Rapport de TIPE informatique : Algorithmique de la planification de mouvement

Mattéo Clémot

Juin 2019

Résumé

On s'intéresse dans ce travail à des méthodes numériques permettant de planifier le mouvement d'un système dans un environnement donné, dans l'optique d'applications diverses, de la chirurgie robotique au désassemblage industriel. Après quelques considérations sur la modélisation de ces problèmes géométriques, on s'intéresse à une façon de partitionner efficacement un espace de configuration de dimension quelconque, pour utiliser ensuite des algorithmes de recherche de chemin dans un graphe. On étudie également le comportement de ces méthodes avec la dimension et la finesse.

1 Introduction

1.1 Définition du problème

On se donne un *système* pouvant se déplacer dans espace $\mathcal{C}$. Étant donné un *environnement*, on peut définir l'*espace de configuration* $\mathcal{C}^*$ comme l'ensemble des positions que peut prendre le système en respectant des contraintes de non collision avec l'environnement.

On se donne également une position initiale $s \in \mathcal{C}^*$ et une position objectif $g \in \mathcal{C}^*$. La question est de savoir s'il existe un chemin continu de s à g dans $\mathcal{C}^*$ (c'est-à-dire, rigoureusement, un arc $\gamma \in \mathcal{C}^0([0,1], \mathcal{C}^*)$ vérifiant $\gamma(0) = s$ et $\gamma(1) = g$), et d'en déterminer un le cas échéant.

1.2 Environnements tests

1.2.1 Bras robotique

On considère un bras robotique dans le plan à n segments de longueur unitaire et n articulations de même débattement $\alpha \in]0, 2\pi[$ et accroché en l'origine : on prend donc $\mathcal{C} = [-\alpha, \alpha]^n$. On dispose de la suite de points $(A_i)_{i \in [\![0,n]\!]}$ définie par :

$$A_i = \begin{pmatrix} -\sum_{j=0}^{i-1} \sin\left(\sum_{k=0}^{j} \theta_k\right) \\ \sum_{j=0}^{i-1} \cos\left(\sum_{k=0}^{j} \theta_k\right) \end{pmatrix}$$

Étant donné un ensemble S de segments qui constituent les obstacles, on a alors la caractérisation de l'espace de configuration par des contraintes de non collision avec les obstacles et de non autocollision :

$$(\theta_0, ..., \theta_{n-1}) \in \mathcal{C}^* \iff \begin{cases} \forall (i,j) \in [\![0, n-1]\!]^2, i \neq j \Rightarrow [A_i A_{i+1}] \cap [A_j A_{j+1}] = \emptyset \\ \forall (i,s) \in [\![0, n-1]\!] \times S, [A_i A_{i+1}] \cap s = \emptyset \end{cases}.$$

1.2.2 Bras robotique translaté

On peut alors ajouter deux dimension à l'espace $\mathcal{C}$ afin de permettre la translation du bras robotique. On travaille donc désormais dans $\mathcal{C} = [-\alpha, \alpha]^n \times [-\beta, \beta] \times [-\gamma, \gamma]$ si on suppose que l'origine peut se déplacer dans $[-\beta, \beta] \times [-\gamma, \gamma]$.

1.2.3 Autres environnements

Parmi de très nombreuses possibilités, on peut également considérer un robot de forme polygonale P se déplaçant dans le plan, avec un ensemble $\mathcal{E}$ d'obstacles ponctuels : $\mathcal{C}$ est l'ensemble des isométries affines du plan conservant l'orientation : $\mathcal{C} = \mathbb{R}^2 \times SO(2)$ et $\mathcal{C}^* = \{(u, r) \in \mathcal{C} | (u + r(P)) \cap \mathcal{E} = \emptyset\}$.

1.3 Résolution de problèmes géométriques

Afin de vérifier si une une configuration est valide, il est nécessaire de résoudre des problèmes géométriques comme l'intersection de deux segments ou l'appartenance d'un point à un polygone.

Pour tester l'intersection de deux segments $[ab]$ et $[cd]$ non parallèles, on résout le système $a + t(b-a) = c + s(d-c)$ avec la formule de Cramer, et on a alors l'équivalence (Annexe **??**) :

$$[ab] \cap [cd] \neq \emptyset \iff (s,t) \in [0,1]^2.$$

Pour tester l'appartenance d'un point à polygone, on décompose celui-ci en la réunion de n-gones convexes que l'on décompose eux-mêmes en la réunion de $n-2$ triangles. Il s'agit alors de tester l'appartenance du point à chacun de ces triangles, en utilisant les coordonnées barycentriques et l'équivalence :

$$P \in ABC \iff (|\overrightarrow{PB}, \overrightarrow{PC}|, |\overrightarrow{PC}, \overrightarrow{PA}|, |\overrightarrow{PA}, \overrightarrow{PB}|) \in [0,1]^3.$$

2 Discrétisation naïve

2.1 Discrétisation

On s'intéresse à un résolution numérique de ce problème. On peut pour cela travailler dans un espace discrétisé, puis se ramener à un problème de recherche de chemin dans un graphe associé à cette discrétisation.

On considère pour simplifier des espaces $\mathcal{C}$ comme des n-orthotopes centrés (appelées ci-après *boîtes*) c'est-à-dire de la forme $\mathcal{C} = [-\alpha_0, \alpha_0] \times ... \times [-\alpha_{n-1}, \alpha_{n-1}], \quad \alpha \in (\mathbb{R}_+^*)^n$.

On se donne également un vecteur $N \in (\mathbb{N}^*)^n$ qui représente le nombre d'échantillons dans chaque dimension de l'espace. On travaille alors en coordonnées $u \in [\![0, N_0 - 1]\!] \times ... \times [\![0, N_{n-1} - 1]\!]$ associées à une configuration $c = \left(-\alpha_0 + u_0 \frac{2\alpha_0}{N_0 - 1}, ..., -\alpha_{n-1} + u_{n-1} \frac{2\alpha_{n-1}}{N_{n-1}-1}\right)$.

On considère alors le graphe dont les sommets sont les éléments de $[\![0, N_0 - 1]\!] \times ... \times [\![0, N_{n-1} - 1]\!]$. Selon les cas, les arêtes sont définies pour les voisins en norme infinie (il est possible pour le bras robot d'actionner plusieurs moteurs en même temps sans pénalité en temps) ou en norme 1.

On cherche alors au sein de ce graphe (pour le moment non pondéré) un chemin reliant le nœud initial s au nœud final g.

2.2 Exploration naïve

Un simple parcours largeur à partir de la position initiale permet de trouver le plus court chemin la reliant à la position finale, sous réserve d'existence. En pratique, on commence par mettre le départ dans une file. Tant qu'elle n'est pas vide, on considère la position en tête, on la marque et on ajoute à la file ses voisins non encore marqués. Il s'agit en réalité de l'algorithme de Dijkstra (Algorithme 1) appliqué au cas particulier d'un graphe non pondéré.

Algorithm 1 Algorithme de Dijkstra

```
U ← {s}; λ(s) ← 0
∀v ≠ s, λ(v) ← +∞
while U ≠ ∅ do
  u ← argmin_{u'∈U} λ(u')
  for v ∈ voisins(u) do
    if λ(v) > λ(v) + d(u,v) then
      λ(v) ← λ(v) + d(u,v)
      U ← U ∪ {v}
    end if
  end for
  U ← U \ {u}
end while
```

2.3 Exploration A*

Avec un parcours en largeur, on explore des directions qui *a priori* ne sont pas le meilleur choix pour atteindre la destination. Or on peut estimer (et plus précisément minorer) la distance du chemin restant le plus court par la distance spatiale dans $\mathcal{C}$. On emploie donc l'algorithme A* (Algorithme 2), qui s'avère généralement plus rapide que le parcours en largeur, sans toutefois garantir d'obtenir le plus court chemin. On adopte donc

l'heuristique h qui est la distance spatiale du barycentre du nœud à la destination g. Elle est dite admissible car elle ne surestime jamais la plus courte distance réelle.

Algorithm 2 Algorithme A*

$U \leftarrow \{s\}; \lambda(s) \leftarrow h(s)$
$\forall v \neq s,\ \lambda(v) = c(v) = +\infty$
while $U \neq \emptyset$ **do**
  $u \leftarrow \text{argmin}_{u' \in U} \lambda(u')$
  for $v \in \text{voisins}(u)$ **do**
    if $c(v) > c(u) + d(u, v)$ **then**
      $c(v) \leftarrow c(u) + d(u, v)$
      $\lambda(v) \leftarrow c(v) + h(v)$
      $U \leftarrow U \cup \{v\}$
    end if
  end for
  $U \leftarrow U \setminus \{u\}$
end while

2.4 Résultats

La Figure 1 compare les algorithmes de Dijkstra et A* pour la discrétisation naïve. On observe bien une réduction significative du nombre de configuration explorées.

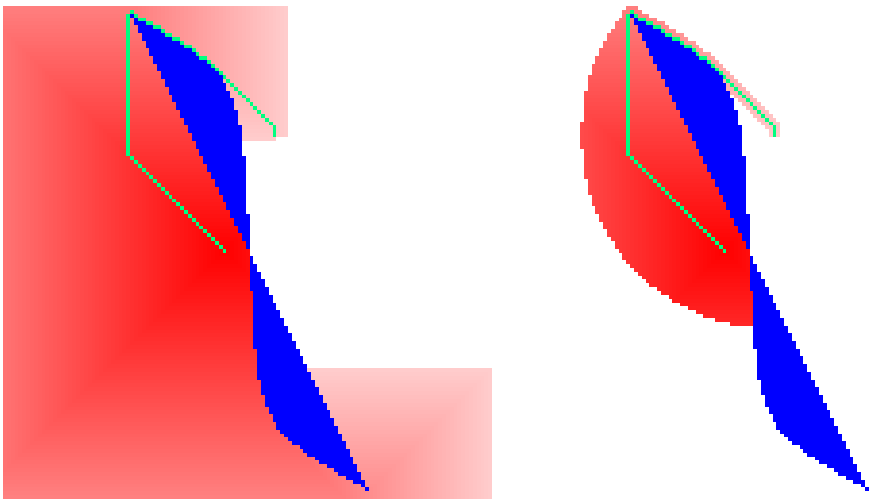

FIGURE 1 – À gauche : exploration bidimentionnelle par un parcours en largeur. À droite : exploration bidimentionnelle A*. La discrétisation est de taille 128^2, les configurations interdites sont en bleu, le dégradé de rouge indique la distance à la configuration initiale, et le chemin trouvé est en vert.

3 Finesse adaptative

La partie précédente montre qu'une grande partie des positions explorées le sont inutilement, dans la mesure où l'on explore précisément de grandes régions où il n'y a pas d'obstacle. On souhaite donc désormais adapter la finesse de la discrétisation à la proximité avec un obstacle : plus spécifiquement, on cherche à être fin au niveau de l'interface $\partial\mathcal{C}^*$, et plus grossier loin de cette interface, afin de gagner en temps de calcul lors de la recherche d'un chemin.

3.1 Arbres pour la partition de l'espace

Dans un premier temps, on procède à une division récursive de l'espace. Pour une boîte donnée, si elle possède des sommets dans $\mathcal{C}^*$ et dans $\overline{\mathcal{C}^*}$, ou si la transition associée à l'une de ses arêtes n'est pas licite, on la divise en 2^d boîtes de côté deux fois plus petit. La construction se termine en s'arrêtant à une taille de boîte suffisamment petite, ou en limitant la hauteur de l'arbre (Figure 2).

Pour représenter cette partition, on utilise la structure d'arbre dont les noeuds sont d'arité 0 ou d (Figure 3). La position d'un nœud par rapport à ses frères peut être codée par un vecteur de $\{0,1\}^d$, qui peut être vu comme l'écriture binaire d'un entier de $[\![0, 2^d - 1]\!]$.

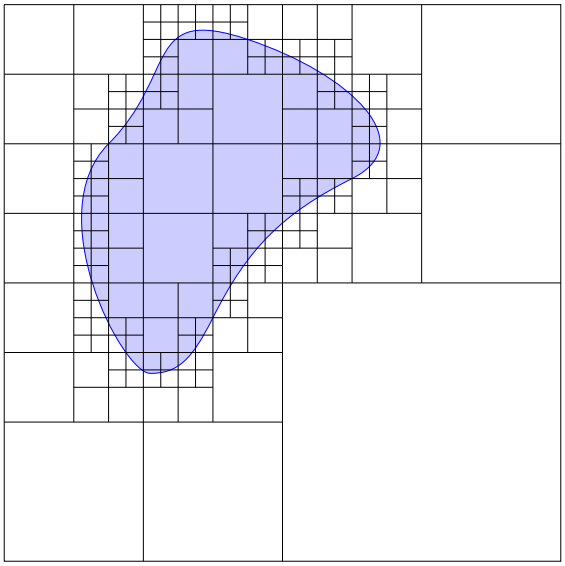

FIGURE 2 – Partition de l'espace ($d = 2$), le complémentaire de l'espace des configurations $\overline{\mathcal{C}^*}$ étant la zone bleue et l'interface $\partial\mathcal{C}^*$ la ligne bleue, l'arbre sous-jacent ayant une hauteur imposée à 5.

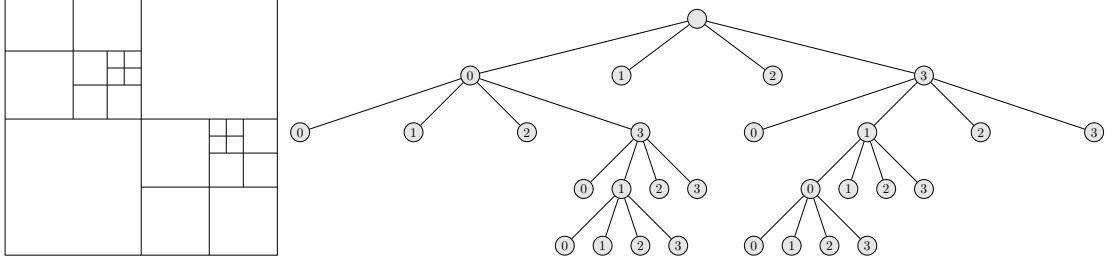

FIGURE 3 – Exemple de *quadtree* (à droite) associé à la partition de l'espace ($d = 2$) à sa gauche.

L'implémentation en C++ exploite le paradigme orienté objet du langage, en utilisant une classe récursive pour stocker l'arbre (Annexe **??**). Chaque nœud possède :

— un booléen `leaf` indiquant si c'est une feuille ;
— la position `coord0` et la taille `size` de la boîte qu'il représente ;
— son codage `child_index` parmi ses frères (sous la forme d'un entier) ;
— un tableau `children` de 2^d pointeurs vers ses éventuels enfants ;
— un pointeur `parent` vers son parent s'il existe.

3.2 Étude de la taille de l'arbre

La construction de l'arbre se fait en fixant sa hauteur. On étudie alors le nombre de nœuds de l'arbre en fonction de cette hauteur et de la dimension de l'espace $\mathcal{C}$ (Figure 4).

Une régression linéaire donne des pentes respectives de 0.97, 1.98, 2.92 avec de bons coefficients de corrélation. On obtient empiriquement la loi $\log_2(N) = (d-1)h + C$ ou encore $N = C2^{(d-1)h}$. Ce résultat est à mettre en relation au nombre d'éléments de la discrétisation naïve qui est 2^{hd}. On peut interpréter de manière informelle cette *réduction de dimension* par le fait que, par le principe même de la construction de l'arbre, la plupart de ses nœuds se concentrent spatialement au niveau de l'interface $\partial\mathcal{C}^*$, qui pour un espace $\mathcal{C}$ de dimension d, est une *hypersurface* de dimension $d - 1$.

3.3 Insertion du départ et de l'arrivée

Pour finir la construction, on insère récursivement (Annexe **??**) la position initiale et la position finale, en divisant si nécessaire le nœud et en insérant récursivement dans la sous-boîte dans laquelle se trouvent les

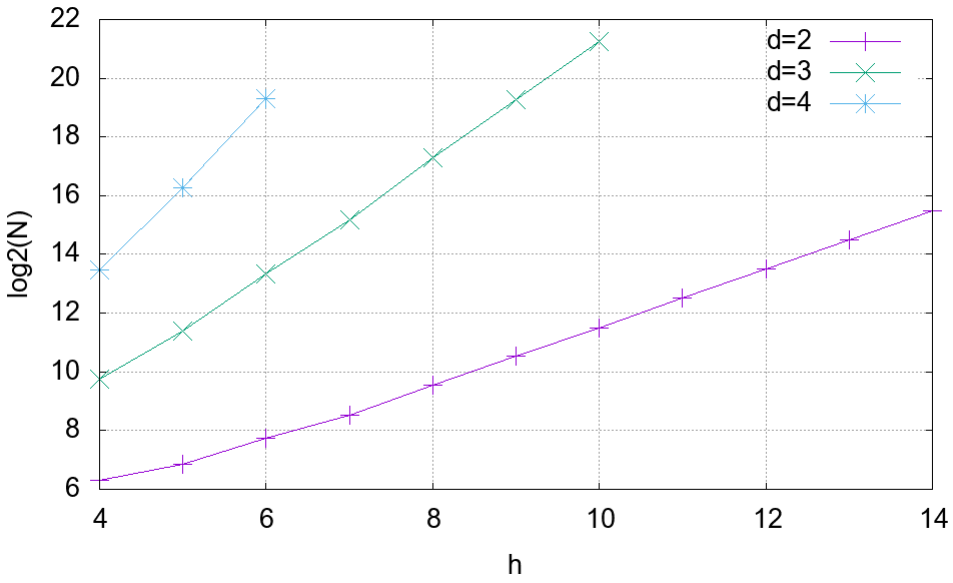

FIGURE 4 – Graphique du logarithme du nombre de nœuds de l'arbre en fonction de la hauteur imposée et de la dimension de l'espace, pour le bras robotique fixé (Annexe A.1).

coordonnées de ces positions. Cela permet par ailleurs d'obtenir un pointeur sur les nœuds correspondant à ces positions.

3.4 Graphes pour la partition de l'espace

L'exploitation de cette partition de l'espace se fait en considérant le graphe dont les nœuds sont les feuilles de l'arbre, deux nœuds étant reliés si et seulement si les boîtes qu'ils représentent ont une frontière commune (voir Figure 5).

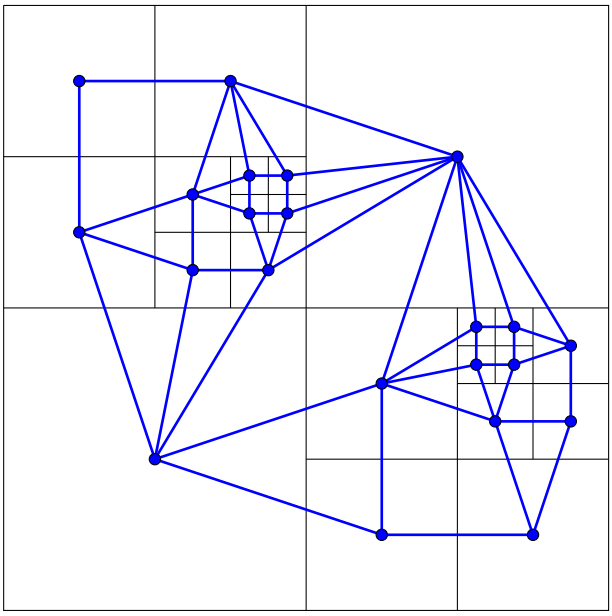

FIGURE 5 – Graphe associé à une partition de l'espace ($d = 2$).

Les liens entre nœuds du graphe ne sont pas déterminés lors de la construction de l'arbre, si bien que la structure de graphe en elle-même n'est pas stockée en mémoire. Au contraire, on détermine les voisins d'un nœud lorsque cela est requis. Il faut donc écrire un algorithme permettant, étant donné une feuille de l'arbre, de trouver celles qui ont une frontière commune avec elle.

Pour trouver les voisins, il faut regarder dans $2d$ directions (2 sens par dimension). Parmi les 2^d frères d'un nœud, d partagent une frontière avec lui : ce sont ceux donc le codage diffère d'un bit i. Pour ces d directions, on

descend alors récursivement dans l'arbre à partir de ces frères en ne prenant à chaque étape que les 2^{d-1} nœuds dont le bit i du codage est égal à celui de nœud de départ (procédure de descente, voir Annexe **??**). À ce stade, il reste à trouver les voisins dans les d autres directions : l'idée est de remonter l'arbre en ajoutant à chaque étape dans une pile le codage du nœud quitté, jusqu'à ce qu'on dispose d'un nœud frère dans la direction voulue, ou qu'on atteigne la racine. On redescend alors l'arbre à partir de ce frère, vers le nœud associé au codage obtenu en dépilant la pile et en inversant le bit correspondant à la direction, et en s'arrêtant éventuellement lorsqu'une feuille est atteinte (procédure de montée-descente symétriques, voir Annexe **??**). On finit si nécessaire par la même procédure de descente que dans le premier cas.

3.5 Complexité

On s'intéresse à la complexité temporelle de l'obtention des voisins d'un nœud à profondeur k, la hauteur de l'arbre étant notée h. La procédure de descente est effectuée entre d et $2d$ fois, et explore au maximum $(2^{d-1})^{h-k}$ feuilles, d'où une complexité en $\mathcal{O}(d2^{(d-1)(h-k+1)})$. La procédure de montée-descente symétriques à lieu d fois, et est de complexité $\mathcal{O}(k)$. Finalement, on obtient la majoration de complexité temporelle :

$$\mathcal{O}(d(k + 2^{(d-1)(h-k+1)})).$$

On peut maintenant s'intéresser à la complexité de la *traversée en ligne droite à profondeur* $k \geqslant 1$: une telle traversée passe par 2^k nœuds dont il faut à chaque fois calculer les voisins, donnant une complexité totale :

$$\mathcal{O}(dk2^k + 2^{(d-1)(h+1)}).$$

3.6 Exploration

On utilise de nouveaux les algorithmes de Dijkstra (Algorithme 1) et A* (Algorithme 2). Les résultats obtenus sont illustrés Figure 6, et les algorithmes sont implémentés respectivement en Annexes **??** et **??**.

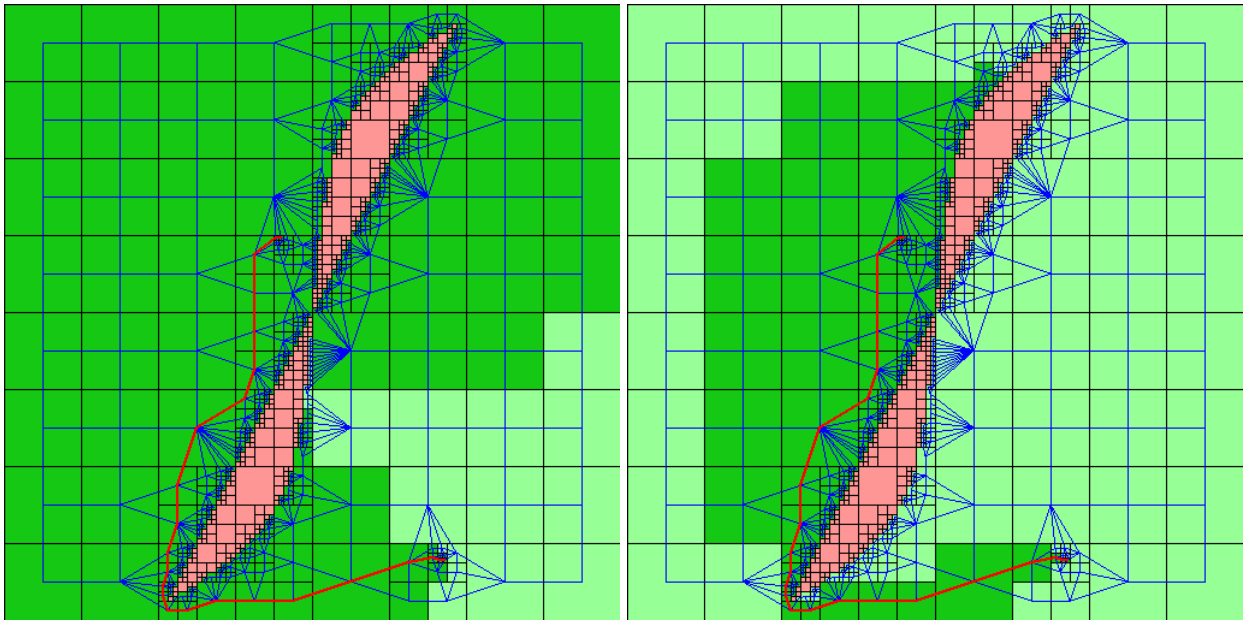

FIGURE 6 – Partition de l'espace (en noir) obtenue pour le bras robotique fixé (Annexe A.1), avec une hauteur imposée à 7, et graphe (en bleu) associé. Les boîtes roses sont interdites, les vertes foncées les boîtes autorisées qui ont été explorées et les vertes pâles celles autorisées mais qui n'ont pas été explorées. À gauche, exploration par algorithme de Dijkstra, à droite par algorithme A*. Le chemin calculé est tracé en rouge.

3.7 Mesure de performance

On mesure ci-dessous le temps de calcul (en millisecondes) d'un chemin pour les différentes méthodes présentées.

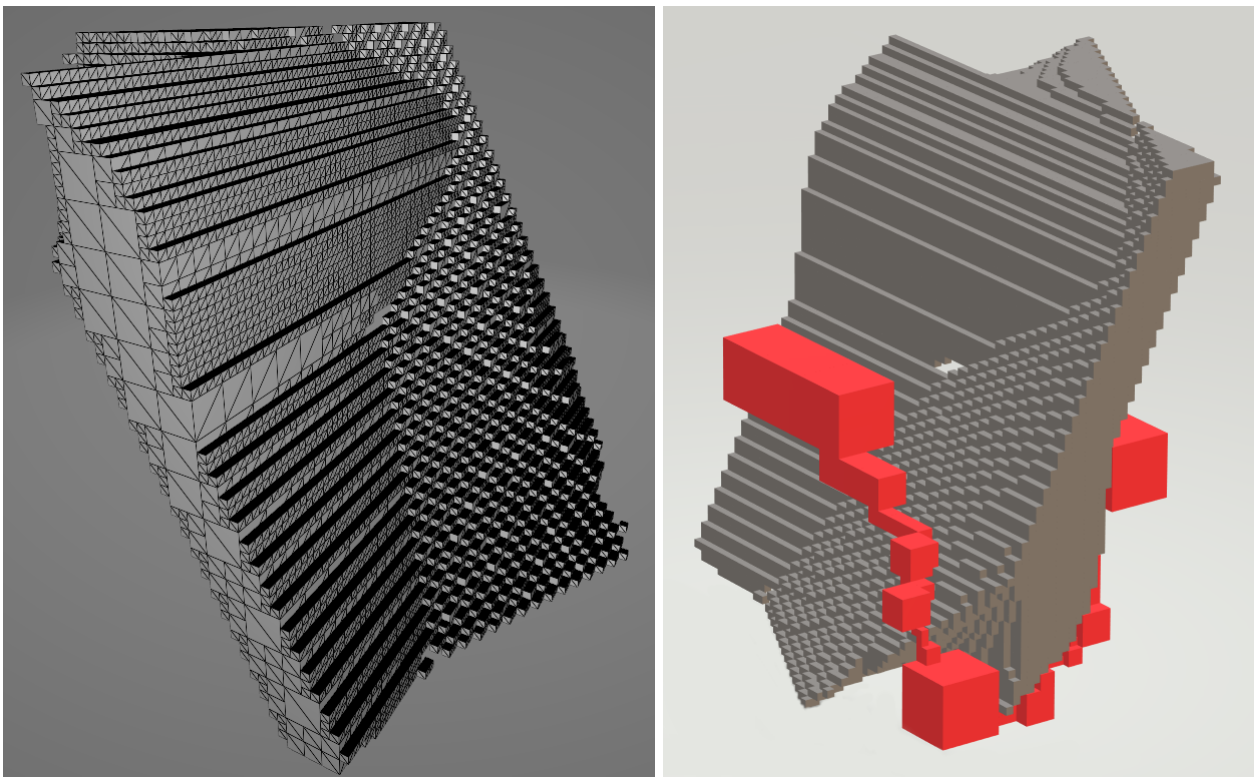

FIGURE 7 – Représentation des boîtes interdites pour une partition de l'espace pour le bras robotique fixé à trois segments. À gauche, mise en évidence de la structure sous-jacente. À droite, le chemin calculé est constitué des boîtes en rouge.

$d = 2$		
Discrétisation naïve		
Hauteur	Dijktra	A*
7	108	43
8	418	183
9	1687	884
10	6765	4874

$d = 2$		
Discrétisation adaptative		
Hauteur	Dijktra	A*
9	31	31
10	62	47
11	140	109
12	313	266
13	859	687
14	2625	2062

$d = 3$	
Discrétisation naïve	
Hauteur	A*
6	2515
7	21565

$d = 3$	
Discrétisation adaptative	
Hauteur	A*
6	313
7	1921
8	22669

De façon analogue à l'étude de la taille de l'arbre construit (section 3.2), on peut remarquer un passage du comportement approximatif du temps de calcul en 2^{dh} vers un comportement en $2^{(d-\delta)h}$ avec $\delta \in [0, 1]$ lors de l'adoption de la discrétisation adaptative.

A Illustration de résultats

Les segments noirs et bleus représentent respectivement le bras robotique et les obstacles.

A.1 Résolution de dimension 2

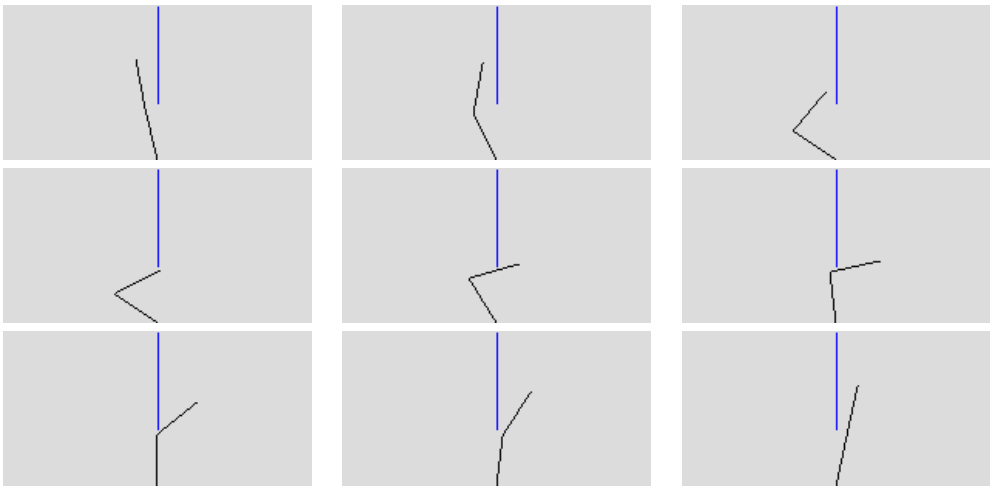

FIGURE 8 – Résolution d'un problème de dimension 2.

A.2 Résolution de dimension 3

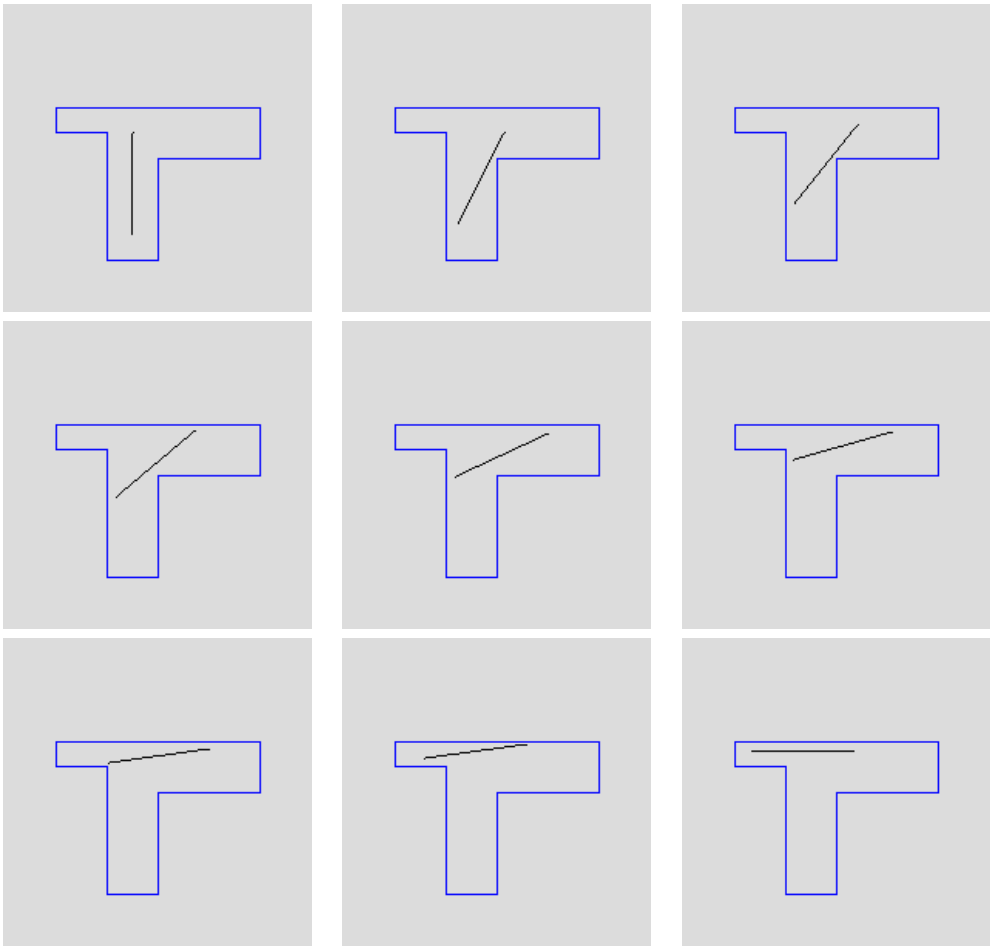

FIGURE 9 – Résolution d'un problème de dimension 3.

A.3 Résolution de dimension 4

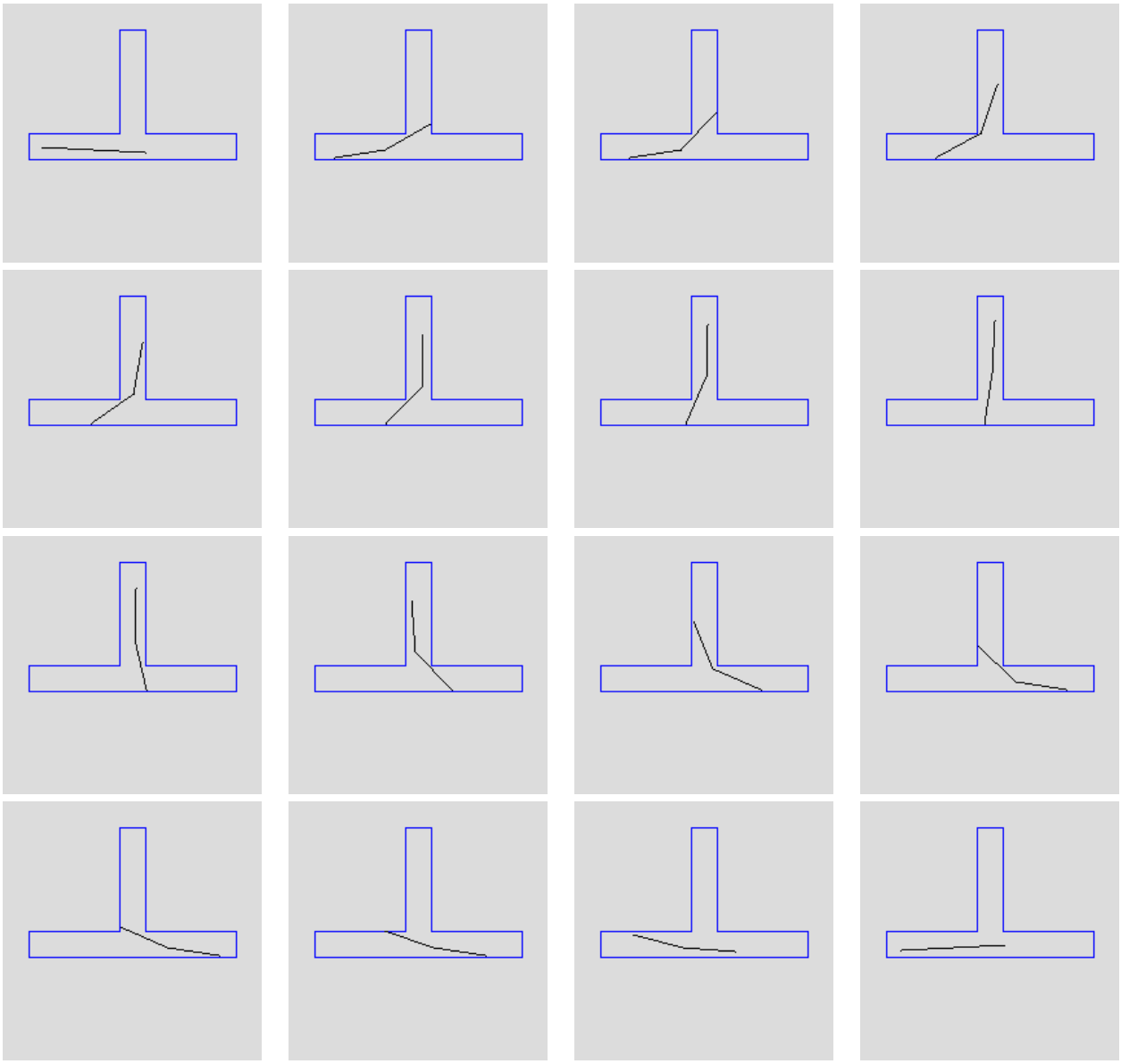

FIGURE 10 – Résolution d'un problème de dimension 4.

Références

[1] Daniele Beauquier, Jean Berstel, and Philippe Chrétienne. *Éléments d'algorithmique*. Masson, 1992.

[2] Luc Jaulin. Path planning using intervals and graphs. *Reliable computing*, 7(1) :1–15, 2001.

[3] Lydia Kavraki, Petr Svestka, and Mark H Overmars. *Probabilistic roadmaps for path planning in high-dimensional configuration spaces*. 1994.

[4] Steven M LaValle. *Planning algorithms*. Cambridge university press, 2006.