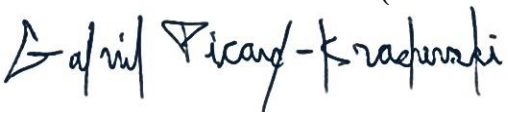
Titre: Title:	Optimisation des algorithmes de planification et d'évitement d'obstacle d'un bras robotique
Auteurs: Authors:	Gabriel Picard-Krashevski
Date:	2018
Type:	Rapport / Report
Référence: Citation:	Picard-Krashevski, G. (2018). Optimisation des algorithmes de planification et d'évitement d'obstacle d'un bras robotique. (Internship Report - Graduate Studies). https://publications.polymtl.ca/3548/

ÉCOLE
POLYTECHNIQUE
M O N T R É A L

Rapport de stage

OPTIMISATION DES ALGORITHMES DE PLANIFICATION ET D'ÉVITEMENT D'OBSTACLE D'UN BRAS ROBOTIQUE

Présenté par
Gabriel Picard-Krashevski (1790381)



Directeur de projet
Sofiane Achiche

Montréal, le 14 août 2018

TABLE DES MATIÈRES

LISTE DES FIGURES

INTRODUCTION ET DESCRIPTION DU PROBLÈME

Le projet de l'étudiant Cédric Leblond Ménard et de son professeur Sofiane Achiche consiste à concevoir un système mécatronique permettant le contrôle d'un bras robotique (développé par la compagnie Kinova) à l'aide du regard d'un utilisateur. Étant déjà conçu pour subvenir aux besoins des gens en fauteuil roulant, ce bras pourra désormais, grâce à ce système, être utilisé par ceux possédant des difficultés motrices. Toutefois, beaucoup de recherches sont encore à compléter pour optimiser l'efficacité du système. L'une d'elles consiste à optimiser l'algorithme de planification de trajectoire et d'évitement d'obstacles. Cet algorithme devra permettre au bras de se rendre le plus rapidement possible à l'objectif défini par le regard de l'utilisateur sans toutefois accrocher un obstacle. De plus, il est nécessaire que le temps de calcul pour un tel algorithme soit le plus court possible.

Ce rapport fera donc un résumé ainsi qu'une évaluation (visant le projet) des différents algorithmes utilisés en industrie pour la planification de trajectoire et l'évitement d'obstacles. Les évaluations seront basées sur le fait que l'environnement sera principalement statique puisque le bras sera conçu pour des tâches simples, que le système devra déterminer le plus rapidement possible le chemin le plus court à parcourir pour se rendre à l'objectif et que le déplacement du bras devra être le plus fluide possible. Finalement, celui-ci se terminera avec diverses suggestions pour permettre d'optimiser le mieux possible l'algorithme de planification de trajectoire pour le bras robotique du projet.

ALGORITHME DE DIJKSTRA

L'algorithme de Dijkstra est un algorithme de planification de trajectoire. Il est conçu pour déterminer le chemin le plus rapide pour se rendre d'un point à un autre. Celui-ci utilise deux listes, surnommées respectivement la Liste Ouverte et la Liste Fermée, et un système de coût qui représente la difficulté pour se rentre jusqu'à un point (cette difficulté est habituellement représentée par la distance à parcourir) [1]. Dans la Liste Ouverte, on retrouve la liste des points dont les alentours devront être explorés (placés selon leur priorité). Dans la Liste Fermée, on retrouve la liste des points dont les alentours ont été explorés [1].

Pour ce qui est des étapes de calculs, celui-ci donne initialement un coût infini à tous les points du plan comme le montre la figure 1. Les points noirs représentent ici les obstacles. Par la suite, l'algorithme change le coût du point de départ pour zéro (puisqu'aucun déplacement n'a été réalisé) et le met dans la Liste Ouverte (voir la figure 2). Ensuite, il choisit le point dans la Liste Ouverte dont le coût est le plus faible et explore les points alentour. Si ceux-ci ne sont pas des obstacles, la valeur de leur coût est changée et ceux-ci sont ajoutés dans la Liste Ouverte. De plus, chaque point gardera aussi comme information leur origine (ils garderont les coordonnées du point voisin qui a permis de se rendre jusqu'à eux). Lorsque l'exploration est terminée, le point de la Liste Ouverte est déplacé dans la Liste Fermé et le principe recommence (l'algorithme choisit le point dans la Liste Ouverte dont le coût est le plus faible et ainsi de suite) [1]. Les figures 3 à 5 donnent un bon exemple de ce fonctionnement. Il est aussi important de mentionner que si. lors d'une exploration, l'algorithme tombe sur un point déjà exploré, celui-ci doit tout de même vérifier son coût. Si le coût qui lui avait été donné est supérieur que celui réalisé par ce nouveau chemin, alors sa valeur sera changée pour la plus faible (il sera aussi remis dans la Liste Ouverte si celui-ci était rendu dans la Liste Fermé) [1]. Après la figure 5, les étapes ont été recommencées à plusieurs reprises jusqu'à ce que l'on explore le point d'arrivée comme le montre la figure 6. Toutefois, l'algorithme n'arrête pas là puisqu'il peut y avoir un chemin plus court qui n'a pas été encore découvert. L'algorithme continue donc les mêmes étapes jusqu'à ce que la valeur du point d'arrivée soit la plus faible de la Liste Ouverte (comme le montre la figure 7). Ceci fonctionne uniquement s'il n'y a pas de coût négatif (comme dans l'exemple et dans le cas du bras robotique), dans le cas contraire toute la liste devra être vérifiée pour assurer de trouver le chemin le plus court [1]. Finalement, il reste seulement à refaire le chemin à l'envers comme le montre la figure 8.

	1	2	3	4	5
A	Départ ∞	∞	∞	∞	∞
B	∞	∞	∞	∞	∞
C	∞	∞	∞	∞	∞
D	∞	∞	∞	∞	∞
E	∞	∞	∞	Arrivé ∞	∞

Liste Ouverte	Liste Fermé

Figure 1 : Exemple de Dijkstra (a)

	1	2	3	4	5
A	Départ 0	∞	∞	∞	∞
B	∞	∞	∞	∞	∞
C	∞	∞	∞	∞	∞
D	∞	∞	∞	∞	∞
E	∞	∞	∞	Arrivé ∞	∞

Liste Ouverte	Liste Fermé
A1 = 0	

Figure 2 : Exemple de Dijkstra (b)

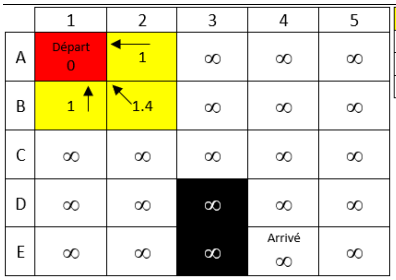


Figure 3 : Exemple de Dijkstra (c)

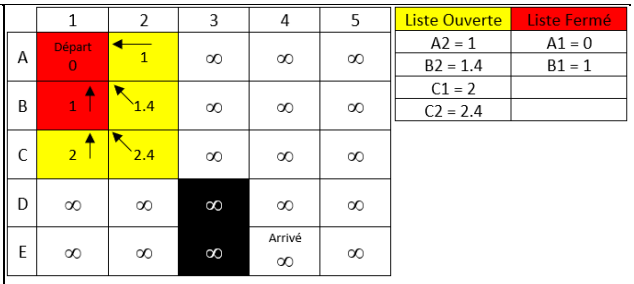


Figure 4 : Exemple de Dijkstra (d)

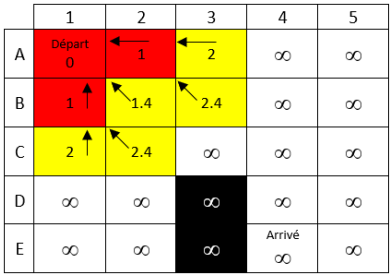


Figure 5 : Exemple de Dijkstra (e)

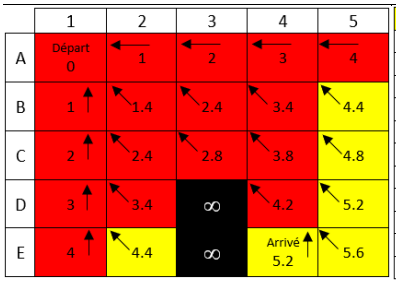


Figure 6 : Exemple de Dijkstra (f)

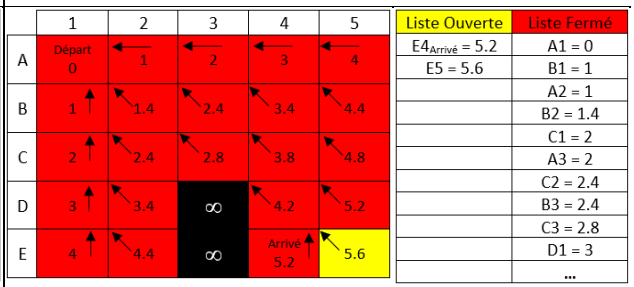


Figure 7 : Exemple de Dijkstra (g)

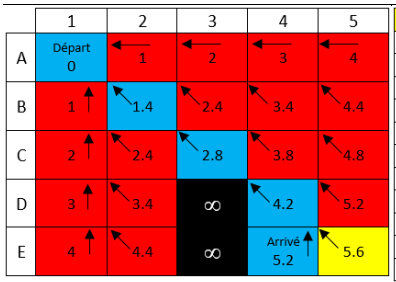


Figure 8 : Exemple de Dijkstra (h)

Cette méthode fonctionne très bien et pourrait efficacement fonctionner pour le bras robotique. Toutefois, le temps de calcul peut devenir très lent, dû au nombre de points explorés sans raison. Même si cet algorithme est encore utile dans l'industrie, il y a d'autres algorithmes qui permettent d'obtenir le même résultat, mais de façon plus rapide et donc de façon plus efficace.

ALGORITHME A*

L'algorithme A* est un algorithme qui est très similaire à l'algorithme de Dijkstra. Les deux fonctionnent exactement de la même façon, mais le coût des points est calculé de façon différente. Contrairement à l'algorithme de Dijkstra, le A* va évaluer le coût des points comme étant le coût parcouru pour s'y rentre plus une prédiction du coût minimal qu'il reste à faire pour arriver à la solution [1]. Cette prédiction est surnommée l'Heuristique. Dans la plupart des cas, puisque l'on utilise souvent le déplacement comme étant le coût parcouru, c'est souvent la distance euclidienne (à vol d'oiseau) entre un point et le point d'arrivée qui est utilisée comme heuristique de ce point. Le coût total d'un point sera donc le déplacement parcouru plus la distance euclidienne [1]. Cette méthode tente donc de prédire quel chemin emmènera plus rapidement à la solution. Pour le reste, celui-ci fonctionne exactement comme l'algorithme de Dijkstra, toutefois, cette différence de coût réduit énormément le nombre de points examiné. En utilisant le même exemple que pour Dijkstra, il est possible de constater avec les figures ci-dessous que le nombre de points examinés est nettement réduit.

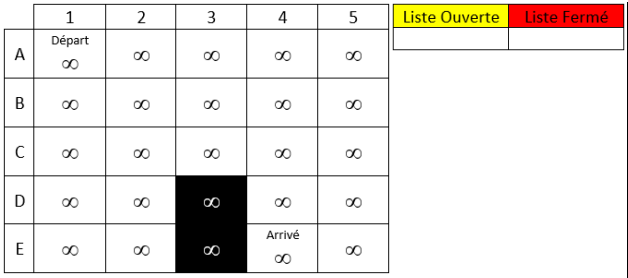


Figure 9 : Exemple A* (a)

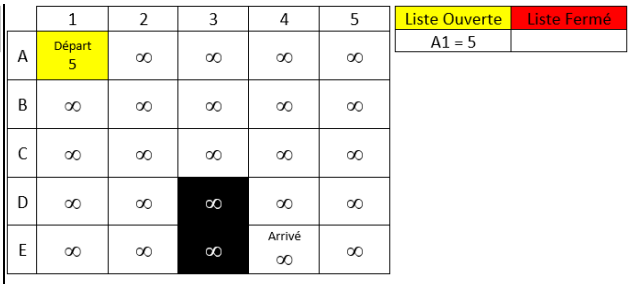


Figure 10 : Exemple A* (b)

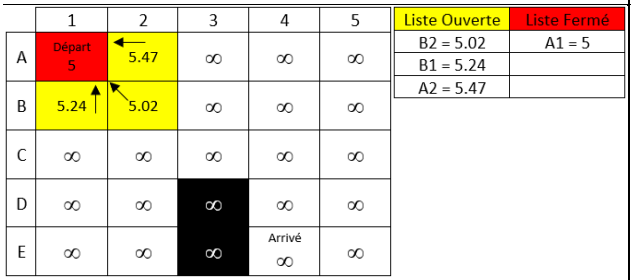


Figure 11 : Exemple A* (c)

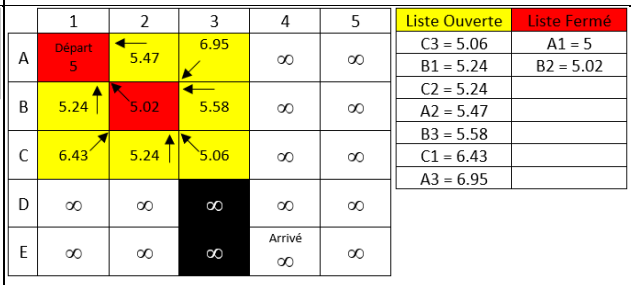
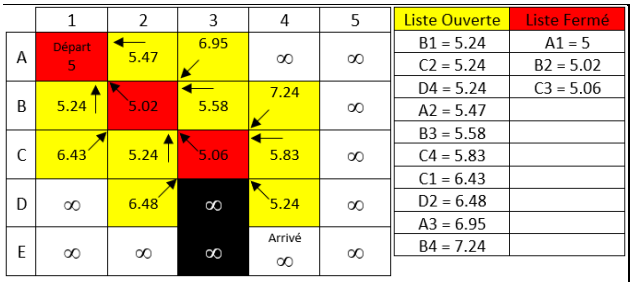


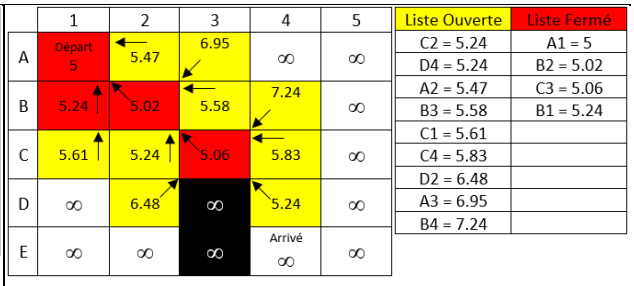
Figure 12 : Exemple A* (d)



	1	2	3	4	5
A	Départ 5	← 5.47	↙ 6.95	∞	∞
B	↑ 5.24	↖ 5.02	← 5.58	↙ 7.24	∞
C	↙ 6.43	↑ 5.24	↖ 5.06	← 5.83	∞
D	∞	↙ 6.48	∞	↖ 5.24	∞
E	∞	∞	∞	Arrivé ∞	∞

Liste Ouverte	Liste Fermé
B1 = 5.24	A1 = 5
C2 = 5.24	B2 = 5.02
D4 = 5.24	C3 = 5.06
A2 = 5.47	
B3 = 5.58	
C4 = 5.83	
C1 = 6.43	
D2 = 6.48	
A3 = 6.95	
B4 = 7.24	

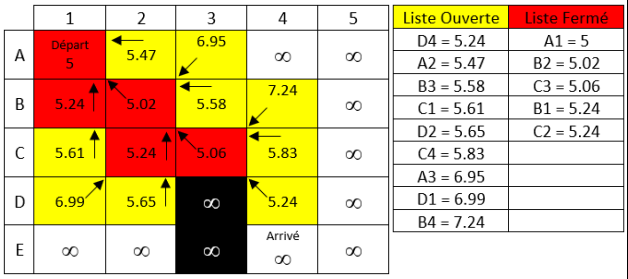
Figure 13 : Exemple A*(e)



	1	2	3	4	5
A	Départ 5	← 5.47	↙ 6.95	∞	∞
B	↑ 5.24	↖ 5.02	← 5.58	↙ 7.24	∞
C	↑ 5.61	↑ 5.24	↖ 5.06	← 5.83	∞
D	∞	↙ 6.48	∞	↖ 5.24	∞
E	∞	∞	∞	Arrivé ∞	∞

Liste Ouverte	Liste Fermé
C2 = 5.24	A1 = 5
D4 = 5.24	B2 = 5.02
A2 = 5.47	C3 = 5.06
B3 = 5.58	B1 = 5.24
C1 = 5.61	
C4 = 5.83	
D2 = 6.48	
A3 = 6.95	
B4 = 7.24	

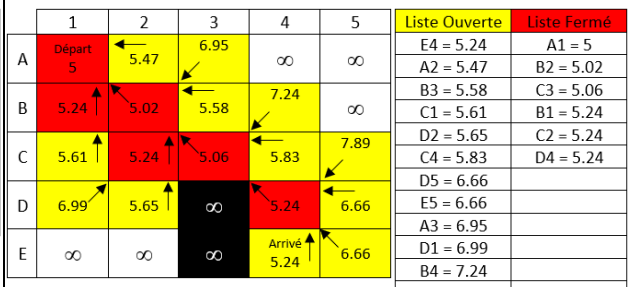
Figure 14 : Exemple A*(f)



	1	2	3	4	5
A	Départ 5	← 5.47	↙ 6.95	∞	∞
B	↑ 5.24	↖ 5.02	← 5.58	↙ 7.24	∞
C	↑ 5.61	↑ 5.24	↖ 5.06	← 5.83	∞
D	↙ 6.99	↑ 5.65	∞	↖ 5.24	∞
E	∞	∞	∞	Arrivé ∞	∞

Liste Ouverte	Liste Fermé
D4 = 5.24	A1 = 5
A2 = 5.47	B2 = 5.02
B3 = 5.58	C3 = 5.06
C1 = 5.61	B1 = 5.24
D2 = 5.65	C2 = 5.24
C4 = 5.83	
A3 = 6.95	
D1 = 6.99	
B4 = 7.24	

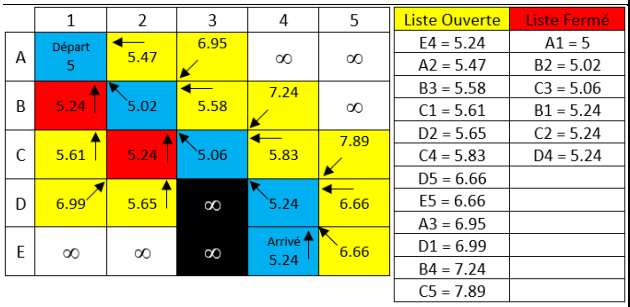
Figure 15 : Exemple A*(g)



	1	2	3	4	5
A	Départ 5	← 5.47	↙ 6.95	∞	∞
B	↑ 5.24	↖ 5.02	← 5.58	↙ 7.24	∞
C	↑ 5.61	↑ 5.24	↖ 5.06	← 5.83	↙ 7.89
D	↙ 6.99	↑ 5.65	∞	↖ 5.24	← 6.66
E	∞	∞	∞	Arrivé 5.24	↖ 6.66

Liste Ouverte	Liste Fermé
E4 = 5.24	A1 = 5
A2 = 5.47	B2 = 5.02
B3 = 5.58	C3 = 5.06
C1 = 5.61	B1 = 5.24
D2 = 5.65	C2 = 5.24
C4 = 5.83	D4 = 5.24
D5 = 6.66	
E5 = 6.66	
A3 = 6.95	
D1 = 6.99	
B4 = 7.24	
C5 = 7.89	

Figure 16 : Exemple A*(h)



	1	2	3	4	5
A	Départ 5	← 5.47	↙ 6.95	∞	∞
B	↑ 5.24	↖ 5.02	← 5.58	↙ 7.24	∞
C	↑ 5.61	↑ 5.24	↖ 5.06	← 5.83	↙ 7.89
D	↙ 6.99	↑ 5.65	∞	↖ 5.24	← 6.66
E	∞	∞	∞	↑ Arrivé 5.24	↖ 6.66

Liste Ouverte	Liste Fermé
E4 = 5.24	A1 = 5
A2 = 5.47	B2 = 5.02
B3 = 5.58	C3 = 5.06
C1 = 5.61	B1 = 5.24
D2 = 5.65	C2 = 5.24
C4 = 5.83	D4 = 5.24
D5 = 6.66	
E5 = 6.66	
A3 = 6.95	
D1 = 6.99	
B4 = 7.24	
C5 = 7.89	

Figure 17 : Exemple A* (i)

Pour le bras robotique, cet algorithme serait nettement plus efficace que l'algorithme de Dijkstra. Celui-ci semble donc être le plus efficace pour le projet jusqu'à présent.

ALGORITHME D*

L'algorithme D* est un autre algorithme fortement utilisé dans l'industrie et celui-ci utilise un principe très similaire à l'algorithme A*. Cet algorithme a été conçu pour fonctionner et réduire le temps de calcul dans un environnement très dynamique. Lorsqu'un obstacle se déplace et entre dans la trajectoire de l'objet en mouvement, celui-ci doit s'arrêter et recommencer ses calculs pour trouver le nouveau chemin qu'il doit prendre. Pour y arriver, il est possible de démarrer à nouveau

l'algorithme A*, mais si ceci se reproduit souvent, le temps de calcul peut devenir très grand. L'algorithme D*, quant à lui, réutilise les données précédentes pour trouver le nouveau chemin. Celui-ci est donc nettement plus rapide dans un environnement dynamique. Cependant, cet algorithme peut uniquement être utilisé lorsqu'il y a un déplacement d'obstacle [1]. Il est donc nécessaire d'utiliser un autre algorithme pour calculer le premier chemin dans l'environnement statique. Malheureusement, pour que le D* fonctionne bien, il est nécessaire d'utiliser l'algorithme de Dijkstra pour l'environnement statique [1]. Puisque la méthode de Dijkstra est nettement plus lente que le A* (dans le cas du bras robotique), le D* pénalise donc grandement les calculs lorsque l'environnement est statique pour permettre un grand avantage de calcul lorsque l'environnement deviendra dynamique. Or, dans le cas du bras robotique, l'utilisation sera, pour la plupart des cas, dans un environnement peu dynamique ou même complètement statique puisque celui-ci sera utilisé pour accomplir des tâches simples. Il n'est donc pas intéressant pour le bras d'utiliser un algorithme qui pénalise les calculs dans l'environnement statique. Il sera donc préférable de simplement recalculer le nouveau chemin au complet dans les quelques cas où il y aura un déplacement d'obstacle. En d'autres mots, il est préférable de rester avec l'algorithme A* que d'utiliser l'algorithme D*.

ALGORITHME HD*

Cet algorithme est un autre algorithme fortement utilisé de nos jours. Toutefois, celui-ci ne peut pas, encore une fois, s'appliquer au bras robotique et ce pour deux raisons. Premièrement, celui-ci utilise une méthode de D* [2]. Or, il a été conclu précédemment que l'algorithme D* est nettement moins avantageux que l'algorithme A* pour le bras robotique. Deuxièmement, cet algorithme est utilisé lorsqu'il est impossible pour un système mécatronique de connaitre son environnement en avance (ce qui n'est pas le cas pour le projet). Cet algorithme a été conçu pour fonctionner avec un plan non complet qui se construit au fur et à mesure que l'objet se déplace et découvre son environnement [2]. Toutefois, ceci fonctionne nettement moins rapidement que si l'environnement est initialement connu et ne permet pas de trouver nécessairement le chemin le plus rapide.

ALGORITHME QUADTREE / OCTREE

Les algorithmes Quadtree et Octree ne sont pas des algorithmes de planification de trajectoire, mais ceux-ci peuvent tout de même être intéressants pour le projet. Les deux utilisent le même principe, mais le premier est pour le 2D et l'autre pour le 3D. Ils peuvent être conçus pour représenter, entre autres, un plan d'obstacles de façon hiérarchique, sauvant ainsi de l'espace mémoire. Une structure hiérarchique de données est basée sur une méthode de décomposition. Au lieu de conserver l'information, par exemple dans une matrice dont toutes les zones sont de même dimension (ce qui forme un plan comme illustré à la figure 18), ceux-ci permettront de conserver l'information dans un arbre hiérarchique permettant d'avoir des zones de dimensions différentes [3]. L'algorithme Quadtree fonctionne tout d'abord en analysant la zone complète du plan pour vérifier si celui-ci est formé uniquement de vide ou uniquement d'obstacles. Si c'est bien le cas, l'algorithme termine et le plan est maintenant formé d'une seule zone de vide ou d'obstacles (dépendamment de ce qui a été déterminé). Toutefois, si la zone possède les deux, alors celle-ci est séparée, à l'aide d'une croix, en 4 rectangles de dimensions égales. Ces 4 nouvelles zones sont donc ensuite analysées de la même manière, pouvant ainsi être séparées à nouveau. Ce principe est reproduit jusqu'à ce que toutes les zones soient formées uniquement de vide ou d'obstacles (à moins qu'une dimension minimum ai été précisée par le programmeur, faisant arrêter la séparation de certaines zones lorsque cette limite est atteinte) [3]. La figure 19 illustre bien à quoi pourrait ressembler un plan formé par un algorithme Quadtree tandis que la figure 20 montre à quoi ressemblerait l'arbre hiérarchique qui conserve les données de ce plan. Pour ce qui est de l'algorithme Octree, celui-ci fonctionne de la même manière, mais sépare, à l'aide d'une croix à trois branches, les zones en 8 prismes à base rectangulaire de dimensions égales.

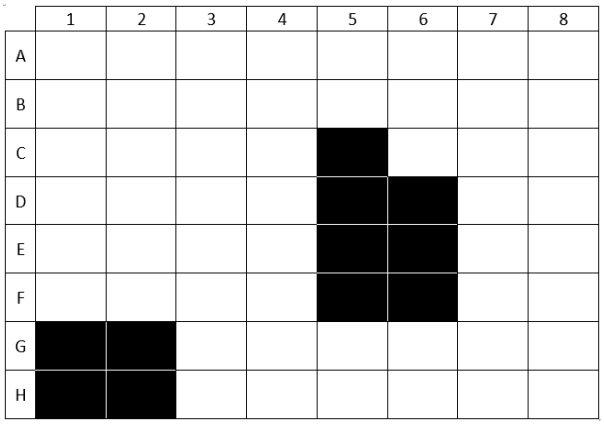


Figure 18 : Plan normal

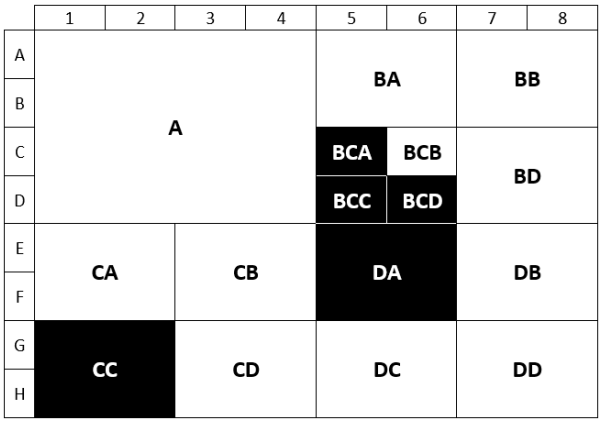


Figure 19 : Plan de l'algorithme Quadtree

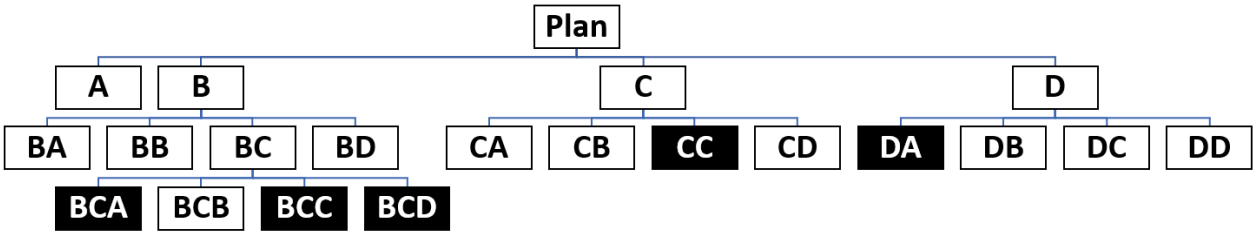


Figure 20 : Arbre hiérarchique de la figure 19

Il serait donc grandement intéressant d'utiliser un tel algorithme pour ainsi sauver de l'espace mémoire du système. De plus, ce genre d'algorithme pourrait aussi permettre de réduire le temps de calcul puisque les zones à vérifier seront plus grandes, réduisant ainsi leur nombre. Toutefois, ceci pourrait aussi apporter certains problèmes. Peu importe si le plan est normal ou décomposé, on utilise habituellement le centre des zones comme coordonnées pour le mouvement de l'objet. Or, avec l'algorithme Quadtree, les dimensions des zones sont maintenant plus grandes, ce qui réduit le nombre de possibilités de déplacement et ce qui rallonge aussi le chemin de l'objet pour se rendre à l'objectif (puisque les zones sont moins précises). Ce genre d'algorithme peut aussi causer d'autres inconvénients, dépendamment de l'algorithme de planification de trajectoire choisi. Cet algorithme devra donc être modifié et adapté au projet et à l'algorithme de planification de trajectoire choisi si jamais la décision est de tenter de l'ajouter au système.

PROBLÉMATIQUE DU A*

L'algorithme A* semble donc être la meilleure solution pour le projet. Toutefois, celui-ci possède un grand désavantage. Malgré le fait que son temps de calcul soit très court, lorsqu'il est question d'explorer le voisinage d'une zone cet algorithme ne vérifie que celles qui sont en contact avec celle-ci. Comme le montre la figure 21, ceci limite grandement les possibilités de déplacement.

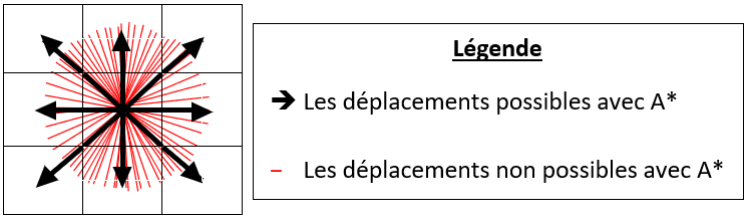


Figure 21 : Déplacement possible et non possible avec l'algorithme A*

Cette limitation cause deux gros problèmes. Le premier est que le chemin peut devenir très saccadé puisque celui-ci peut facilement zigzaguer vers la solution au lieu de s'y rendre avec une ligne droite. De plus, ceci s'aggrave grandement lorsque le plan est en 3D au lieu d'être en 2D et continue à s'aggraver plus la dimension grandit. Ceci réduira donc grandement la fluidité lors du déplacement du bras. Deuxièmement, ceci ne permet pas de trouver nécessairement le chemin le plus court, comme le montre l'exemple de la figure 22. Toutefois, l'algorithme de Dijkstra, D* et HD* possèdent ce même problème. Heureusement, il existe d'autres variantes de l'algorithme A* qui permettent d'aider à contrer ces problèmes. Toutefois, ceux-ci possèdent d'autres désavantages.

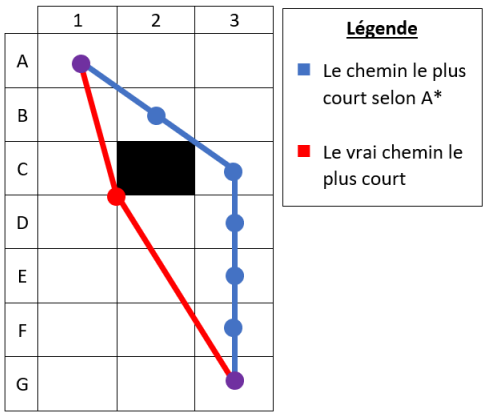


Figure 22 : Comparaison entre le chemin optimal et celui du A*

ALGORITHME A* AVEC DES CHEMINS « POST-SMOOTHED » (A* PS)

L'algorithme A* PS utilise l'algorithme A* et analyse ensuite le parcours déterminé dans le but de le lisser. Supposons que l'algorithme A* trouve le chemin [p_0, p_1, p_2, … , p_n] où « p_0 » est le point de départ et le « p_n » est le point d'arrivée. L'algorithme A* PS va ensuite prendre le premier point de la liste (soit « p_0 ») et vérifier si celui-ci peut atteindre le successeur (soit « p_2 ») de son successeur (soit « p_1 ») en réalisant une ligne droite. Si c'est possible, alors son successeur (« p_1 ») est enlevé de la liste et l'algorithme vérifie si le premier point est capable d'atteindre le prochain successeur (pour tenter d'enlever aussi « p_2 » et ainsi de suite). Toutefois, dès que ce n'est pas possible, l'algorithme passe au prochain point de la liste et vérifie le successeur du successeur de ce nouveau point, recommençant ainsi le même principe [4]. Comme on peut le voir à la figure 23, cette méthode permet non seulement de lisser le chemin, mais aussi de le rendre plus court. Toutefois, même si cette méthode améliore le résultat obtenu à l'aide du A*, elle n'est pas conçue pour trouver le chemin optimal [4]. De plus, ajouter des étapes de calculs augmente le temps nécessaire pour les réaliser. Cette méthode prend donc plus de temps à réaliser que le A*, mais permet un meilleur déplacement du bras robotique. Ceci semble donc être le prix à payer pour le projet. Cependant, d'autres méthodes seront montrées plus loin permettant d'optimiser encore plus la solution sans prendre autant de temps que l'algorithme A* PS.

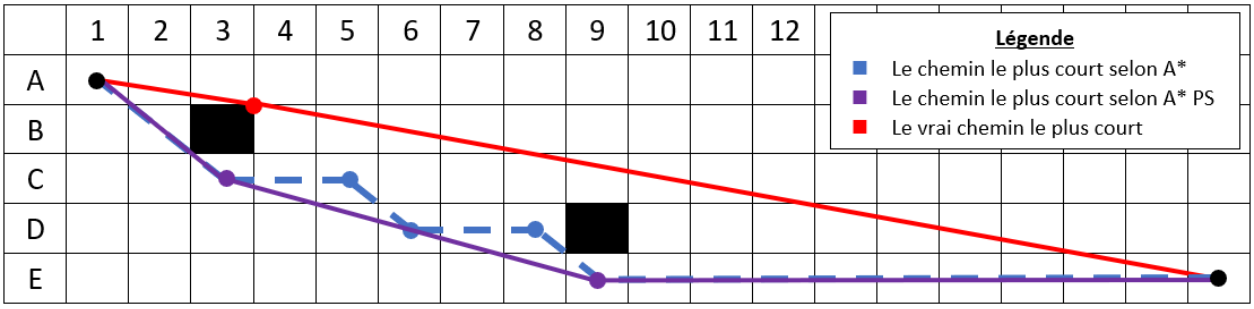


Figure 23 : Comparaison en A*, A*PS et le chemin optimal

ALGORITHME FIELD D* (FD*)

Cet algorithme utilise un principe similaire à l'algorithme Lite D* (une variante de l'algorithme D*) pour permettre de lisser le résultat obtenu par l'algorithme A*. Tout comme l'algorithme A* PS, celui-ci est donc utilisé après que le chemin ai été obtenu par l'algorithme A*. Toutefois, cette méthode est très complexe et nécessite plus de temps de calcul que l'algorithme A* PS [4]. Elle permet tout de même d'obtenir, dans la plupart des cas, un chemin un peu plus court que l'algorithme A* PS [4]. Cependant, il existe d'autres algorithmes plus rapides que A* PS qui permettent d'obtenir un chemin encore plus optimisé que l'algorithme Field D*.

ALGORITHME BASIC THETA*

L'algorithme Basic Theta* est un algorithme qui utilise le même principe que A* PS mais en appliquant la méthode tout en utilisant l'algorithme A* au lieu de le faire après. Lorsque l'algorithme roule A* et met à jour la valeur du coup d'un nouveau point, celui-ci considère un deuxième chemin possible au lieu de considérer simplement le chemin trouvé par l'algorithme A*. Ce deuxième chemin est celui qui relit d'une ligne droite ce nouveau point avec le parent de son parent. Si ce deuxième chemin est possible, alors c'est celui-ci qui sera considéré au lieu du chemin trouvé par l'algorithme A* [4]. La figure 24 montre deux exemples où ce deuxième choix est possible et impossible. Ce genre d'algorithme est très simple à implémenter dans un A* standard et, même si celui-ci augmente le nombre d'étapes de calcul, il permet à la solution de converger plus rapidement vers l'objectif. Ceci le rend donc énormément plus rapide que le A* PS mais tout de même un peu plus lent que le A* normal. De plus, sa marge d'erreur selon le chemin le plus court est beaucoup plus faible. Il est donc un algorithme fort intéressant pour le projet. Il existe aussi un autre algorithme similaire nommé le « Angle-Propagation » Theta* (AP Theta*). Toutefois, ce dernier nécessite plus de temps de calcul que le Basic Theta* sans permettre de trouver un chemin plus optimisé que celui-ci. Cependant, il permet de visiter moins de points, permettant de réduire un peu l'espace mémoire [4].

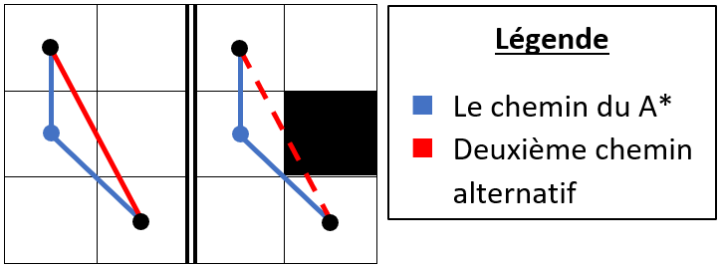


Figure 24 : Exemple de l'algorithme Basic Theta*

ALGORITHME A* SUR « VISIBILITY GRAPHS » (A* VG)

Cet algorithme fonctionne avec le principe que le chemin le plus rapide va toujours passer par les coins des obstacles sauf si le point d'arrivée est dans le champ de vision du point de départ. En 3D, c'est par les coins et les arrêtes des obstacles que le chemin le plus rapide passera. De plus, il fonctionne exactement comme le A* normal, mais au lieu d'être sur une grille, celui-ci fonctionne sur un « Visibility Graphs » (VG) [4]. Habituellement, pour ce genre d'algorithme, le VG va relier ensemble tous les points importants d'un plan qui seront dans le champ de vision de l'autre. Ces points importants sont constitués du point de départ, du point d'arrivée et des coins des obstacles. La figure 25 montre un exemple de « Visibility Graphs ». Ceci permet donc à l'algorithme A* de créer un chemin en passant directement par les coins des obstacles au lieu de devoir examiner et passer par tous les points voisins. De plus, cette méthode permet de trouver réellement (sans exception) le chemin le plus court possible [4]. Toutefois, malgré que cette méthode réduise considérablement le temps de calcul de l'algorithme A*, le temps de calcul pour produire le VG est extrêmement long et rend donc cette méthode très lente comparativement à toutes les autres méthodes vues dans ce rapport [4]. Il existe toutefois un seul moyen pour réduire de beaucoup le temps de calcul, sauf si le plan possède énormément de longs murs, réduisant ainsi de beaucoup le champ de vision tout en augmentant la quantité d'obstacles (ce qui n'est pas le cas dans le cadre du projet). Cette méthode consiste à calculer quels sont les points importants dans le champ de vision d'un point au fur et à mesure que le A* trouve le chemin [4]. Cependant, malgré que cette méthode réduise énormément le temps de calcul, ceci rend tout de même cet algorithme plus lent que tous les autres expliqués dans le rapport [4]. Cet algorithme est donc très peu utilisé.

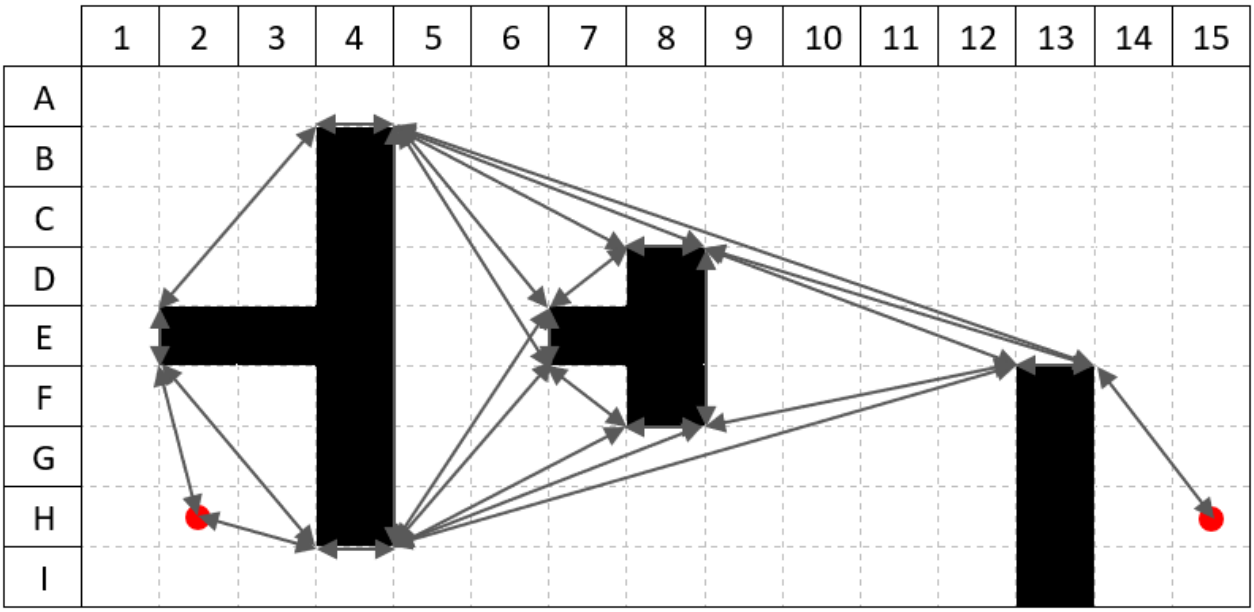


Figure 25 : Exemple de « Visibility Graphs »

CONCLUSION ET SUGGESTIONS

Pour conclure, il serait donc fort intéressant d'utiliser l'algorithme Basic Theta* pour la planification de la trajectoire. Dans le cadre du projet, celui-ci est le meilleur algorithme faisant un compromis entre la fluidité du déplacement du bras, la rapidité de calcul et la capacité de déterminer le chemin le plus court. De plus, peu importe l'algorithme utilisé, il serait intéressant d'essayer d'implémenter un algorithme Octree pour tenter de réduire l'espace mémoire et le temps de calcul général. Pour finir, il pourrait aussi s'avérer intéressant de programmer le tout dans un espace de travail en 5 dimensions. La raison est que le bras possède, dans le projet, 5 moteurs permettant à celui-ci de se déplacer dans l'environnement 3D. Les 5 variables pourront donc représenter les angles de ces 5 moteurs. Les deux autres moteurs du bras (puisqu'il y en a 7 en tout) servent uniquement à tourner la main sur le poignet et à ouvrir et fermer les doigts. Ceux-ci ne sont donc pas utiles pour le déplacement, mais uniquement pour agripper l'objet final. En étant en 5D, le temps de calcul serait grandement réduit et le déplacement du bras serait facilité. De plus, chaque algorithme suggéré peut fonctionner en 5 dimensions. Cependant, ceci augmentera probablement la quantité de données dans l'espace mémoire.

RÉFÉRENCES

[1] H. Choset, A. Mills-Tettey, K. Tantisevi et V. Lee-Shue Jr. Prasad Narendra Atkar, «Robotic Motion Planning: A* and D* Search,» Carnegie Mellon University, Pittsburgh, 2005.

[2] M. David Solomon, «Development of a Real-Time Hierarchical 3D Path Planning Algorithm for Unmanned Aerial Vehicles,» University of Maryland, College Park, 2016.

[3] H. Samet , «AN OVERVIEW OF QUADTREES, OCTREES, AND RELATED HIERARCHICAL DATA STRUCTURES,» University of Maryland, College Park, 1988.

[4] K. Daniel, A. Nash et S. Koenig, «Theta*: Any-Angle Path Planning on Grids,» University of Southern California, Los Angeles, 2010.