Planification de mouvement pour des robots mobiles en ligue SSL

Emmanuelle Challier Maëlle Dubucq Florine Lefer

6 mai 2021

1 Introduction

Ce projet se place dans le cadre de la Robocup, plus précisément de la ligue SSL (Small Size League) où deux équipes de robots mobiles à roues s'affrontent dans un match de football. Nous étudierons dans ce projet la planification de mouvement pour des robots mobiles, et son implémentation dans le cadre spécifique des robots SSL. Problématique : Quelles approches s'offrent à nous pour gérer la planification de mouvement ? Comment ces approches se traduisent-elles sur les robots SSL ?

2 Robocup et Ligue SSL

2.1 Présentation

La RoboCup Research Federation est une initiative scientifique internationale dont le but est de faire progresser la recherche dans le domaine de la robotique intelligente, dans le cadre d'une compétition de football. Elle a été fondée en 1997 par trois professeurs (Hiroaki Kitano, Manuela M. Veloso, et Minoru Asada), avec l'objectif initial d'arriver avant 2050 à créer une équipe de football de robots humanoïdes, complètement autonomes, capables de gagner un match contre l'équipe championne du monde la plus récente.

La RoboCup Small Size League fait partie de la division "soccer", l'une des 4 divisions de la RoboCup ("soccer, "rescue", "industrial" et "@home") qui est divisée en plusieurs ligues. Elle se focalise sur le contrôle et la coordination multi-agent. Les équipes de cette ligue sont composées de 11 petits robots roulants (non humanoïdes) (dimensions maximum : diamètre 18cm, hauteur 15cm), qui jouent dans un terrain de 12m de long et 9m de large avec une balle de golf orange. Un système de vision central composé d'un serveur de vision central et d'un ensemble de caméras fournit aux équipes des données de localisation via un réseau filaire (Ethernet). Chaque équipe met en œuvre des algorithmes pour prendre des décisions, et envoie les commandes correspondantes aux robots à l'aide d'une communication sans fil.

Des compétitions en ligue SSL s'organisent plusieurs fois par an (deux fois depuis plusieurs années) dans différents pays. Une compétition devait se tenir en 2020 à Bordeaux mais est finalement reportée à 2023 à cause de la situation sanitaire.

Il existe des dizaines d'équipes du monde entier dans cette ligue, dont deux équipes françaises qui sont toutes deux bordelaises. L'équipe NAMEC a été créée dans la dynamique de la candidature française à l'organisation de la RoboCup à Bordeaux en 2020, à l'initiative d'un membre de l'équipe Rhoban - aussi bordelaise - quadruple championne du monde dans d'autres ligues. Une deuxième équipe, NAELIC, composée d'étudiants de Bordeaux avec d'Etienne Schmitz étudiant à l'Einserb comme Team Leader, a été fondée il y a un an et participera pour la première fois à la Robocup 2022 à Bangkok.

2.2 Planification

Dans le cadre de la SSL la planification de trajectoire est utilisée pour trouver le chemin le plus court et/ou plus rapide sans collision entre 2 points dans un environnement (dynamique ou statique). On retrouve souvent une distinction entre la planification de trajectoire et la planification de mouvement qui elle contrairement à la première ce souci des contraintes dynamiques (limite des joints, limite de vitesse/vélocité...). Il existe 2 approches possibles, la première consiste à diviser la planification de mouvement en une phase de planification de trajectoire suivi d'une phase d'exécution qui est utilisé pour suivre le chemin trouver en tirant partie des méthodes de la théorie du contrôle.

La deuxième approche consiste à directement trouver la trajectoire du robot, en adaptant les algorithmes pour prendre les contraintes et ainsi trouver le mouvement complet du robot.

L'algorithme $RRT*$ (une variante de RRT) est vastement utilisé par les équipes de la SSL car il est problement complet et asymptotiquement optimale (voir partie 4.3).

3 Planification de mouvement

La planification de mouvement consiste à calculer le chemin menant d'un point de départ à un point d'arrivée, généralement appelé but. Notons dès maintenant qu'il existe également des déplacements non planifiés, qu'on appelle aussi réactifs, qui se distinguent par le l'absence de but. C'est par exemple le déplacement d'un humain virtuel, aussi appelé avatar dans le langage des jeux vidéo, qui va se retrouver confronté à la présence d'un gros chien et va, par réflexe, faire quelques pas en arrière. Dans cette situation il n'y a pas de but à atteindre mais seulement un mouvement réflexe, d'évitement.

3.1 Représentation de l'espace

Avant de définir l'algorithme de calcul du chemin, un point important est la représentation de l'espace dans lequel il va être défini. Une représentation continue n'est pas envisageable pour des raisons de complexité de calcul car elle impliquerait la manipulation de polynômes de degré gigantesque pour des environnement complexes. Une représentation discrète de l'espace est donc définie. On peut classifier les représentations en deux grandes catégories : les partitions spatiales et les graphes.

Une partition spatiale va consister à découper l'espace praticable en éléments de surface simples sur lesquels les calculs vont être également simples. Les deux représentations les plus courantes sont la grille et la triangulation.

Une grille est constituée de carreaux de même taille alignés sur les axes, ce qui permet un adressage direct de chaque carreau : on a une bijection simple permettant de faire correspondre les indices et les coordonnées d'un carreau. L'avantage de cette représentation est la régularité de la structure qui va par exemple permettre de déterminer facilement, connaissant une position et un vecteur déplacement, dans quel carreau se trouve la position d'arrivée.

Une triangulation est une partition de l'espace en une collection de triangles. Même si on pourrait contraindre la triangulation afin qu'elle respecte la régularité d'une grille (sommets alignés sur les axes), ce n'est en général pas le cas car l'intérêt de la triangulation est justement sa souplesse, qui va permettre de l'adapter à l'environnement et notamment à la présence d'obstacles, la triangulation collant au bord des obstacles, quelque soit leur orientation. Un autre avantage est que cette représentation a été largement étudiée et utilisée en synthèse d'images et qu'il existe de très nombreuses solutions algorithmiques pour manipuler des triangulations (mesh en anglais). Les modèles de terrain sont notamment représentés de cette façon et introduisent une coordonnée Z donnant l'altitude (on parle de 2,5D et non de 3D car il existe une bijection entre la coordonnée Z et les coordonnées X et Y). C'est également la représentation utilisée par les cartes graphiques, ce qui permet d'effectuer certains traitements sur les GPU, environ 1 million de fois plus rapides que les CPU.

On peut également représenter l'espace praticable à l'aide d'un graphe, qui va être plaqué sur l'espace de départ et porter les informations au niveau de ses noeuds. L'avantage de cette représentation est que la théorie des graphes fournit de nombreux algorithmes permettant de trouver des solutions à la plupart des problèmes.

Le choix de la représentation de l'espace se fera en fonction des contraintes de l'application mais nous pouvons donner quelques éléments de réflexion. La différence la plus importante tient à la nature de la représentation et aux conséquences quant à la définition d'un chemin au sein de cette représentation. Si dans le cas d'une partition spatiale n'importe quel point de l'espace peut faire partie d'un chemin, avec un graphe celui-ci doit suivre les arêtes du graphes. Un chemin sera donc dans le premier cas une suite ordonnées de points de l'espace, définis par leur coordonnées (x,y) et dans le second cas une suite ordonnées de sommets, chaque paire de sommets consécutifs du chemin devant être connectés

par une arête du graphe. On travaille donc dans un espace continu dans le cas d'une partition spatiale mais dans un espace discret dans le cas d'un graphe. La conséquence est qu'il sera plus facile de déterminer des trajectoires lisses, "naturelles", dans l'espace continu que dans l'espace discret. Un autre aspect à prendre en considération est l'adaptabilité de la structure. Nous verrons au paragraphe 3.3 qu'optimiser le calcul nécessitera de pouvoir faire varier la distance entre deux points consécutifs du chemin calculé, ce qui ne posera pas de problème si on travaille en espace continu mais sera au contraire impossible dans le cas discret, sauf à redéfinir tout ou partie du graphe, ce qui semble un problème compliqué et surtout coûteux en temps de calcul.

3.2 Gestion des obstacles

Les algorithmes de planification de mouvement créent des chemins sur lesquels un mobile se déplace, pour aller d'un point à un autre. Ces chemins doivent être libres, autrement dit le déplacement du mobile ne doit engendrer de collision avec un obstacle, qu'il soit fixe (bâtiments par exemple) ou mobile. En pratique, dans la majorité des algorithmes de planification de mouvement, la majeure partie du temps de calcul est consacrée à la gestion de collision. C'est pourquoi cette partie requiert une attention particulière, surtout si on se place dans un cadre temps réel, comme par exemple dans les jeux vidéo. On trouve sur le marché des moteurs physiques, qui sont des logiciels optimisés pour gérer les collisions et qui utilisent le plus souvent la puissance des GPU (Graphic Processing Unit) afin de proposer un fonctionnement en temps réel avec gestion critique du timing.

Pour ce qui est des obstacles fixes, une solution consiste à mailler l'espace praticable autour des obstacles, ce qui est plus simple avec une triangulation (des algorithmes connus existent) ou un graphe qu'avec une grille (alignement des bords compliqué avec un bord non parallèle aux axes). Lorsque l'obstacle est mobile il n'est pas possible de recalculer la représentation de l'espace à chaque pas de temps, pour des raisons de coût de calcul mais aussi parce que cela remettrait en cause l'état du planificateur (sommets du graphes seraient modifiés et certains objets ne seraient plus sur un noeud du graphe). Il faut alors implémenter un algorithme de détection de collision entre deux objets. C'est cette partie qui est assurée par un moteur physique.

3.2.1 Intersection entre deux objets

Tester si deux objets géométriques entrent en collision revient à tester s'il existe un point d'un des objets qui est à l'intérieur de l'autre. Si les objets sont représentés de façon implicite, le test peut être trivial (cas d'un cercle en 2D ou d'une sphère en 3D) ou très compliqué si l'équation définissant le bord de l'objet est de degré élevé. Si l'objet est défini par son bord, suite ordonnée de sommets en 2D ou collection de triangles en 3D, et sous l'hypothèse qu'il soit convexe, l'algorithme va consister à tester chaque sommet de chaque objet pour déterminer s'il est à l'intérieur de l'autre objet. Remarquons que ce test n'est

pas suffisant en 3D et que deux cubes peuvent s'interpénétrer bien que le test décrit ici ne permettra pas de le détecter mais ce ce genre de configuration est suffisamment rare pour pouvoir être négligé. Il existe des techniques permettant de résoudre le problème pour des objets concaves mais ils sont beaucoup plus coûteux et on préférera dans ce cas décomposer l'objet en une collection de parties convexes et traiter chaque partie indépendamment.

Une technique simple pour déterminer si un point est à l'intérieur d'un polygone convexe en 2D est la technique de la demi-droite : on trace une demi droite, horizontale ou verticale, afin de simplifier le calcul, qui va du point à l'infini et on teste l'intersection de cette demi-droite avec chaque section du bord du polygone. Si le nombre d'intersection est impair le point est à l'intérieur sinon il est à l'extérieur. Remarquons que cette technique fonctionne également pour des objets concaves. Si l'objet est convexe une méthode encore plus efficace consiste à injecter les coordonnées du point dans l'équation de la droite supportant l'arête du polygone et à tester le signe du résultat. En 3D, de façon similaire, on va tester que le point est du "bon" côté de chaque plan contenant une face du bord de l'objet, en injectant les coordonnées du point dans l'équation du plan. Si tous les tests par rapport aux droites (cas 2D) ou aux plans (cas 3D) renvoient un résultat de même signe alors le point est à l'intérieur de l'objet, sinon il est à l'extérieur.

Précisons enfin comment nous allons déterminer l'équation cartésienne d'une droite ou d'un plan supportant une arête ou une face d'un objet polygonal, en 2D ou en 3D respectivement. Pour simplifier la description, nous allons la faire en 2D mais elle se généralise facilement au cas 3D. Le bord d'un objet polygonal est décrit par la suite ordonnée des sommets qui le compose. Soient $P0(x_0, y_0)$ et $P1(x_1, y_1)$ les extrémités d'une arête en 2D. L'équation cartésienne de la droite supportant l'arête est alors : $(x_1 - x_0)x - (y_1 - y_0)y + c = 0$. La constante c peut être déterminée en injectant les coordonnées de $P0$ dans l'équation, ce qui donne : $c = (y_1 - y_0)y_0 - (x_1 - x_0)x_0$. Si maintenant on injecte les coordonnées d'un point quelconque $P(x, y)$ dans cette équation, on obtiendra 0 si le point est sur la droite, une valeur positive s'il se trouve à droite de la droite et une valeur négative s'il se trouve à gauche de la droite. Ce que nous appelons la droite de la droite est quand on regarde cette droite orientée selon l'ordre des sommets. Si un triangle est décrit par la suite de ses sommets $(P0, P1, P2)$ alors les droites seront orientées de $P0$ à $P1$, de $P1$ à $P2$ et de $P2$ à $P0$ respectivement. Ainsi avec $P0(0, 0)$ et $P1(0, 1)$, le point $P(1, 1)$ sera à droite de la droite et le point $P'(-1, 1)$ sera à gauche. Remarquons que la convention en modélisation 3D impose d'ordonner les sommets de sorte que la normale au triangle soit dirigée vers l'extérieur de l'objet. Ainsi le triangle de sommets $P0(0, 0)$, $P1(1, 1)$, $P2(0, 1)$ sera décrit par le triplet $(P0, P1, P2)$ et non par $(P0, P2, P1)$. Cette convention est importante car elle permet de conclure que les points sont du "bon" côté du bord de l'objet si le signe du test décrit plus haut est négatif. On pourra donc arrêter les calculs dès qu'on aura trouvé un résultat positif ou nul.

3.2.2 Complexité

Intéressons nous maintenant à la complexité du test global de détection de collision qui doit être effectué à chaque calcul d'un nouveau point du chemin. Pour tester l'intersection entre deux triangles en 2D, cas simplissime, nous avons besoin au pire de 6 tests tels que décrits à la section précédente. La complexité est donc en $O(n^2)$, n étant le nombre de sommets du bord de chaque objet. En 3D le nombre de sommets pourra être très grand pour des formes complexes, de l'ordre de plusieurs centaines. Et ce test devra être fait pour toutes les paires d'objets contenus dans l'espace, qui peut être de plusieurs centaines dans des environnements complexes. La complexité du test de collision est donc de $O(N^2 \times n^2)$ où N est le nombre d'objets que comporte l'espace et n le nombre moyen de sommets que comporte le bord de chaque objet. Nous avons donc affaire à une complexité quadratique en le nombre d'objets et le nombre de sommets de chaque objet.

3.2.3 Techniques d'accélération

Il y a deux approches afin de réduire le coût de cette étape de recherche de collisions : diminuer le coût du test de base et diminuer le nombre de tests qui sont effectués. Nous décrivons maintenant deux méthodes qui utilisent ces approches.

Volumes englobants Afin de réduire le coût du test d'intersection entre deux objets, on va construire une forme géométrique qui contient entièrement l'objet mais a comme bonne propriété de permettre de tester l'intersection avec un coût faible du test. Un exemple évident est un cercle en 2D ou une sphère en 3D. Tester l'intersection entre deux cercles ou deux sphères est trivial. Le principe consiste alors à d'abord tester l'intersection des deux volumes englobants et seulement si cette intersection existe, affiner le calcul en faisant le test décrit plus haut. Le succès de cette approche repose sur le fait que pour un nombre important d'objets le nombre d'intersections entre deux objets est très faible. Soit p le pourcentage de tests d'intersection positifs des volumes englobants lorsqu'on teste toutes les paires d'objets de l'espace, c le coût du test d'intersection de deux volumes englobants et C le coût du test d'intersection des deux objets eux-mêmes. La condition pour que cette technique d'accélération soit rentable est : $c + p \times C < C$ ou encore : $c < (1 - p)C$. Si le test des volumes englobants permet d'éliminer 90% des cas (p=0,1), il suffit que son coût soit inférieur à 0,9 fois le coût du test décrit au paragraphe 3.2.1. On voit que ce score est très facile à atteindre. On peut diminuer p en choisissant une géométrie de volume englobant qui épouse mieux la forme de l'objet qu'un cercle ou une sphère. Par exemple des paires de droites ou de plans délimitants ont cette propriété, même si le test va être un peu plus coûteux mais ce sera rentable au regard du gain en complexité.

On peut également réduire la complexité afin de diminuer le nombre de tests effectués en hiérarchisant la structure de volumes englobants. On va ainsi définir

des super volumes englobants qui vont contenir plusieurs volumes englobants et ceci de façon récursive afin d'obtenir une hiérarchie de volumes englobants. Le test de collision va alors consister à parcourir cet arbre en largeur pour tous les super volumes d'un niveau donné mais en profondeur seulement lorsque le super volume aura détecté une collision.

Méthode incrémentale Plutôt que d'introduire un nouveau test et une nouvelle structure de données, cette méthode, appelée calcul de distance incrémentale, propose de réécrire le calcul de la distance entre deux objets afin de le rendre plus efficace. Elle consiste à évaluer la distance minimale entre deux objets puis à mettre à jour cette information alors que les objets sont en mouvement. Sous l'hypothèse qu'entre deux appels successifs à l'algorithme de détection de collision, les objets se déplacent seulement d'une courte distance, cette méthdode s'exécute en temps presque constant pour le cas de polyèdres convexes (ce qui est le cas pour les robots). Les objets non convexes peuvent être décomposés en composantes convexes.

3.3 Planification

Le module (partie du logiciel) chargé de la construction d'un chemin libre de toute collision menant au but va construire ce chemin pas à pas. Celui-ci sera défini comme une succession de segments de droite, chaque segment étant relié au précédent et au suivant par les sommets qui sont à ses extrémités. On peut donc aussi définir ce chemin par la suite ordonnée de sommets de l'espace qui mènent du point de départ au but, qui sont respectivement le premier et le dernier point de la suite. Chaque étape de la construction du chemin va consister à déterminer le sommet suivant, une condition afin que celui-ci soit valide étant qu'il corresponde à une configuration exempte de collision. Pour le vérifier, le module de planification communique au module de détection de collision les coordonnées du sommet et la géométrie de l'objet. Le module de détection de collision renvoie vrai ou faux selon qu'il a détecté ou non une collision. L'ordre de grandeur du nombre d'appels au module de détection de collision est ainsi le nombre de sommets que comporte le chemin, à un facteur multiplicatif près car il faut aussi comptabiliser les appels qui feront l'objet d'une réponse négative car une collision aura été détectée (nombre d'essais versus nombre d'essais valides). La détection de collision étant la partie la plus coûteuse en temps de calcul d'un logiciel de planification, il convient de limiter au maximum le nombre d'appels au module de détection de collision et donc le nombre de sommets que comportera le chemin. Mais on comprend bien que réduire ce nombre présente deux inconvénients :

— plus les points sont espacés et plus le test de détection de collision est susceptible d'être entaché d'une erreur. En effet seules les configurations correspondant aux extrémités de chaque segment sont testées et on fait l'hypothèse que si cette condition est satisfaite alors tous les autres points du segment ne génèrent pas de collision. On comprend bien que plus les arêtes seront longues, moins cette hypothèse sera réaliste.

— plus les points sont espacés et moins la trajectoire calculée sera une bonne approximation d'une trajectoire "naturelle", c'est-à-dire celle que pourrait utiliser un individu qui se déplacerait du point de départ au but en évitant les obstacles. Si dans un jeu vidéo par exemple on fait se déplacer un avatar (personnage virtuel) le long de ce chemin, le déplacement de celui-ci ne paraîtra pas naturel car il semblera zigzaguer de droite à gauche plutôt que d'emprunter une ligne ou une courbe régulière.

Il y a donc un compromis à trouver entre la complexité du calcul et la qualité du résultat. Il pourra être trouvé par essais successifs, en faisant varier le pas entre chaque sommet calculé et en vérifiant le nombre de collisions manquées lorsqu'on augmente ce pas. On pourra également apprécier la régularité de la trajectoire obtenue.

4 Algorithmes de planification

4.1 A^*

L'algorithme A^* a été proposé pour la première fois par Peter E. Hart, Nils John Nilsson et Bertram Raphael en 1968. C'est un algorithme simple de recherche de chemin dans un graphe entre un nœud initial et un nœud final tous deux donnés. Il s'agit d'une extension de l'algorithme de Dijkstra de 1959.

L'algorithme de Dijkstra qui porte le nom de son inventeur, l'informaticien néerlandais Edsger Dijkstra, sert à résoudre le problème du plus court chemin. Ce problème a plusieurs variantes. La plus simple est la suivante : étant donné un graphe non orienté, dont les arêtes sont munies de poids, trouver un chemin entre deux sommets donnés du graphe, de poids minimum. L'algorithme de Dijkstra calcule des plus courts chemins à partir d'une source vers tous les autres sommets dans un graphe orienté pondéré par des réels positifs.

La figure 2 montre les étapes successives dans la résolution du chemin le plus court dans un graphe non-orienté. Les nœuds symbolisent des villes identifiées par une lettre et les arêtes indiquent la distance entre ces villes. On cherche à déterminer le plus court trajet pour aller de la ville A à la ville J. En neuf étapes, on peut déterminer le chemin le plus court menant de A à J, il passe par C et H et mesure 487 km.

Mais Dijkstra n'est pas vraiment adapté pour chercher une seule cible donnée, puisqu'il calcule les plus courtes distances d'un point à tous les autres points d'un graphe. L'algorithme A^* s'appuie sur celui de Dijkstra et introduit donc la notion de cap. Dans A^*, on fixe un cap, et on essaie de le suivre avec plus ou moins de précision en le réévaluant au fur et à mesure.

Principe : il est identique à celui de Dijkstra (croissance d'un arbre de racine s, la source, par ajout de feuilles) sinon que le choix du sommet u se fait selon $score[u]$, une valeur qui tient compte non seulement de $cot[u]$ (du coût du chemin dans l'arbre de s à u), mais aussi d'une estimation de la distance entre u et la cible t.

Cette estimation est ce qu'on appelle une heuristique. C'est une fonction $h(x, t)$

G : graphe orienté
$w : E(G) \to \mathbb{R}^+$
s : source de G

Algorithme 7 Dijkstra(G, w, s)

```
1: pour tout v de V(G) faire
2:    d(v) ← ∞
3:    pere(v) ← NIL
4:    couleur(v) ← BLANC
5: fin pour
6: d(s) ← 0
7: F ← FILE_PRIORITE({s}, d)
8: tant que F ≠ ∅ faire
9:    pivot ← EXTRAIRE_MIN(F)
10:   pour tout e = (pivot, v) arc sortant de pivot faire
11:      si couleur(v) = BLANC alors
12:         si d(v) = ∞ alors
13:            INSERER(F, v)
14:         fin si
15:         si d[v] > d[pivot] + w(e) alors
16:            d[v] ← d[pivot] + w(e)
17:            pere[v] ← pivot
18:         fin si
19:      fin si
20:   fin pour
21:   couleur[pivot] ← NOIR
22: fin tant que
```

FIGURE 1 – Algorithme de Dijkstra.

définie pour tout sommet x du graphe.
Def : algorithme supposé efficace en pratique qui produit un résultat sans garantie de qualité par rapport à la solution optimale.
A^* est un algorithme simple, ne nécessitant pas de prétraitement, et ne consommant que peu de mémoire.

Exemple : On se trouve à l'intersection A et on veut se rendre à l'intersection B dont on sait qu'elle se trouve au nord de notre position actuelle. Pour ce cas classique, le graphe sur lequel l'algorithme va travailler représente la carte, où ses arcs représentent les chemins et ses nœuds les intersections des chemins.
Si on faisait une recherche en largeur comme le réalise l'algorithme de Dijkstra, on chercherait tous les points dans un rayon circulaire fixe, augmentant graduellement ce cercle pour rechercher des intersections de plus en plus loin de notre point de départ. Ceci pourrait être une stratégie efficace si on ne savait pas où se trouve notre destination.
Cependant, c'est une perte de temps si on connaît plus d'informations sur notre

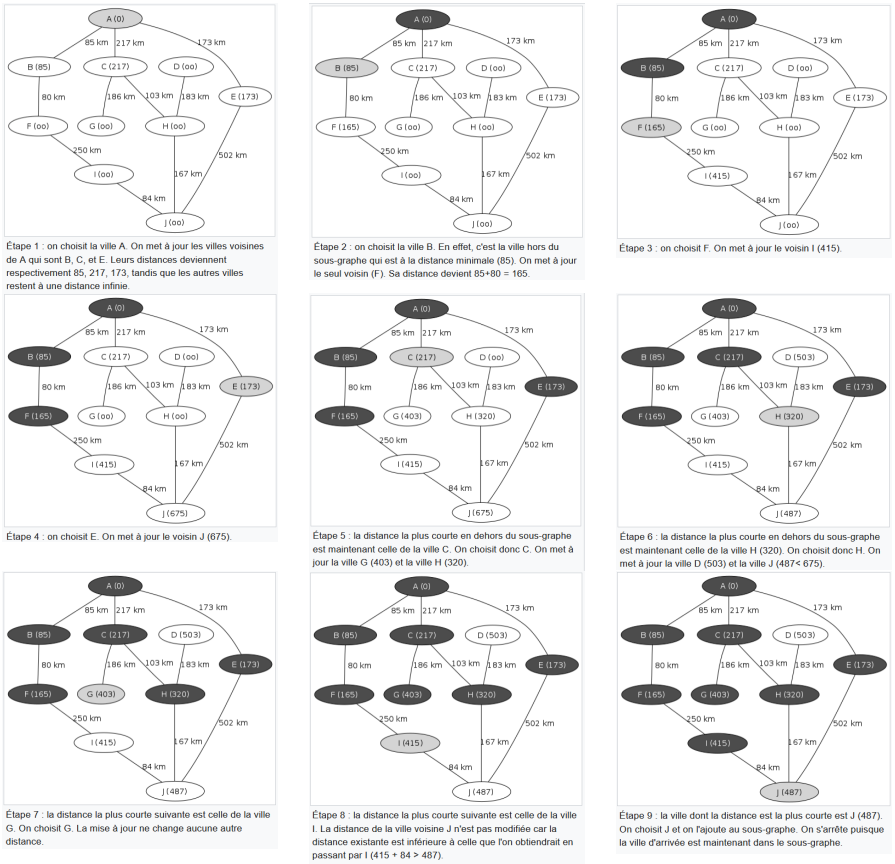

FIGURE 2 – Etapes successives de la construction d'un plus court chemin allant de A à J.

destination. Une meilleure stratégie est d'explorer à chaque intersection la première directement qui va vers le nord, car le chemin le plus court est la ligne droite. Tant que la route le permet, on continue à avancer en prenant les chemins se rapprochant le plus de l'intersection B. Certainement devra-t-on revenir en arrière de temps en temps, mais sur les cartes typiques c'est une stratégie beaucoup plus rapide. D'ailleurs, souvent cette stratégie trouvera le meilleur itinéraire, comme la recherche en largeur le ferait. C'est l'essence de la recherche de chemin A^*.
Toutefois, comme pour tous les algorithmes de recherche de chemin, leur efficacité dépend fortement du problème que l'on souhaite résoudre (c'est-à-dire : du graphe).
Ainsi l'algorithme A* ne garantit pas de trouver un itinéraire plus rapidement qu'un autre algorithme, et dans un labyrinthe, la seule manière d'atteindre la destination pourrait être à l'opposé de la position de la destination, et les nœuds

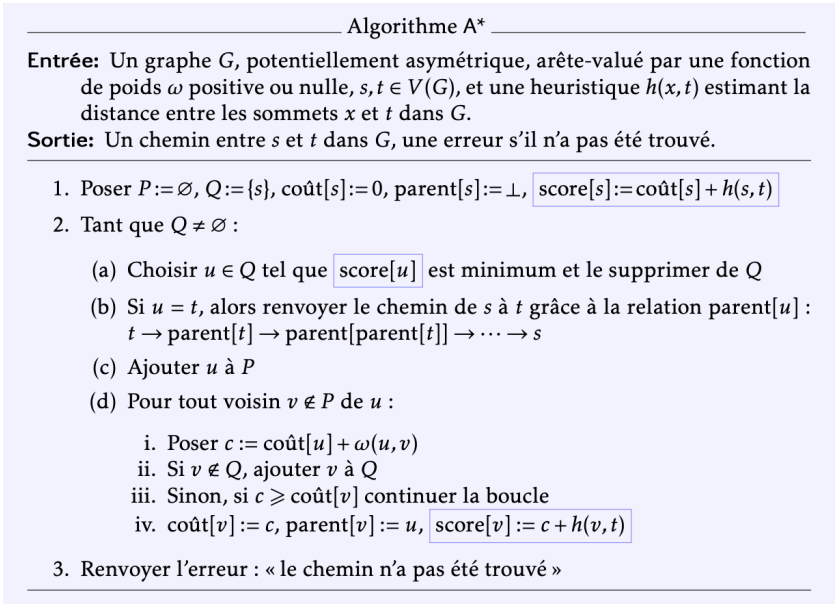

FIGURE 3 – Algorithme A^*.

les plus proches de cette destination pourraient ne pas être sur le chemin le plus court, ce qui peut coûter beaucoup de temps de calcul.

Par extension de Dijkstra, l'algorithme A^* assure de trouver un chemin entre 2 points lorsqu'il existe, il est donc dit complet. Nous allons à présent prouver l'optimalité de A^*, c'est-à-dire le fait que lorsqu'il existe, il trouve bien le plus court chemin entre deux points dans un graphe.

Nous allons à présent prouver que l'algorithme A^* est bien un algorithme de plus court chemin entre deux points dans un graphe. Comme expliqué précédemment, A^* est une combinaison de l'algorithme de Dijkstra et d'une estimation de la distance de chaque noeud au point d'arrivée. Cette estimation est une fonction $h(x,t)$ qui est une heuristique sur la distance entre un sommet x et le point d'arrivée t. Pour que A^* calcule un plus court chemin, il faut que l'heuristique sous estime la distance.

Définition L'heuristique h sous-estime la distance si $h(x,y) \leq distG(x,y)$ pour toutes les paires de sommets (x,y) où h est définie.

Preuve Soit C le plus court chemin retourné par A^*. Supposons qu'il existe un chemin plus court que l'on appelle C'. Considérons le moment juste avant de choisir le sommet d'arrivée t à l'étape 2a. Alors à ce moment, une partie du chemin C' n'a pas encore été explorée. Appelons cette partie C'' et notons c le sommet de C'' ayant le plus petit score. Puisque l'algorithme A^* a choisi t à l'étape 2a, et non c'', $score[t] \leq score[c'']$. Comme t est le point d'arrivée, $h(t) = 0$. Donc $cot[t] \leq cot[c''] + h(c'')$. Or h sous estime la distance, donc $cot[c''] + h[c''] \leq cot(C') = dist(s,t)$.

Finalement, $cot[t] = cot(C) \leq cout(C')$, ce qui contredit la minimalité de C'.

4.2 *RRT*

Tandis que l'algorithme A^* est basé sur la recherche, l'algorithme *RRT* est basé sur l'échantillonnage. *RRT* signifie Rapidly Exploring Random Trees, en anglais, qui veut dire arbre aléatoire à exploration rapide. Cet algorithme construit au hasard un arbre remplissant l'espace. Les *RRT* ont été développés par Steven M. LaValle et James J. Kuffner Jr et gèrent facilement les problèmes d'obstacles.

[Ici une série de 16 images et explications pour suivre pas à pas le fonctionnement de l'algorithme RRT]

A cette étape, nous avons un chemin viable. Bien que, probablement pas optimal puisque cette méthode a tendance à zigzaguer au fur et à mesure de son cheminement. Mais, nous avons trouvé une solution, et surtout une solution qui utilise probablement moins de nœuds que ce qui aurait été nécessaire pour A*, car les nœuds peuvent être plus espacés, sans oublier la distance maximale que nous avions fixée.

Ainsi, tant que les nœuds de départ et d'arrivée sont accessibles, il est très probable que *RRT* soit complet, c'est-à-dire qu'il trouve un chemin quand le nombre d'échantillons approche l'infini, et, dans la plupart des cas, un nombre fini d'échantillons beaucoup plus petit.

4.3 Comparaison A^* et *RRT*

Dans cette partie nous allons essayer de comparer les 2 algorithmes précédemment présentés. Il ne s'agit pas ici de répondre à la question "Quel est le meilleur algorithme" mais bien de présenter des approches de comparaison en fonction de différents facteurs.

Dans le cadre de la planification de mouvement pour les robots, il y a des enjeux :

- Calculer le chemin optimal
- Générer rapidement des actions dans le cas d'obstacles imprévus (être efficace et fiable dans un environnement dynamique)

La performance d'un algorithme de recherche peut être ainsi mesurée de 4 façons différentes :

- Complétude : est-ce que l'algorithme trouve une solution lorsqu'il y en une ?
- Optimalité : est-ce que la solution est la meilleure solution, en terme de coût du chemin ?
- Complexité en temps : combien de temps l'algorithme prend t-il pour trouver une solution ?
- Complexité en espace : quel espace mémoire est nécessaire pour faire la recherche ?

4.3.1 A^*

A^* trouve toujours une solution car cet algorithme est basé sur l'algorithme de Dijkstra qui garantit de trouver une solution si celle-ci existe.

Le terme "optimalité" dans le contexte de la planification de mouvement fait généralement référence à trouver le chemin le plus court entre deux positions, mais dans le cadre des robots, il fait aussi référence au chemin le plus rapide. Pour de faibles vélocités la distance à la cible est un bon indicateur du temps que va mettre le robot pour se rendre à la cible, car celui-ci peut freiner en une fraction de seconde. Cependant pour des vélocités plus grandes, des chemins plus rapides peuvent être construits autrement. En effet, il arrive que pour de grandes vélocités, des chemins plus longs peuvent amener à un temps de voyage plus court quand des obstacles sont présents.

L'optimalité de A^* a été montrée dans la partie X, en effet la solution trouvée par A^* est garantie d'être optimale à condition que l'heuristique choisie soit admissible, c'est-à-dire qu'elle sous-estime toujours le coût pour atteindre l'objectif. De plus si A^* est admissible (complet et optimal) et possède une heuristique admissible, nous pouvons même aller plus loin et montrer que A^* est d'une efficacité optimale, c'est-à-dire que parmi tous les autres algorithmes qui étendent la recherche du chemin à partir d'un état initial, aucun autre algorithme ne peut faire mieux en terme de nombre de nœuds étendus. A^* trouve le chemin le plus court avec un nombre minimum d'expansion. Tout algorithme qui utilise la même heuristique que A^* et possède le même noeud de départ et qui n'étend pas tous les noeuds ayant un $fcost < f*$ (le chemin optimal trouvé par A^*) risque de passer à côté de la solution optimale.

En effet, supposons $f*$ le chemin le plus court à une cible et considérons un algorithme A', qui a la même heuristique et le même noeud de départ que A^*, et qui manque d'étendre le chemin p' étendu par A^* tel que $fcost(p') = g(p') + h(p') < f*$.
$g(p')$ = distance par rapport au noeuds de départ.
$h(p')$ = l'heuristique qui estime la distance jusqu'au noeud d'arrivée.
On suppose que A' est optimal.
Considérons, un problème de recherche différent qui est identique à l'original et sur lequel h retourne les mêmes estimations que le problème original pour chaque chemin, excepté que p' à un chemin fils p'' qui va jusqu'à la cible (p'' est la cible).
On a donc $fcost(p'') = g(p'')(h(p'') = 0)$.
Or $fcost(p'') = g(p')+h(p') = fcost(p')$ donc le véritable coût du chemin jusqu'à p'' est $f(p')$ (l'heuristique est précisément correcte sur le coût pour aller de p' à la cible). A' se comporterait pareil sur ce nouveau problème, la seule différence étant que A' n'étend pas le chemin au-delà de p' et donc ne trouvera jamais ce chemin optimal. On a $fcost(p'') = fcost(p') < f*$ donc le coût du chemin jusqu'à p'' est plus petit que le coût du chemin trouvé par A' $\Rightarrow$ ABSURDE car A' est supposé optimal.

4.3.2 *RRT*

Contrairement à A^* (basé sur la recherche), on ne peut pas être complètement sûr que l'algorithme *RRT* (basé sur l'échantillonnage) est complet. Cependant, il existe de nombreuses preuves qui montre que *RRT* est probablement complet, ie, si une solution existe la probabilité que l'algorithme la trouve tends vers 1 quand le nombre d'itérations tends vers l'infini. De plus, il a aussi été prouvé que certaine versions de *RRT* ($RRT*$) sont asymptotiquement optimal, c'est-à-dire, qu'il produit de façon presque sûr une solution qui se rapproche de la solution optimale quand le nombre d'itérations tend vers l'infini.

L'un des avantages des A^* c'est qu'il est assuré de trouver la solution optimale si elle existe et c'est un algorithme intéressant et performant pour des espaces de configurations de faible dimension (2D/3D) avec des obstacles statiques. Mais pour des espaces de configuration de haute dimension (complexité du robot, nombre de joints,complexité de l'environnement ...), il devient intraitable sur le plan informatique et le réaménagement du chemin est impossible, ce qui rend l'algorithme A^* inefficace dans des environnements dynamiques.

Les algorithmes basés sur l'échantillonnage tel que *RRT* ne caractérisent pas l'espace de configuration de façon explicite mais utilisent un algorithme pour sonder l'espace de configuration et regarder si une configuration appartient à l'espace libre(sans obstacle) ou non. *RRT* est un algorithme spécifiquement conçu pour être efficace dans des hautes dimensions, en construisant un arbre qui remplit aléatoirement l'espace de configuration.

5 En pratique

Afin de comparer les algorithmes en pratique, nous avons travaillé avec un Baptiste Bordenave, étudiant en DUT à Bordeaux actuellement en stage dans l'équipe NAELIC. Il s'est occupé d'implémenter les algorithmes A* et RRT que nous lui avons fournis en python sur ROS, un logiciel de robotique, afin d'étudier les comportements des robots. Le cadre utilisé pour la simulation est celui de la ligue SSL (même terrain et mêmes robots).

Le premier algorithme utilisé est A*. Celui-ci présente, en pratique, plusieurs inconvénients. Comme il construit le chemin optimal au fur et à mesure, en ajoutant un à un les sommets qui minimisent la distance au chemin en construction, le robot qui se déplace marque un temps d'arrêt à chaque fois qu'il avance d'un "pas", c'est-à-dire qu'il rejoint un nouveau sommet. Le déplacement n'est donc déjà pas fluide et donc pas réaliste. De plus, le robot est lent, car entre deux temps d'arrêts, la distance parcourue (celle d'une arête), est tellement courte qu'il n'a pas le temps de prendre de la vitesse.

6 Conclusion

A écrire...

Table des matières

Références