ROB316 - TD3

BRAMBILLA Davide Luigi - GOMES DA SILVA Rafael

16 Décembre 2019

1 Le cours

L'objective du cours est de présenter des algorithmes qui traitent de la planification du chemin, c'est à dire la recherche d'un chemin entre la position courante et la position désirée.
Le concept de **plan** a été présenté comme la suite de noeuds et de liens à suivre associée aux actions associées à chaque noeuds.
Il a été mis en évidence l'importance de la **discrétisation de l'espace** où le robot devra se déplacer afin de pouvoir traiter le problème sous forme de graphe et différents méthodes ont été présentées. Les positions parcourues par le robot constituent l'**espace des configurations**.
En suite, nous avons abordé le problème de la **recherche de chemin dans un graphe** qui est traité par des algorithmes qui estiment le *coût* (associé aux *noeuds* et aux *liens*) des possibles chemins et choisissent lequel à moindre *coût*. Parmi ces algorithmes on trouve lequel de **Dijkstra** qui vérifie les poids associés à tous les points et des variantes comme le **Wavefront planner**, qui traite les cas où les poids des liens sont similaires, le **A*** que nous allons traiter lors du TD et le **D*** qui est performante si l'environnement peut changer.
Enfin, il a été mentionné le fait que pour la planification il est recherché non seulement le chemin le plus court mais il peut être intéressant de rechercher le chemin le plus fiable comme dans le cas de **Coastal Navigation**.

2 TP

2.1 Q1 - Le méthode A* pour la recherche de plus court chemin

L'algorithme A* est un algorithme utilisé pour la recherche du plus court chemin entre un nœud initial et un nœud final l'intérieur à un graphe en permettant de ne pas explorer toutes les possibles chemins. Pour cela, cet algorithme représente une extension de l'algorithme de *Dijkstra* qui s'occupe d'explorer tous les possibles parcours avant d'en choisir le meilleur parmi les possibles.
L'idée de l'algorithme est d'essayer de se rapprocher de la destination finale et de privilégier les possibilités plus proches de la destination, en mettant de côté toutes les autres. D'abord il va donc se diriger vers les chemins les plus directs: si ces chemins ne permettent pas d'arriver à la solution finale, il examinera les autres solutions mises à coté.

Ci-dessous nous allons étudier étape par étape l'algorithme A*.
Dans un premier moment, nous allons définir notre environnement en choisissant la position des obstacles, le point de départ et le point d'arrivée.
Nous allons donner un poids infini aux obstacles de façon qu'ils soient évités.

Fichier 1: A*.m

```
%Define Number of Nodes
xmax = 80;
ymax = 20;

%Start and Goal
start = [10,ymax/2];
goal = [xmax,ymax/2-3];

%Nodes
MAP = zeros(xmax,ymax);

%To define objects, set their MAP(x,y) to inf
MAP(40,5:15) = inf;
MAP(35:40,5) = inf;
MAP(35:40,15) = inf;
MAP(60:62,1:7) = inf;
MAP(60:62,9:12) = inf;
```

En suite nous allons calculer une heuristique (méthode de calcul qui fournit rapidement - en temps polynomial - une solution réalisable, pas nécessairement optimale) qui permet d'estimer la distance d'un point au but en utilisant la fonction *norm()* qui calcule la distance euclidienne. En fait, la matrice H représente le coût pour aller du noeud courant au point désiré.

Fichier 2: A*.m

```
%Compute Heuristic for each node%
weight = 3;

for x = 1:size(MAP,1)
    for y = 1:size(MAP,2)
        if(MAP(x,y)~=inf)
            H(x,y) = weight*norm(goal-[x,y]);
            G(x,y) = inf;
        end
    end
end

%Set node Traversal cost%
W = zeros(size(MAP));
% W = 2./bwdist(W);
```

Ici nous allons initialiser les conditions initiales et les tableaux des nodes que nous allons parcourir: la *liste ouverte* qui va contenir tous les noeuds étudiés et *liste fermée* contiendra tous les noeuds qui ont été considéré dans la recherche du chemin de solution. De plus, ici nous allons définir les matrices G et F qui représentent, respectivement, le coût pour aller du point de départ au noeud courant et la somme des coûts précédents. En particulier, la *liste ouverte*, pour chaque noeud inclut les coordonnées du noeud courant, les matrices G et F associées et le noeud parent.

Fichier 3: A*.m

```
%initial conditions%
G(start(1),start(2)) = 0;
F(start(1),start(2)) = H(start(1),start(2));

closedNodes = [];
openNodes = [start G(start(1),start(2)) F(start(1),start(2)) 0]; %[x y G F cameFrom]
```

En suite l'algorithme traite tous les noeuds à l'intérieure du tableau *openNodes* et il les déplace en *closedNodes* afin de pouvoir distinguer entre les noeuds à traiter et lesquels déjà traités.

Nous pouvons résumer le fonctionnement de l'algorithme au travers plusieurs étapes: à partir du noeud courant nous allons considérer les noeuds voisins et, après avoir vérifié que le noeud ne représente pas un obstacle où soit déjà dans *liste fermée*, on les ajoute à la *liste ouverte* en lui associant le noeud courant comme parent. En suite, on cherche le meilleur noeud dans la *liste ouverte* et on le déplace dans la *liste fermée*. Finalement, après avoir mis à jour les valeurs des matrices G et F pour le nouveau noeud, on recommence l'algorithme jusqu'au moment où le noeud courant soit le noeud final de destination.

Fichier 4: A*.m

```
while(~isempty(openNodes))

    pause(1e-5)

    %find node from open set with smallest F value
    [A,I] = min(openNodes(:,4));

    %set current node
    current = openNodes(I,:);
    color = cmap(min([round(G(current(1),current(2)))+1 120]),:);
    plot(current(1),current(2),'o','color',color,'MarkerFaceColor',color)

    %if goal is reached, break the loop
    if(current(1:2)==goal)
        closedNodes = [closedNodes;current];
        solved = true;
        break;
    end

    %remove current node from open set and add it to closed set
    openNodes(I,:) = [];
    closedNodes = [closedNodes; current];

    %for all neighbors of current node
    for x = current(1)-1:current(1)+1
        for y = current(2)-1:current(2)+1

            %if out of range skip
```

```
            if (x<1||x>xmax||y<1||y>ymax)
                continue
            end

            %if object skip
            if (isinf(MAP(x,y)))
                continue
            end

            %if current node skip
            if (x==current(1)&&y==current(2))
                continue
            end

            %if already in closed set skip
            skip = 0;
            for j = 1:size(closedNodes,1)
                if(x == closedNodes(j,1) && y==closedNodes(j,2))
                    skip = 1;
                    break;
                end
            end
            if(skip > 0)
                continue
            end


            A = [];

            %Check if already in open set
            if(~isempty(openNodes))
                for j = 1:size(openNodes,1)
                    if(x == openNodes(j,1) && y==openNodes(j,2))
                        A = j;
                        break;
                    end
                end
            end


            newG = G(current(1),current(2)) + round(norm([current(1)-x,current(2)-y]),1) + W(x,y);

            %if not in open set, add to open set
            if(isempty(A))
                G(x,y) = newG;
                newF = G(x,y) + H(x,y);
                newNode = [x y G(x,y) newF size(closedNodes,1)];
                openNodes = [openNodes; newNode];
                plot(x,y,'x','color','b')
                continue
            end

            %if no better path, skip
            if (newG >= G(x,y))
                continue
            end

            G(x,y) = newG;
            newF = newG + H(x,y);
            openNodes(A,3:5) = [newG newF size(closedNodes,1)];
        end
    end
end
```

2.2 Q2 - Le paramètre *weight = 0*

Lorsque nous définissons la variable $weight = 0$, l'algorithme qui est exécuté est effectivement l'algorithme de Dijkstra. Cela est dû au fait que nous n'allons pas considérer l'heuristique donné par la matrice H car l'heuristique sera initialisé à 0 dans le calcul de H pour chaque noeud.
De cette façon nous initialiserons aussi la matrice F à zéro de façon que tous les noeuds auront un coût de 0 pour arriver au noeud finale
Dans ce cas, nous n'allons considérer toutes les points avant d'arriver à la solution désirée.
Le graphe obtenu est presenté sur la Figure 2.1 où il est, effectivement, possible noter que tous les noeuds ont été pris en compte lors de la recherche du chemin le plus court.

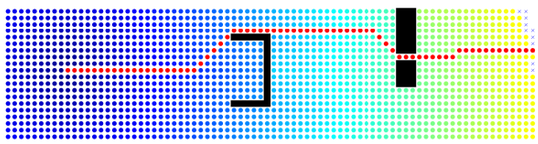


Figure 2.1: Résultat avec *weight = 0*

2.3 Q3 - La variation du paramètre *weight*

Nous allons maintenant changer la pondération *weight* que l'on donne à l'heuristique. Plus on donnera une valeur élevé, plus nous allons donner importance à l'heuristique et plus vite sera notre algorithme. Par contre, en donnant une valeur trop élevé on pourrait perdre le parcours optimale car nous ne considérerons pas tous les solutions possibles car dans la choix du chemin, l'importance de l'heuristique sera prédominante. Nous allons essayer trois différents valeurs: 1, 3 et 1.4 et nous allons reporter les résultats que nous avons obtenus ci-dessous.

2.3.1 Le paramètre *weight = 1*

Dans ce cas, nous allons avoir une réduction dans le temps d'exécution de l'algorithme d'un facteur 3 (Tableau 2.1) et le même résultat en terme de longueur du chemin par rapport à l'algorithme de Dijsktra. Il est possible d'observer ici l'amélioration donnée par l'algorithme A*: le fait de ne pas considérer tous les points augmente considérablement la vitesse d'exécution de l'algorithme. La Figure 2.2 montre le résultat obtenu en utilisant $weight = 1$.

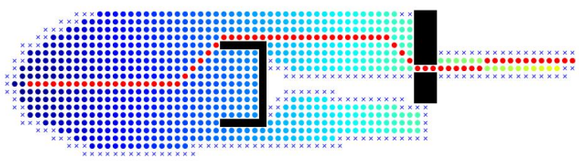


Figure 2.2: Résultat avec *weight = 1*

2.3.2 Le paramètre *weight = 3*

Dans ce cas, nous pourrons analyser le fait que, en augmentant la valeur de pondération que on donne à l'heuristique, nous aurons un algorithme encore plus rapide mais qui ne trouve pas le parcours optimale car il donne *"trop"* d'importance à l'heuristique et il explore un nombre mineur de noeuds qui ne garantit pas de trouver la meilleure solution. La Figure 2.3 montre le résultat obtenu en utilisant $weight = 3$.

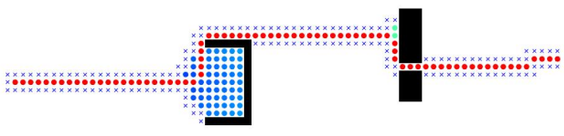


Figure 2.3: Résultat avec *weight = 3*

2.3.3 Le paramètre *weight = 1.4*

Enfin, nous avons analysé le cas avec *weight = 1.4* qui donne le meilleur résultat en permettant de réduire d'un facteur 16 la rapidité d'exécution de l'algorithme et en restant avec la longueur du chemin la plus bas possible. Cette amélioration est donné par le fait que la valeur 1.4 est proche de la valeur de $\sqrt{2}$ qui est la valeur associé à la norme euclidienne et permet de utiliser l'information donné par l'heuristique au maximum. La Figure 2.4 montre le résultat obtenu en utilisant $weight = 1.4$.

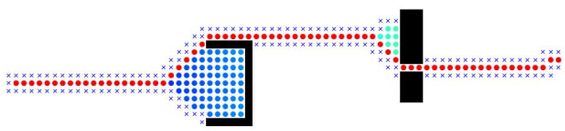


Figure 2.4: Résultat avec *weight = 1.4*

2.3.4 Tableau pour les performances dans les différents cas

Dans le Tableau 2.1 suivant, nous avons reporté la longueur du chemin trouvé comme solution ainsi que le temps que l'algorithme a pris pour donner la solution:

Tableau 2.1: Tableau pour les performances, avec $weight = 0$

	Path length	Elapsed time(sec)
Weight = 0	74.5563	64.793019
Weight = 1	74.5563	23.801684
Weight = 1.4	74.5563	4.624237
Weight = 3	78.0711	4.616933

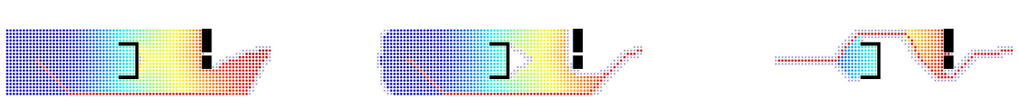
Il est possible d'observer que la meilleure solution est laquelle avec *weight = 1.4* étant donné que donne le meilleur résultat trouvé aussi dans le cas de algorithme de Dijkstra dans le temps mineur. Il est légèrement plus lent du cas avec *weight = 3* mais, par rapport à ce dernier, il donne un résultat meilleur en terme de longueur de chemin.

2.4 Q4 - Les poids associés aux noeuds

Dans cette section, il nous est demandé d'agir directement sur les poids donnés aux nodes. Il nous est demandé de modifier ces poids (c'est à dire la variable W) afin que le chemin choisi ne soit pas proche des obstacles.

Il nous est suggéré d'utiliser la fonction *bwdist(MAP)* qui permet de réaliser une transformée en distance, c'est à dire la calcul, pour chaque position libre de la carte, la distance à l'obstacle le plus proche. Étant donnée que nous voudrions rester loin des obstacles, nous allons donner à la variable W une valeur proportionnelle à l'inverse de *bwdist(MAP)*. Nous avons choisi de donner une valeur de $\frac{5}{bwdist(MAP)}$ afin de pénaliser les lieux étroits comme le trou de l'obstacle sur la gauche.

Dans la Figure 2.5, nous avons reporté les graphiques obtenus dans les cas de variation de la variable *weight* comme dans le point précèdent.



(a) Résultat avec *weight = 1* (b) Résultat avec *weight = 1.4* (c) Résultat avec *weight = 3*

Figure 2.5: Les résultats avec les différents valeurs de la variable *weight* avec un coefficient de 5

Dans ce cas aussi nous allons reporter le Tableau 2.2 avec les performances de l'algorithme avec les différents valeurs.

Tableau 2.2: Tableau pour les performances, avec $weight \neq 0$

	Path length	Elapsed time(sec)
Weight = 1	79.5269	50.80864
Weight = 1.4	79.5269	38.51582
Weight = 3	84.3553	7.54939

Il est possible de voir qu'on peut traire les mêmes conclusions du cas précèdent: une différence est dans le fait que l'algorithme avec 1.4 dans ce cas prends plus de temps à trouver la solution et l'amélioration n'est pas important comme dans le cas précèdent.

Finalement, nous allons reporter le parcours désiré et obtenu dans le cas précédents:

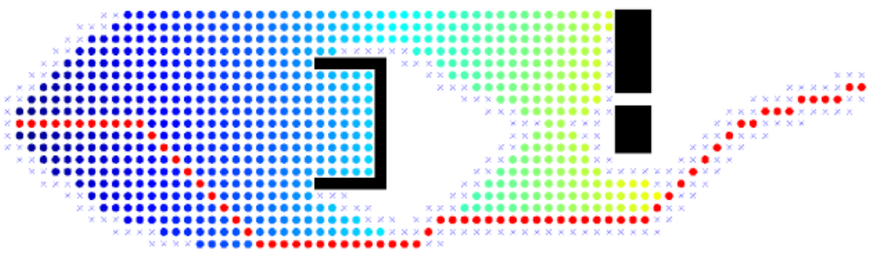


Figure 2.6: Résultat espéré

2.4.1 Variation de la valeur de W

Dans ce paragraphe nous allons expliquer la raison pour laquelle nous avons choisi une valeur de 5 comme coefficient associé à la valeur de W.

Pour faire cela nous allons comparer les résultats précédents avec lesquels obtenus pour une valeur du coefficient de 2:

(a) Résultat avec *weight = 1* (b) Résultat avec *weight = 1.4* (c) Résultat avec *weight = 3*

Figure 2.7: Resultats avec les différents valeurs de la variable *weight* avec un coefficient de 2

Ici il est possible de voir la raison pour laquelle nous avons choisi d'augmenter la valeur du coefficient de 2 à 5: dans ce cas, avec la valeur de *weight = 3*, le parcours choisi n'est pas lequel que on recherche car la solution proposée passe dans le trou au milieu de l'obstacle de droite et se rapproche trop des obstacles.
Cela est dû au fait que les poids associés aux noeuds ont une pondération mineure par rapport à laquelle donnée par l'heuristique qui nous ramène à donner plus d'importance au fait d'arriver dans la position désirée plutôt que à éviter les obsacles.

2.5 Q5 - L'influence de l'environnement

Dans le dernier point, nous allons étudier l'influence de l'environnement dans l'algorithme.

Pour faire cela nous allons documenter le code pour la génération casuelle des obstacles à l'intérieure de notre plan et nous allons considérer les cas avec *weight = 0* et *weight = 1.4*. Nous allons étudier un cas avec 5, 10 et 15 obstacles et nous avons choisi de maintenir la pondération .

2.5.1 Cas avec 5 obstacles

Tableau 2.3: Longueur du chemin et temps écoulé avec 5 obstacles

	Path length	Elapsed time(sec)
Weight = 0	82.4558	59.4719
Weight = 1.4	80.1127	49.5655

(a) Résultat avec *weight = 0*

(b) Résultat avec *weight = 1.4*

Figure 2.8: Résultats avec un plan qui possède 5 obstacles

2.5.2 Cas avec 10 obstacles

Tableau 2.4: Longueur du chemin et temps écoulé avec 10 obstacles

	Path length	Elapsed time(sec)
Weight = 0	83.6274	113.7665
Weight = 1.4	83.6274	55.3414

(a) Résultat avec *weight = 0*

(b) Résultat avec *weight = 1.4*

Figure 2.9: Résultats avec un plan qui possède 10 obstacles

2.5.3 Cas avec 15 obstacles

Tableau 2.5: Longueur du chemin et temps écoulé avec 15 obstacles

	Path length	Elapsed time(sec)
Weight = 0	80.1127	51.6266
Weight = 1.4	79.5269	42.7907

(a) Résultat avec *weight = 0*

(b) Résultat avec *weight = 1.4*

Figure 2.10: Résultats avec un plan qui possède 15 obstacles

2.5.4 Conclusion

Il est possible de voir que l'avantage donné par l'algorithme **A*** est dans le fait que nous n'allons pas considérer toutes les points présentes dans le plan mais nous basons notre recherche sur des chemins que nous allons privilégier et qui sont donnés par les informations dérivant de l'heuristique.

Même s'il n'est pas immédiat comparer les résultats, étant donnée que les obstacles sont générés aléatoirement à chaque exécution et donc que les environnements que l'algorithme doit traiter sont différents, nous pouvons noter que **A*** a généralement des performances meilleures par rapport à l'algorithme de Dijkstra.

L'algorithme **A*** apporte l'avantage le plus grand dans le cas les obstacles sont disposés de façon que il y a un chemin évident qui nous ramène de la position initiale à la position finale: comme cela nous pourrons traiter un nombre significativement mineure des noeuds et augmenter la rapidité de calcul.