$$

En effet cela s'explique par le fait que dans le cas idéal, placer le robot le plus éloigné de sa cible est la tache qui prend le plus de temps. Mais une fois de plus dans ce problème, on a affaire à des obstacles et à l'influence mutuelle des robots. Pour minimiser le makespan, il est essentiel de *bien* déplacer le plus de robots possibles en une itération. et c'est en quelque sorte ce que nous éssayons de faire dans notre algorithme. Le pire pour le *makespan* serait de bouger sequentiellement chaque robot de son point de départ à son point d'arrivée (Cette méthode est trop naive pour calculer à tous les coups une solution valide car les robots arrivés peuvent bloquer irreversiblement des routes d'autres robots). Dans notre algorithme on bouge au maximum les robots, en éssayant de ne pas s'éloigner arbitraiment des points que l'on veut atteindre. Une fois de plus on pourrait s'intéresser a la qualité de notre solution par rapport à la borne trouvée, i.e la constante z telle que :

$$\xi \leq z * max_{i \in [1 \ldots n]} (|x_i - s_i| + |y_i - t_i|) \tag{4}$$

En se basant sur les cas sur lesquels on a testé notre algorithme, on trouve que z est de l'ordre de 3.

2.3 Implementation

Cette partie fait un tour des structures utilisées, cela ne concerne en fait que deux types de structures de données :

- Une matrice est représentée pour nous en Java par une table de Hachage de tables de Hachage. De sorte que pour tout i, Matrice[i] est une table de Hachage dont les clés sont les indices de la ligne (j), et les valeurs, les éléments de la matrice. L'avantage est de pouvoir accéder au contenue d'une case directement par ses coordonnées (x, y) qui ne sont pas necessairement positives.

- Pour les listes d'éléments on utilise tres souvent des $ArrayList$, cela inclus les tableaux de dimension 3, qui sont traités comme des listes de matrices.

Dans l'implementation en Java, on L represente $Libre$, P positions, D quant à lui est $listeDamier$. A lui s'ajoute $listeCouleur$.

2.4 Quelques Resultats

On obtient les resultats suivant pour nos tests :

1. socge2021 : marge = 5, makespan = 80, distance-totale = 3428, distance = 2086.

2. election109 : marge = 4, makespan = 88 , distance-totale = 3752, distance = 1816.

3. king-94 : marge = 4, makespan = 78, distance-totale = 2991, distance = 1827.

4. small-free-00-10x10-30-30 : marge = 4, makespan = 38, distance-totale = 337, distance = 187.

5. small-free-40 : marge = 6 , makespan = 59, distance-totale = 704, distance = 266.

6. galaxy-cluster-80 : marge = 6, makespan = 46, distance-totale = 1303, distance = 845.

7. small-000-10x10-20-10 : marge = 4, makespan = 26, distance-totale = 116, distance = 86.

8. small-001-10x10-40-30 : marge = 2, makespan = 54, distance-totale = 562, distance = 240.

9. redblue-00000-20x20-30-113 : marge = 4, makespan = 149, distance-totale = 4074, distance = 1510.

10. buffalo-000-25x25-20-63 : marge = 3, makespan = 96, distance-totale = 2455, distance = 1149.

11. small-microbe ne donne pas de resultat

Instance	Marge	Makespan	Distancetotale	Borne distance	$\frac{Distancetotale}{Bornedistance}$
socge2021	5	80	3428	2086	1.64
election-109	4	88	3752	1816	2.07
king-94	4	78	2991	1827	1.64
small-free-00-10x10-3–30	4	38	337	187	1.80
small-free-40	6	59	704	266	2.65
galaxy-cluster-80	6	46	1303	845	1.54
small-000-10x10-20-10	4	26	116	86	1.35
small-001-10x10-40-30	2	54	562	240	2.34
redblue-00000-20x20-30-113	4	149	4074	1510	2.7
buffalo-000-25x25-20-63	3	96	2455	1149	2.14

Faute de capacité des machines à notre disposition, nous n'avons pas pu tester notre algorithme sur les plus grandes instances du problème.

3 Conclusion

Pour résoudre le problème auquel nous étions confronté, nous avons opté pour une approche qui consistait à gérer chaque robot individuellement en le rapprochant au mieux de son objectif. A priori cette méthode ne semble pas garantir l'existence de solution comme les tests le montrent. On peut imaginer des soucis tels que le cas où un robot est entouré par des cibles statiques et que son point d'arrivée est justement à l'extérieur de ce "cercle". En traitant les robots les uns indépendamment des autres, on ne se prémuni pas nécéssairement de ce cas. La solution triviale qui consiste quant à elle à envoyer sequentiellement chaque robot de son point de départ au point d'arrivée coure le même risque. Nous avons opté pour un greedy, mais d'autres paradygmes pourraient être plus appropriés.