Laboratoire d'Ingénierie des Systèmes Automatisés

Université d'Angers

–Modèles et systèmes dynamiques–

DEA Automatique et Informatique Appliquée

Mémoire DEA

thème

PLANIFICATION DE TRAJECTOIRES DE ROBOTS MOBILES VIA DES METHODES ENSEMBLISTES

KENZAI ABDERRAHMANE

Juillet 2005

Responsables de stage : Laurent Hardouin, Professeur *LISA ANGERS*
Lhommeau Mehdi, Maître de Conférences *LISA ANGERS*

Laboratoire d'Ingénierie des Systèmes Automatisés
FRE 2656 CNRS
62, avenue Notre Dame du Lac
49000 Angers

Remerciements

Ce mémoire de DEA présente le travail de recherche que j'ai effectué au sein du **L**aboratoire d'**I**ngénierie des **S**ystèmes **A**utomatisés. J'adresse mes premiers remerciements à **M. Jean-Louis Ferrier**, Directeur de ce laboratoire pour m'y avoir accueilli en stage de DEA.

Je remercie les membres du jury qui ont accepté de juger ce travail et d'y apporter leur caution.

Je remercie tout particulièrement **M. Lhommeau Mehdi** et **M. Laurent Hardouin**, pour la qualité de leur encadrement scientifique, ainsi que pour leurs conseils et leurs remarques avisées.

Je remercie aussi **M. Luc Jaulin**

Je tiens à remercier les membres du LISA, (personnel administratif, chercheurs, et doctorants) qui, à titres divers, ont participé au bon déroulement de ce travail. Je pense en particulier à **M. Samir Hamaci**, pour son aide, sa disponibilité et son soutien pour la réalisation de ce travail.

A la mémoire de mon père (K.R) et de mes grands parents .

Enfin j'adresse mes remerciements à ma mère (K.Z) et à mon oncle et sa femme (K.Y,K.S), et ainsi qu'à toute ma famille et amis qui, de près comme de loin m'ont aidé et encouragé au moment opportun.

Table des matières

Introduction générale

Un robot est une machine équipée de capacités de perception, de décision et d'action qui lui permette d'agir de manière autonome dans son environnement en fonction de la perception qu'il en a.

Les robots mobiles sont largement utilisés dans les environnements industriels pour le transport de produits par exemple. Le plus souvent ces tâches sont répétitives et suivent un chemin bien défini, parfois même bien matérialisé comme des lignes sur le sol. Il y a actuellement une forte tendance à élargir les milieux où évoluent les robots à des environnements de bureaux ou à des environnements domestiques (robots de service). Les types d'applications possibles sont innombrables. Cela peut être des tâches de nettoyage et d'entretien ou encore une assistance à une personne handicapée dans des tâches d'exploration et de préhension. Un robot peut également servir de guide pour la visite d'un musée. On parle alors, de façon générale, de robotique d'intérieur.

Un tel cadre d'utilisation requiert que le système robotisé dispose d'un niveau minimum d'autonomie et de facilités de navigation. Pour ce faire, le système doit généralement accomplir trois tâches de base qui sont la localisation, la planification et la navigation. La robotique mobile vise à rendre un système autonome dans ses déplacements. Un robot, doté de capacités de perception et d'information sur son environnement, doit pouvoir se mouvoir en autonomie, sans se perdre et tout en évitant les obstacles. Une des tâches que doit accomplir le robot et donc de planifier sa trajectoire dans l'environnement.

Chapitre 1

Planification de trajectoires et chemins optimaux

Dans ce chapitre, nous présentons le problème général de planification et nous passons en revue les grandes approches et méthodes de résolution proposées dans la littérature et pertinentes pour le problème abordé dans ce manuscrit. Cela comprendra en plus des techniques purement géométriques, une synthèse des travaux majeurs en planification de mouvements en présence de contraintes holonomes et de contraintes dynamiques. Ensuite nous présentons les principaux algorithmes de recherche du plus court chemin.

1.1 La problématique générale

Pour un robot $\boldsymbol{A}$ évoluant dans un environnement $\boldsymbol{W}$ donné, le problème général de planification consiste à déterminer pour $\boldsymbol{A}$ un mouvement lui permettant de se déplacer entre deux configurations données tout en respectant un certain nombre de contraintes et de critères. Ceux-ci découlent de plusieurs facteurs de natures diverses et dépendent généralement des caractéristiques du robot, de l'environnement et du type de tâche à exécuter. En l'occurrence, les contraintes relatives au robot concernent sa géométrie, sa cinématique et sa dynamique et leur prise en compte peut être complexe selon l'architecture initiale considérée. Cette architecture pouvant correspondre à un système articulé d'objets rigides tel qu'un bras manipulateur, une main à plusieurs doigts ou un véhicule tractant des remorques, ou encore à plusieurs systèmes de robots à coordonner tels que des bras manipulateurs ou des robots mobiles de type voiture évoluant dans un réseau routier. Les contraintes émanant de l'environnement concernent essentiellement la non collision aux obstacles fixes encombrant W et la prise en compte d'interactions de contact avec le robot. L'évitement d'obstacles dépend de la géométrie de l'environnement et est commun à toutes les tâches robotiques.

De plus aucune hypothèse simplificatrice n'est faite sur la géométrie du robot ou sur celle de l'environnement.

Les critères à satisfaire pendant la résolution du problème de planification concernent le fait qu'une solution doit optimiser une fonction de coût exprimée en terme de la distance parcourue par le robot entre les deux configurations extrémités, de la durée ou de l'énergie nécessaires à l'exécution de son mouvement.

D'autres critères peuvent être également considérés tels que la prise en compte de distance de sécurité aux obstacles pour un robot mobile ou manipulateur ou encore la "qualité" et la stabilité des prises pour une main articulée. Face à la nature aussi diverse de ces aspects et aux difficultés qu'elle peut induire sur un processus de résolution, la plupart des travaux proposés dans le domaine de la planification de mouvement ont porté sur la considération de certaines instances du problème général. Nous présentons dans la suite de ce chapitre les principaux concepts et approches développés à cet effet. Vu le type de problématique abordé dans ce mémoire, nous nous intéresserons plus particulièrement aux instances et travaux abordant l'évitement d'obstacles sans la prise en compte de contraintes cinématiques et dynamiques.

Ainsi planifier une trajectoire pour un corps solide dans l'espace cartésien revient à planifier la trajectoire d'un point dans l'espace de travail[Pérez, 1983]. Si l'on considère maintenant les obstacles de l'environnement, on se rend compte que les "plonger" dans l'espace des configurations du système n'est pas une tâche aisée, surtout si le système comporte de nombreuses variables de configuration et que par conséquent la dimension de l'espace est élevée. Après ce bref exposé de la problématique de la planification de trajectoire, plusieurs questions restent en suspens :

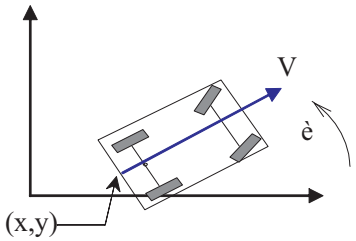


FIG. 1.1 – Cinématique d'un robot de type voiture.Un tel système est repéré dans le plan par 2 variable x et y donnant sa position et une variable θ donnant son orientation.L'espace de configuration est donc de dimension 3 .

1. Comment exprimer les obstacles dans l'espace des configurations ?
2. Comment trouver un chemin dans l'espace libre ?

1.2 Synthèse des principaux travaux en planification de mouvements

1.2.1 Introduction

Historiquement, le problème de planification a été initialement abordé dans le cadre d'un système de robots se déplaçant dans un environnement contenant des obstacles fixes et soumis à la seule contrainte de non-collision. Cette instance du problème général de planification est connue sous le nom du *paradigme du déménageur de piano* et sa résolution a fait l'objet de plusieurs travaux dont nous présentons une brève synthèse dans la suite de cette section. La plupart de ces travaux sont basés sur le concept d'espace des configurations du robot introduit par Lozano-Perez[Pérez, 1983]. Par configuration est désignée tout n-uplet de paramètres indépendants de position /orientation caractérisant d'une manière unique le robot dans son environnement ou espace de travail. Par ce concept, le problème de planification est formulé dans un espace CS_A de dimension n, l'espace des configurations, où le robot est assimilé à un point et les obstacles à des sous-ensembles également de dimension n. Ainsi, un mouvement solution est donné par une courbe dans CS_A correspondant à une séquence continue de configurations sans collision reliant les configurations initiale et finale considérées. Selon les cas, la solution à générer est contrainte à rester entièrement ou partiellement sans contact avec les obstacles. On parlera alors d'un mouvement dans l'espace des configurations libres ou au contact. De par sa généralité, le concept d'espace des configurations a permis d'apporter une réponse quant à l'existence ou non de chemins solutions au problème de planification et de caractériser cela par l'existence d'une composante connexe dans CS_A.

1.2.2 Méthodes par décomposition cellulaire

Une première approche pour la planification de chemin est connue sous le nom d'approche par décomposition cellulaire. Elle consiste à partitioner l'espace des configurations (ou de travail) libres du robot en un ensemble de régions connexes adjacentes. La description de la décomposition obtenue est alors capturée dans un graphe de connectivité dont les noeuds correspondent aux différentes régions (ou cellules) et les arcs aux relations d'adjacence entre elles. Le problème de planifier un mouvement entre deux configurations situées initialement dans deux cellules différentes est résolu en deux étapes :

a. exploration du graphe de connectivité et détermination d'un chemin reliant les cellules contenant les deux configurations initiales du graphe ;
b. recherche de la solution au problème de planification à partir de l'enveloppe définie par la liste de cellules adjacentes trouvée en a.

De par sa généralité, l'approche par décomposition a été adoptée dans plusieurs travaux et a conduit à la mise en oeuvre de nombreuses méthodes. Celles-ci sont généralement classées en deux catégories et se distinguent par les modèles de décomposition qu'elles utilisent et leur complétude quant à la résolution du problème de planification.

Méthodes exactes

La première catégorie regroupe les méthodes de résolution dites exactes en ce sens que la décomposition effectuée se base sur un recouvrement exact de l'espace libre du robot de ses contacts en cellules convexes. La complétude et la capacité de ce type de méthode à apporter une réponse quant à la résolution du problème général de planification ont été démontrées par Schwartz et Sharir[J.T.Schwartz and M.sharir, 1983]. Le problème de planification est formulé sous forme algébrique en considérant une décomposition de l'espace libre du robot sous forme de composantes cylindriques semi-algébriques (cellules de Collins)[P.Tournassoud, 1988]. Par cette formulation, les auteurs aboutissent à l'un des résultats importants sur la complexité de la planification, à savoir qu'elle est polynômial en la complexité de l'environnement et doublement exponentiels en la dimension de l'espace des configurations. La complexité de l'environnement est mesurée par le nombre et le degré des polynômes utilisés pour la description semi-algébrique de l'espace libre. A ce point une remarque s'impose, car le mot "cellule" peut prêter à confusion. Il est important en effet de noter que cette décomposition de l'espace libre est une méthode dite exacte, qui doit être distinguée des méthodes non-exactes. Au premier rang de ces dernières, on peut citer la discrétisation de l'espace cartésien en cellules cubiques de tailles données, qui produit des solutions pour une résolution donnée.

Méthodes approchées

Afin de remédier à cette complexité, des méthodes dites approchées sont généralement appliquées. Elles se distinguent des précédentes par la simplicité de la structure des cellules utilisées pour la décomposition et l'approximation faite de l'espace libre du robot. La structure de celui-ci est généralement capturée dans une représentation hiérarchique en cellules élémentaires identiques et permettant l'adaptation de la taille de celles-ci à la géométrie des zones à recouvrir. Plusieurs types de cellularisation de l'espace du robot sont généralement utilisés en pratique : représentation en octrees, découpage en tranches, ou encore en polyèdres[P.Tournassoud, 1988]. En comparant les deux approches de décomposition, il ressort que les méthodes exactes sont plus complètes en théorie que les méthodes approchées. Toutefois, la complexité de leur mise en oeuvre, les rend moins appropriées en pratique que les techniques approchées même si celles-ci restent parfois limitées à des espaces de dimension réduite. Cette constatation n'est pas générale puisqu'il apparaît dans[3] qu'une méthode exacte peut conduire dans le cas plan et pour une modélisation polygonale du robot et de son environnement à une implantation efficace.

1.2.3 Méthodes de résolution de type rétraction

Une seconde grande approche pour la résolution du problème de planification consiste à ramener la recherche du mouvement du robot dans un espace de plus faible dimension que celle de l'espace admissible. Cela consiste à représenter la connectivité de l'espace libre du robot par un réseau de courbes unidimensionnelles pouvant être entièrement dans l'espace libre ou des contacts du robot. La planification de mouvement entre deux configurations données est alors résolue en trois étapes : détermination d'un chemin ramenant le robot de sa configuration initiale à un point situé sur l'une des courbes du réseau de connectivité, détermination d'un chemin entre la configuration finale et le réseau, et enfin exploration de celui-ci afin d'en extraire un chemin reliant les deux points connectés aux configurations initiale et finale. Une première méthode basée sur ce concept a été proposée par Nelsson pour un espace de travail bidimensionnel encombré d'obstacles polygonaux[N.J.Nelsson, 1969]. Dans cette méthode, le réseau est décrit par un graphe, dit graphe de visibilité, où les noeuds correspondent aux sommets des obstacles et les arêtes à des segments de droite reliant ces sommets dans l'espace libre. L'inconvénient de cette méthode est que les solutions obtenues conduisent le robot à être en contact avec les obstacles en certains de leurs points (les sommets) même en présence d'une solution entièrement contenue dans l'espace libre. Afin d'y remédier, une seconde méthode, connue sous le nom de rétraction et développée par Ô'Dùnlaing et Yap [O'Dùnlaing and N.Yap, 1982], consiste à utiliser un diagramme de Voronoi[P.Tournassoud, 1992] pour capturer la connectivité de l'espace libre et générer des solutions éloignées le plus des obstacles. Une telle méthode, bien qu'elle soit générale en théorie, reste limitée à des espaces de dimension peu élevée. Une autre variante de cette méthode est développée par Brooks pour des espaces de travail polygonaux et se base sur la représentation de ceux-ci par des cylindres généralisés. Les chemins à suivre par le robot sont alors déterminés en considérant la connectivité entre les axes de ces cylindres. Enfin, une dernière méthode de type rétraction a été présentée par Canny dans le cadre de la résolution du problème général de la planification[P.Tournassoud, 1992]. L'algorithme proposé est exponentiel en la dimension de l'espace des configurations, et constitue l'un des résultats les plus importants sur la complexité du problème de planification de mouvement.

1.2.4 Autres méthodes de planification

Les méthodes exposées dans les paragraphes précédents sont généralement dites globales sachant qu'elles se basent toutes sur la structuration de l'espace des configurations et la modélisation a priori de sa connectivité. Afin d'éviter la complexité d'une telle étape de structuration, d'autres méthodes dites locales sont généralement utilisées. Leur principe consiste à déterminer les déplacements du robot en ne considérant qu'une représentation locale de l'environnement et à percevoir la planification de mouvement comme un problème d'optimisation.

Méthode du potentiel

Une première méthode, largement utilisée dans la littérature, consiste à assimiler le robot à une particule contrainte à se déplacer dans un champ de potentiel fictif obtenu par la composition d'un premier champ attractif au but et d'un ensemble de champs répulsifs modélisant la présence d'obstacles dans l'espace du robot. Les déplacements de celui-ci sont alors calculés itérativement par un algorithme de descente du gradient du potentiel obtenu. Un tel concept a été initialement introduit par Khatib et utilisé pour la commande et l'évitement d'obstacles pour un bras manipulateur[O.Khatib, 1986], et étendu par la suite pour la planification de mouvement. Si une méthode de type potentiel peut être facilement implantée et appliquée en temps réel pour des tâches de manipulation ou de navigation d'un robot mobile, elle reste sensible à l'occurrence de minimums locaux engendrant des configurations de blocage ou d'oscillation. Ces minimums sont généralement liés à la géométrie et à la répartition des obstacles dans l'espace de travail du robot et surtout aux coefficients de pénalités qui leur sont associés pendant la construction du champ de potentiel. Afin de remédier à ces problèmes, Barraquand et Latombe proposent dans[J-C.Latombe, 1991] d'intégrer à la minimisation du potentiel appliqué au robot l'exploration d'une représentation hiérarchique en bitmap de l'espace des configurations. L'évitement ou plutôt la sortie d'un minimum local est effectuée par l'application de mouvements aléatoires, dits browniens.

Dans le cadre d'une approche par champs de potentiel, un autre type de méthode se base sur le calcul variationnel pour la minimisation d'une fonctionnelle en considérant l'ensemble des chemins admissibles. Contrairement à la première méthode où le robot est réduit à un point, les méthodes basées sur le calcul variationnel considèrent que la variable de base est donnée par une courbe déformable reliant les deux configurations extrémités. L'évolution de la courbe est donnée par la résolution d'un système de n équations correspondant à la discrétisation spatiale de la courbe en n points. La fonctionnelle à minimiser est généralement composée de divers termes dont un potentiel de non-collision aux obstacles et des critères de continuité et de différentiabilité de la courbe. Enfin, la méthode peut fournir des solutions locales au problème de minimisation de la fonctionnelle considérée est sensible à la position de la courbe choisie initialement avant le début de la recherche. Ce problème présente une méthode combinant le calcul variationnel et une technique de programmation dynamique pour la recherche de chemins maintenant une distance de sécurité par rapport aux obstacles.

Méthode des contraintes

Dans le cadre d'une approche locale, Faverjon et Tournassoud proposent une méthode, dite des contraintes, qui aborde le problème de l'évitement d'obstacles en modélisant localement chacun de ceux-ci par l'ensemble de ses plans tangents [B.Faverjon and P.Tournassoud, 1987]. La génération des mouvements du robot est obtenue par la minimisation d'un critère quadratique sur la tâche à effectuer en présence de contraintes linéaires associées aux équations des plans tangents aux obstacles. A la différence de la méthode du potentiel, les obstacles n'agissent sur le robot pendant le processus de minimisation que quand il en est très proche et a tendance à y pénétrer. Toutefois, la méthode reste sensible à la présence de minimums locaux. Ce problème est abordé dans[B.Faverjon and P.Tournassoud, 1987] en combinant la méthode locale avec une technique d'apprentissage.

1.3 La théorie des graphes et chemins optimaux

1.3.1 Intérêt des graphes

Les graphes représentent un instrument puissant pour modéliser de nombreux problèmes combinatoires, qui seraient sans cela difficilement abordables par des techniques classiques comme l'analyse mathématiques.

En plus de son existence en tant qu'objet mathématique, le graphe est aussi une structure de données puissante pour l'informatique.
Les graphes sont irremplaçables dés qu'il s'agit de décrire la structure d'un ensemble complexe, en exprimant les relations, les dépendances entre ces éléments. Des exemples sont les diagrammes hiérarchique en sociologie, les arbres génétiques, et les diagrammes de succession de tâches en gestion de projet[P. Lacomme, 2003].
Un autre grand groupe d'utilisation est la représentation de connexion et de possibilités de cheminement : détermination de la connexité d'un réseau quelconque (électrique, d'adduction d'eau), calcul de plus court chemin dans un réseau routier, Les graphes sont enfin précieux pour décrire des systèmes dynamiques, c'est-à-dire qui évoluent dans le temps : diagrammes de transition, automates d'état finis, chaînes de Markov.
Il existe deux familles des graphes : orientés et non orientés c'est-à-dire que les relations entre les éléments d'un ensemble sont orientés ou non.
Un graphe est défini par un couple $\mathcal{G} = (\mathcal{X},\mathcal{U})$ de deux ensembles :

- $\mathcal{X}$ est un ensemble $(x_1, x_2, \dots x_N)$ de sommets, également appelés noeuds ;
- $\mathcal{U} = (u_1, u_2, \dots u_M)$ est une famille de couples ordonnés de sommets appelés arêtes (arcs).

Un graphe peut être valué, ou pondéré, il est alors muni de poids ou coût sur ses arêtes .

1.3.2 Structure de tas

Considérons un ensemble $\mathcal{H}$, dont tout élément x, possède une valeur numérique $W[x]$, et dans lequel les prélèvements ne concernent que le plus petit élément. Une solution simple est de chercher avant tout prélèvement quel est le petit élément, ce qui coûte $\mathcal{O}(k)$ si $\mathcal{H}$ contient k éléments. Le **tas** est une structure de données puissante qui permet de réaliser ces prélèvements de minima et l'ajout d'éléments quelconques en seulement $\mathcal{O}(\log k)$. Nous l'utiliserons pour accélérer des algorithmes de calcul de plus court chemin dans un graphe.
La structure de **tas (binaire)**[P. Lacomme, 2003] est un objet tabulé qui peut être vu comme arbre binaire presque complet. Chaque noeud de l'arbre correspond à un élément du tableau qui contient la valeur du noeud. L'arbre est complètement rempli sur tous les niveaux, sauf parfois le plus bas qui est rempli en partant de la gauche, jusqu'à un certain point. Un tableau **A** qui représente un **tas** est un objet avec deux attributs : *longueur*[**A**], qui est le nombre d'éléments du tableau, et *taille*[**A**], qui est le nombre d'éléments du tas rangés dans le tableau **A**. Autrement dit, bien que **A**[1..*longueur*[**A**]] puisse contenir des nombres valides, aucun élément après **A**[*taille*[**A**]], où *taille*[**A**] $\leqslant$ *longueur*[**A**], est un élément du tas. La racine de l'arbre est **A**[1], et sachant l'indice i d'un noeud, les indices de son père *Père(i)*, de son fils gauche *Gauche(i)*, et de son fils droit *Droit(i)* peuvent être facilement calculés.
L'algorithme suivant produit un tas à partir d'un tableau non-ordonné.

```
Entrée(A)
Sortie(Tas(A))
begin
    taille[A] :=longueur[A]
    for i ← ⌊ longueur[A]/2⌋ à 1 do
        l ←Gauche(i) ; r ←Droit(i) ;
        if l ⩽ taille[A] et A[l]>A[i] then
            max ← l ;
            else max ← i ;
            if r ⩽taille[A] et A[r]>A[max] then
            max ← l ;
            if max ≠ i then échangerA[i]↔ A[max] ;
            i ← max
        end
    end
    return Tas(A)
end
```

Algorithm 1: Construction d'un tas à partir d'un tableau non-ordonné.

1.3.3 Les problèmes de chemins optimaux

Les problèmes de chemins optimaux sont très fréquents dans les applications pratiques. On les rencontre dés qu'il s'agit d'acheminer un objet entre deux points d'un réseaux, de façon à minimiser un coût, une distance ou une durée. Ils apparaissent aussi en sous-problèmes combinatoires, notamment les flots dans les graphes et les ordonnancements. Tout ceci a motivé très tôt la recherche d'algorithmes efficaces.

1.3.4 Les grands types de problèmes

Considérons un graphe valué $\mathcal{G} = (\mathcal{X}, \mathcal{A}, \mathcal{W})$. $\mathcal{X}$ désigne un ensemble de $\mathcal{N}$sommets (ou noeuds) et $\mathcal{A}$ un ensemble de $\mathcal{M}$ arêtes. $\mathcal{W}$(i,j) aussi noté $\mathcal{W}_{i,j}$, est la valuation (aussi appelée *poids* ou *coût*) de l'arête (i,j), par exemple une distance, un coût de transport, ou temps de parcours. Pour la fonction économique la plus répandue, le *coût d'un chemin* entre deux sommets est la somme des coûts de ces arêtes. Les problèmes associés consistent à calculer des *chemins de coût minimal* (en abrégé chemins minimaux, ou plus courts chemins). Ils ont un sens si $\mathcal{G}$ n'a pas un circuit négatif, sinon on pourrait diminuer infiniment le coût d'un chemin en tournant dans un tel circuit, appelé pour cette raison *circuit absorbant*[P.Chrétienne, 1994]. Bien entendu on peut aussi chercher des chemins de valeurs maximales. En l'absence de circuit absorbant, on peut restreindre la recherche des plus courts chemins aux seuls *chemins élémentaires*, c'est-à-dire ne passant pas deux fois par un même sommet. En effet, si un chemin emprunte un circuit, on peut enlever la portion passant par le circuit sans augmenter le coût. Le problème de recherche d'un chemin optimal en présence de circuits absorbants existe, mais il est *NP*-difficile. Il existe d'autres fonctions économiques que la somme des coûts des arêtes, mais elles sont moins répandues dans les applications.
La littérature distingue trois types de problème, que nous notons A,B et C :

A : **Plus court chemin à origine unique** étant donné un sommet de départ s. Trouver un plus court chemin de s vers tout autre sommet.

B : **Plus court chemin pour un couple de sommets** étant donné deux sommets s et t. Trouver le plus court chemin de s à t.

C : **Plus court chemin pour tout couple de sommets** trouver un plus court chemin entre tout couple de sommets, c'est-à-dire calculer une matrice $N \times N$ appelée *Distancier*.

1.4 Algorithmes de recherche du plus court chemin

1.4.1 Les deux familles d'algorithmes

La plupart des algorithmes de recherche de plus court chemin calculent pour chaque sommet x une **étiquette** $V[x]$, valeur des plus courts chemins du sommet de départ au sommet x. Cette valeur représente au début une estimation par excès (majorant) de la valeur des plus courts chemins.
Certains algorithmes traitent définitivement un sommet à chaque itération : ils sélectionnent un sommet x et calculent la valeur définitive de $V[x]$. Ces algorithmes sont dits à **fixation d'étiquettes**[P. Lacomme, 2003] et sont représentés par l'algorithme de *Dijkstra* et ses dérivés. D'autres algorithmes peuvent affiner jusqu'à la dernière iteration l'étiquette de chaque sommet. On les appelle algorithmes à **correction d'étiquettes**[P. Lacomme, 2003].L'algorithme de *Bellman* est un algorithme à correction d'étiquettes très connu, de type programmation dynamique

1.5 Algorithme à fixation d'étiquettes

1.5.1 Algorithme de Dijkstra

L'algorithme de Dijkstra[P.Chrétienne, 1994] n'est valable que pour les graphes à valuation positive ou nulle, qui ne contiennent pas de circuits négatifs.A chaque itération, un sommet x reçoit son étiquette définitive, on dit qu'il est fixé.
L'itération principale sélectionne le sommet x d'étiquette minimale parmi ceux déjà atteints par un chemin provisoire d'origine s. Pour tout successeur y de x, on regarde si le chemin passant par x améliore le chemin déjà trouvé de s à y : si oui on remplace $V[y]$ par $\min(V[y], V[x] + W(x, y))$ et on mémorise qu'on parvient en y via x en posant $P[y] := x$.
Si tout les sommets sont accessibles au départ de s, l'algorithme se déroule en N itérations. En pratique, des sommets peuvent ne pas être accessibles. Si on veut calculer seulement un plus court chemin de s vers un

autre sommet t (problème B), il suffit d'arrêter l'algorithme dés que t est fermé, par exemple en modifiant le test de fin en "jusqu'à ($VMin = +\infty$) ou $x = t$".

```
Entrée(G, list(Adj[G]), s)
Sortie(ℓ)
begin
    R ensemble de tous les noeuds ;
    Initialiser le tableau V à +∞ ;
    Initialiser le tableau P à 0 ;
    V[s] := 0
    P := s while R ≠ ∅ do
        u :=Extraire-Min(V) ;
        P := P ∪ u ;
        R := R − u ;
        for All Noeud v voisin de u do
            if V[v] > V[u] + W(u, v) then
            V[v] := V[u] + W(u, v) ;
        end
    end
    return (ℓ)
end
```

Algorithm 2: *DIJKSTRA*

L'inconvénient majeur de l'algorithme de Dijkstra est qu'il est insensible à la densité du graphe. Le nombre d'itération de la boucle **While**, au plus N, ne peut pas être amélioré par construction de l'algorithme. En revanche l'essentiel du travail est dû à la boucle interne trouvant le prochain sommet i à fixer. Cette boucle coûte $\mathcal{O}(N)$. Ceci suggère d'utiliser un tas pour avoir x plus rapidement.
Dans le cas de graphes connexes, les plus courants en pratique, on a $M \geqslant N - 1$. L'algorithme de Dijkstra avec tas est alors en $O(M.\log N)$ et il est donc avantageux pour les graphes peu denses. Si tous les arcs possibles existent ($M = N^2$), il est donc aussi coûteux que la version sans tas.

1.5.2 Aperçu sur d'autres algorithmes

Algorithme de Sedgewick et Vitter

Cet algorithme a été conçu pour les graphes non orientés dont les sommets sont des points d'un espace euclidien, et les arrêtes des segments entre les points valués par la longueur euclidienne de segment (*distances entre les points*). Il est conçu pour le problème A, c'est-à-dire le calcule d'un plus court chemin entre deux sommets s et t. Sa structure générale est similaire à celle de l'algorithme de Dijkstra.

Algorithmes à buckets

Les algorithmes à buckets sont des variantes de l'algorithme de Dijkstra intéressantes quand les coûts des arcs sont entiers et leurs maximum U n'est pas trop grand. On partitionne l'intervalle des valeurs des étiquettes en B intervalles de largeur commune L, numérotés de 0 à $B - 1$, et on associe à chacun d'eux un ensemble de sommets appelé bucket. Ce système est codable comme un tableau de B listes. Pour trouver le sommet x à fixer dans l'algorithme de Dijkstra à buckets, on cherche d'abord le bucket non vide de plus petit indice k. On balaie ensuite ce dernier pour localiser et extraire le sommet x d'étiquette minimale. Pour chaque successeur y dont on peut améliorer l'étiquette : on cherche le bucket de y, on le balaie pour localiser et enlever y, on modifie l'étiquette de y avec ($V[y] := V[x] + W(x, y)$), on localise le nouveau bucket de y et enfin, on insère y en tête de ce bucket. Les sommets en pratique sont bien répartis et les listes buckets sont courtes, ce qui donne de très bonnes performances moyennes.

1.6 Algorithme à correction d'étiquettes

1.6.1 Algorithme de Bellman

Cet algorithme à correction d'etiquettes a été conçu dans les années 50[P.Chrétienne, 1994] par Bellman et Moore. Il est prévu pour des valuations quelconques et peut être adapté pour détecter un circuit de coût négatif. Il s'agit d'une méthode de programmation dynamique, c'est-à-dire d'optimisation récursive, décrite par la relation suivante :

$$\begin{cases} V_0(s) = 0 \\ V_0(y) = +\infty, y \neq s \\ V_k(y) = \min_{x \in \Gamma^{-1}(y)} \{V_{k-1}(x) + W(x,y)\}, k > 0 \end{cases}$$

$V_k(x)$ désigne la valeur des plus courts chemins d'au plus k arcs entre le sommet s et le sommet x. Les deux premières relations servent à stopper la récursion. Le sommet s peut être considéré comme un chemin de 0 arc et de coût nul. La troisième relation signifie qu'un chemin optimal de k arcs de s à y s'obtient à partir des chemins optimaux de $k-1$ arcs de s vers tout prédécesseur x de y. En effet, tout chemin optimal est formé de portions optimales, sinon on pourrait améliorer le chemin tout entier en remplaçant une portion non optimale par une portion plus courte.
La formulation récursive étant peu efficace, on calcule en pratique le tableau V itérativement, pour les valeurs croissantes de k. Nous donnons ci-après un algorithme simple. Les étiquettes en fin d'étape k sont calculées dans un nouveau tableau à partir du V des étiquettes disponibles en début d'étape. Pour tout sommet y, on regarde si $V[y]$ est améliorable en venant d'un prédécesseur de y. En fin d'étape on écrase V par le nouveau tableau et on passe à l'étape suivante. En l'absence de circuit absorbant, on peut se restreindre aux chemin élémentaires pour trouver un plus court chemin de s vers tout autre sommet. Or, un tel chemin n'a pas plus de $N-1$ arcs. Les étiquettes sont donc stabilisées en au plus $N-1$ itérations. En pratique, elles peuvent se stabiliser plus tôt, et un meilleur test de fin est quand $V_k = V_{k-1}$. La complexité est en $\mathcal{O}(N.M)$: il y a au plus $N-1$ itérations principales, consistant à consulter les prédécesseurs de tous les sommets, c'est-à-dire les M arcs.

```
Entrée(G, s)
Sortie(ℓ)
begin
    R ensemble de tous les noeuds ;
    Initialiser le tableau V à +∞ ;
    V[s] := 0
    for i=1 jusqu'à Nombre de sommets de G do
        for All lien (u, v) du G do
            if V[v] > V[u] + W(u, v) then
                V[v] := V[u] + W(u, v) ;
            end
        end
    end
    for All lien (u, v) du G do
        if V[v] > V[u] + W(u, v) then
            return Faux
        end
    end
    return (ℓ = P)
end
```

Algorithm 3: *BELLMAN*

1.6.2 Aperçu sur d'autres algorithmes

Algorithme FIFO

Cet algorithme simple à correction d'étiquettes examine les sommets dans l'ordre FIFO grace à une file $\mathcal{Q}$ de sommets. Au début, seul s est dans $\mathcal{Q}$. Une iteration principale traite tous les sommets présents dans $\mathcal{Q}$ au début de l'itération. Ensuite l'algorithme balaye les successeurs des sommets et les placé, ceux dont l'étiquettes améliorées, à la fin de $\mathcal{Q}$. L'algorithme se termine quand $\mathcal{Q}$ est vide. En fait, l'algorithme FIFO est un dérivé de l'algorithme de Bellman, très intéressant car n'utilisant pas les prédécesseurs.

Algorithme de D'Esopo et Pape

Cet algorithme utilise une pile-file Next, il s'agit d'une file où on peut ajouter un élément à la fin (En Queue) ou en tête (Push). Comme dans FIFO, un sommet atteint la première fois et mis en fin de file. Par contre si on le revisite, on l'insère en tête de file (à condition qu'il ne soit pas déjà en file). Ce critère heuristique de gestion peut s'expliquer intuitivement. Quand on atteint pour la fois un sommet, il n'est pas urgent de développer ses successeurs car les chemins obtenus risquent d'être mauvais dans un premier temps. En revanche un sommet déjà visité et dont l'étiquette vient de diminuer doit être développée en priorité pour propager l'amélioration.

Algorithme de Floyd

L'algorithme de Floyd calcule un distancier $N \times N$ donnant les valeurs des plus courts chemins entre tout couple de sommets (problème C). Pour cet algorithme le tableau V des étiquettes devient une matrice $N \times N$, $V[i, j]$ désignant le coût des plus courts chemins de i à j.

1.7 Conclusion

La plupart des algorithmes vu précédemment consistent à balayer un espace de recherche (contenant une infinité non dénombrable de points) dans le but de trouver une solution optimale. Une approche ponctuelle ne permet d'analyser qu'une infime portion de cet espace de recherche. Ce dernier peut fréquemment être recouvert par un nombre fini de sous ensembles simples, sur lesquels on sait calculer, et qui eux, contiennent une infinité de points.

Chapitre 2

Analyse par intervalles

2.1 Historique succinct et subjectif

La naissance de l'arithmétique par intervalles n'est pas datée avec certitude, pour une large de communauté son père est Ramon Moore qui l'a mentionnée pour la première fois en 1962 dans [R.Moore, 1962] et pour la deuxième fois en 1966[R.E.Moore, 1966] dans. Pour d'autres on trouve déjà dans un article de T. Sunaga daté de 1958 les fondements de l'arithmétique par intervalles. L'origine de l'arithmétique par intervalles est attribuée à Rosalind Cecil Young qui l'aurait proposée dans sa thèse de doctorat à l'Université de Cambridge en 1931. Il est très facile de se procurer ces références, grâce au site `www.cs.utep.edu$/$interval-comp` rubrique Early papers.

Que l'arithmétique par intervalles soit née en 1931, en 1958 ou en 1962, son enfance se prolonge jusqu'à la fin des années 1970 : à ses débuts en effet, cette arithmétique semble rester assez confidentielle.

La première raison est que le calcul scientifique ne dispose pas encore d'assez de puissance pour pouvoir se permettre de perdre un facteur de vitesse d'au moins 4 et de mémoire d'au moins 2 en changeant d'arithmétique. La seconde est que l'arithmétique flottante n'est pas encore assez bien spécifiée (arrondis en particulier) pour qu'implanter l'arithmétique par intervalles à partir de l'arithmétique flottante disponible alors soit une tâche aisée.

A partir des années 1980, l'arithmétique par intervalles prend son essor, en Allemagne particulièrement, sous l'impulsion d'U. Kulisch à Karlsruhe. Il réussit à convaincre Nixdorf de développer un processeur spécifique, puis IBM de mettre au point un jeu d'instructions et un compilateur intégrant l'arithmétique par intervalles. Il va également former un grand nombre d'étudiants qui ont maintenant essaimé dans toute l'Allemagne et contribuent à la maintenir dans le peloton de tête des pays intervallistes. A la même époque, l'arithmétique flottante voit naître la norme IEEE-754 qui spécifie en particulier les modes d'arrondis et facilite l'implantation de l'arithmétique par intervalles. Cependant, l'arithmétique par intervalles peine à convaincre des utilisateurs potentiels. Il semble qu'elle n'a pas répondu à des espoirs, à vrai dire injustifiés, quant à son utilisation comme outil de validation numérique de calculs flottants. En effet, tout calcul par intervalle est un calcul garanti, c'est-à-dire que le résultat calculé est un intervalle garanti contenir la valeur ou l'ensemble de valeurs cherché. On s'imaginait alors qu'il suffisait de remplacer le type float ou double (ou real...) par le type interval dans un programme pour obtenir un résultat donnant un encadrement fin des erreurs d'arrondi, ce n'est pas le cas comme nous le verrons cet échec a desservi l'arithmétique par intervalles à ces débuts.

L'arithmétique par intervalle continue à se développer, mais avec des objectifs différents depuis le début des années 1990. Son atout majeur, qui est de calculer sur des ensembles, est désormais mis à profit, c'est par exemple le seul outil déterministe permettant de déterminer l'optimum global d'une fonction continue, de déterminer tous les zéros d'une fonction et l'image directe ou inverse d'un ensemble par fonction[L.Jaulin and E.Walter,].

2.2 Calcul par intervalles

2.2.1 Intervalles

En arithmétique par intervalles, on ne manipule plus des nombres, qui approchent plus ou moins fidèlement une valeur, mais des intervalles contenant cette valeur. Par exemple, on peut tenir compte d'une erreur de mesure en remplaçant une valeur mesurée x avec une incertitude ε par l'intervalle $[x-\varepsilon, x+\varepsilon]$. On peut également remplacer une valeur non exactement représentable, telle que π, par l'intervalle $[3.14, 3.15]$. En effet, l'objectif de l'arithmétique par intervalles est de fournir des résultats qui contiennent à coup sûr la valeur ou l'ensemble cherché ; on parle alors de résultats garantis ou validés, ou encore certifiés.

2.2.2 Calculs

Définition 1 *Appliquer aux intervalles les opérations arithmétiques de base $\{\times, \div, +, -\}$ est une idée naturelle pour évaluer l'ensemble que décrit la somme, le produit,...etc, de deux réels incertains inclus dans des intervalles.*

On définit ainsi pour $\circ \in \{\times, \div, +, -\}$,

$$[x] \circ [y] = \{x \circ y | x \in [x] \ et \ y \in [y]\}.$$

La caractérisation de $[x] \circ [y]$ se fait à l'aide du formulaire

$$\begin{cases} [x] + [y] = [\underline{x} + \underline{y}, \bar{x}, \bar{y}]; \\ [x] - [y] = [\underline{x} - \bar{y}, \bar{x}, \underline{y}]; \\ [x] \times [y] = [\min(\underline{x}.\underline{y}, \bar{x}.\underline{y}, \underline{x}.\bar{y}, \bar{x}.\bar{y}), max(\underline{x}.\underline{y}, \bar{x}.\underline{y}, \underline{x}.\bar{y}, \bar{x}.\bar{y})]; \\ [x]/[y] = [x][1/\bar{y}, 1/\underline{y}]; \end{cases} \tag{2.1}$$

Les propriétés algébriques

On peut d'ores et déjà constater que les opérations définies ci-dessus ne présentent pas les propriétés algébriques de leurs contreparties ponctuelles. Tout d'abord la soustraction n'est pas la réciproque de l'addition. Par exemple, si

$$X = [2,3], \quad X - X = [2,3] - [2,3] = [-1,1] \neq 0$$

même s'il le contient. En effet,

$$X - X = \{x - y | x \in X, y \in X\} \supset \{x - x | x \in X\} = \{0\}$$

et l'inclusion est stricte.
De même façon la division n'est pas la réciproque de la multiplication, si :
$X = [2,3]$, l'intervalle $X/X = [2,3]/[2,3] = [2/3, 3/2]$ n'est pas égale à 1 même s'il le contient.
De plus la multiplication d'un intervalle par lui-même n'est pas égale à l'élévation au carré, si

$$X = [-3,2] X \times X = [-3,2] \times [-3,2] = [-6,9],$$

alors

$$X^2 = \{x^2 | x \in X\} = [0,9]$$

Enfin la multiplication n'est pas distributive par rapport à l'addition :
si $X = [-2,3]$; $Y = [1,4]$; $Z = [-2,1]$.

$$X \times (Y + Z) = [-2,3] \times ([1,4][-2,1]) = [-2,3] \times [-1,5] = [-10,15].$$

$$X \times Y + X \times Z = [-2,3] \times [1,4] + [-2,3][-2,1] = [-8,12] + [-6,4] = [-14,16].$$

Comme l'illustre cet exemple, la multiplication est sous-distributive par rapport à l'addition, c'est-à-dire

$$X \times (Y + Z) \subset X \times Y + X \times Z.$$

La raison en est toujours que dans le premier cas on détermine

$$\{p_x \times (p_y + p_z) | p_x \in X, p_y \in Y, p_z \in Z\}$$

alors que dans le second on calcule

$$\{p_x \times p_y + p_x \times p_z | p_x \in X\, p_{x'} \in X\ , p_y \in Y\, p_z \in Z\}$$

autrement dit on n'a pas identité de p_x et px'.

2.2.3 Fonctions élémentaires

On peut également définir des fonctions élémentaires (sin ,exp, acoth, ...) en prenant des intervalles pour argument, à l'aide de la définition 1 ci-dessus. Pour les fonctions monotones telles l'exponentielle ou l'arc-cotangente hyperbolique, on déduit facilement les formules permettant de les calculer :

$$\exp([\underline{x}, \bar{x}]) = [\exp \underline{x}, \exp \bar{x}], \text{ puisque exp est croissante.}$$

$$\text{acoth}([\underline{x}, \bar{x}]) = [\text{acoth}\underline{x}, \text{acoth}\bar{x}] \text{ puisque acoth est décroissante sur } \mathbb{R}^{+*} \text{ et } \mathbb{R}^{-*}\ .$$

En revanche il faut être plus soigneux pour les fonctions périodiques, mais il est possible d'établir des algorithmes de calculs pour ces fonctions, dés que l'on dispose de ces fonctions sur les réels.
Par exemple $\sin[\pi/3, \pi] = [0, 1]$. Enfin on ne sait définir les fonctions élémentaires que sur des intervalles inclus dans leur domaine de définition : on a vu ci-dessus que acoth n'était définie que pour des intervalles ne contenant pas 0, de la même manière, le logarithme ne sera défini que pour des intervalles strictement positifs.

2.3 Pavage et sous-pavage

2.3.1 Pavés

Un pavé $[\mathbf{x}]$ de $\mathbb{R}^n$ est le produit cartésien de n intervalles :

$$[\mathbf{x}] = [\underline{x}_1, \overline{x}_1] \times \ldots \times [\underline{x}_n, \overline{x}_n] = [x]_1 \times \ldots \times [x]_n \tag{2.2}$$

L'ensemble des pavés est dénoté $\mathbb{IR}^n$.
La longueur $w([\mathbf{x}])$ d'un pavé $[\mathbf{x}]$ est la longueur du plus grand de ses côtés.

2.3.2 sous-pavage

Un sous-pavage k de $\mathbb{R}^n$ est un ensemble (fini ou non) de pavés de $\mathbb{IR}^n$ de volume non nul qui ne se recouvert pas, ce qui signifie que leur intersection deux à deux est vide sauf éventuellement sur leurs frontières.

2.3.3 Fonctions d'inclusions

Soit f est une fonction élémentaire (cos, cosh, sin, exp, tan, arg, $\cdots$) définie sur un domaine $\mathcal{D} \subset \mathbb{R}$. La définition de cette fonction est étendue aux variables intervalles $[x] \subset \mathcal{D}$ en considérant l'évaluation intervalle de f sur $[x]$.

$$f([x]) = \{f(x) | x \in [x]\}. \tag{2.3}$$

C'est un ensemble qui contient toutes les valeurs prises par f sur [x]. Ce n'est évidemment pas toujours un intervalle, mais par construction, l'évaluation intervalle est toujours monotone pour l'inclusion :

$$[x] \subseteq [y] \Rightarrow f([x]) \subseteq f([y]). \tag{2.4}$$

Il est aisé de caractériser l'évaluation intervalle des fonctions monotones usuelles. Ainsi, par exemple :

$$\exp([x]) = [\exp(\underline{x}), \exp(\overline{x})]\,.$$

L'évaluation intervalle d'une fonction f plus générale est souvent beaucoup plus difficile. La recherche du minimum et du maximum de f sur un intervalle $[x]$ représente en effet un véritable problème d'optimisation en soi. Il est par contre en général facile de définir une nouvelle fonction *intervalle* , aisée à évaluer, dont les images contiendrons les évaluations intervalles de f.

Définition 2 *Soit $f : \mathcal{D} \subset \mathbb{R}$, une fonction ensembliste $f_{[]} : \mathbb{I}\mathcal{D} \to \mathbb{IR}$ est une fonction d'inclusion de f si*

$$\forall [x] \in \mathbb{I}\mathcal{D}, f([x]) \subset f_{[]}([x]). \tag{2.5}$$

Cette fonction d'inclusion sera dite convergente lorsque la propriété suivante est satisfaire :

$$si\ w([x]) \to 0\ alors\ w(f_{[]}([x])) \to 0. \tag{2.6}$$

Rappelons que $w([x])$ dénote la longueur de $[x]$.

Insistons sur le fait que $f_{[]}([x])$ est bien un intervalle. Les fonctions d'inclusions sont définies de la même manière pour des fonctions ayant pour valeur des vecteurs ou des matrices de réels. L'un des objectifs de l'analyse par intervalles est d'obtenir une fonction d'inclusion donnant des intervalles les plus proches possibles des évaluations intervalles fournies pour la fonction réelle correspondante. Lorsque l'inclusion (2.5) est une égalité, la fonction $f_{[]}$ est dite *minimale*.
Il existe plusieurs méthodes pour obtenir une fonction d'inclusion f. La plus simple est de remplacer les occurrences des variables scalaires $(x, y, \ldots)$ par les variables intervalles correspondantes $([x], [y], \ldots)$, une fonction d'inclusion *naturelle* de f est alors obtenue. Cependant, bien souvent, cette fonction d'inclusion n'est pas la meilleure (elle n'est minimale que lorsque chacun des variables n'intervient qu'une seule fois dans l'expression de f), en outre, la taille des intervalles obtenus dépend assez fortement de l'expression initiale de la fonction.
Une fonction d'inclusion qui fournit un intervalle image qui n'est pas égal à l'évaluation intervalle de fonction réelle sur l'intervalle considéré est dite *pessimiste*.
Il n'existe pas de méthode systématique permettant pour une fonction donnée d'obtenir une fonction d'inclusion minimale. Cependant, on peut constater que plus une variable apparaîtra fréquemment, plus le pessimisme des intervalles images sera important. Cela peut s'interpréter par le fait que deux occurrences d'une même variable sont considérées comme variant indépendamment l'une de l'autre. Ainsi, les incertitudes se combinent, pour donner un résultat plus mauvais que si l'on avait considéré les occurrences comme variant identiquement.
Il existe d'autres types de fonction d'inclusion, faisant intervenir des développements en séries de la fonction initiale. Ainsi, pour une fonction $f : \mathcal{D} \to \mathbb{R}$, une fois dérivable sur un intervalle $[x] \subset \mathcal{D}$, on sait que $\forall x, m \in [x], \exists \xi \in [x]$ tel que :

avec $m \in [x]$ et $f'_{[]}$ une fonction d'inclusion de la dérivée de f. Cette fonction est appelée *forme centrée standard* de f sur l'intervalle $[x]$, de centre m. Elle donne de bons résultats, son pessimisme est assez faible lorsque l'intervalle considéré a une faible longueur. pour évaluer la qualité d'une fonction d'inclusion, il est nécessaire d'évaluer la distance entre l'évaluation fournie par la fonction d'inclusion et l'évaluation intervalle de la fonction sur un intervalle test.

2.4 Inversion ensembliste

Définition 3 *Si $f : \mathbb{R}^n \to \mathbb{R}^m$ est une fonction continue et differentiable et $\mathbb{Y}$ un sous ensemble de $\mathbb{R}^m$. La théorie de l'inversion ensembliste permet de trouver l'ensemble image réciproque de $\mathbb{Y}$ par la fonction f. La formalisation mathématique de ceci s'écrit :*

$$\mathbb{X} = \{x \in \mathbb{R} | f(x) \in \mathbb{Y}\} = f^{-1}(\mathbb{Y}) \tag{2.7}$$

Résoudre un problème d'inversion ensembliste revient à caractériser le sous ensemble $\mathbb{X}$

2.4.1 Algorithmes d'inversion ensemblistes-SIVIA

L'algorithme d'inversion ensembliste de Moore et SIVIA de Jaulin permet de résoudre plusieurs problèmes ensemblistes. Un algorithme d'inversion ensembliste permet de réaliser une approximation de l'ensemble $\mathbb{X}$ par un sous-pavage. Si $\mathbf{f}$ est une fonction continue, et $\mathbb{Y}$ un sous ensemble alors :

$$\mathbb{X} = \mathbf{f}^{-1}(\mathbb{Y}). \tag{2.8}$$

Les algorithmes d'inversion ensembliste sont de type arborescent. Ils se composent essentiellement de trois étapes qui sont : le choix d'un pavé initial censé contenir les solutions recherchées, les tests d'inclusions, et la bisection.

Tests d'inclusion

Les différents tests réalisés à partir de la fonction d'inclusion (voir section 2.3.3) $f_{[]}([x])$. Pour un pavé initial $[x_0]$, trois cas sont possibles :

1. $f_{[]}([x_0]) \subset \mathbb{Y} \Rightarrow [x_0] \subset \mathbb{X}$ $[x_0]$ est acceptable.
2. $f_{[]}([x]) \cap \mathbb{Y} = \emptyset \Rightarrow [x_0] \cap \mathbb{X}$ $[x_0]$ est inacceptable.
3. $f_{[]}([x]) \cap \mathbb{Y} \neq \emptyset$ $[x_0]$ est incertain ou ambigu.

Un test d'inclusion associé au test $t(x)$ sera noté $[t]([x])$

Bissection

Si le pavé $[x_0]$ est incertain, alors nous réalisons une bissection sur celui-ci. Les deux pavés générés sont de nouveau testés, puis l'algorithme se propage tant qu'il reste un pavé incertain. Nous devons fixer une limite appelée précision noté ε, cette limite marque l'arrêt de l'algorithme. Ainsi les pavés acceptables et inacceptables sont les feuilles de l'arborescence de l'algorithme, les pavés ambigus en sont les noeuds (voir figure 2.1.

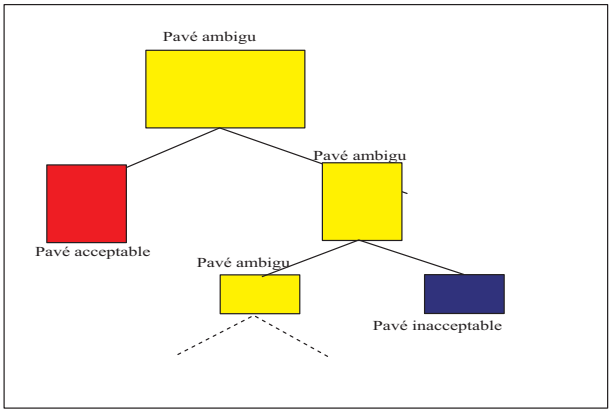


FIG. 2.1 – *Les pavés rouges et bleus forment les feuilles de l'arborescence de bissection d'un pavé ambigu (noeud).*

Algorithme SIVIA

Nous utilisons l'algorithme SIVIA pour caractériser le sous ensemble $\mathbb{X}$ avec une précision requise ε. Cet algorithme utilise une pile $\mathcal{P}$ servant à stocker les pavés intermédiaires et une pile auxiliaire $\mathcal{P}^0$ dans laquelle nous stockerons uniquement les pavés solutions ou les pavés de petite taille. Si nous choisissons le pavé $[x_0]$ tel que $\mathbb{P} = [x_0]$, celui-ci est alors initialement rangé dans la pile $\mathcal{P}$. Lors de la procédure de parcours $\mathcal{P}$, nous récupérons à chaque fois le pavé au sommet de la pile dans un pavé $[x]$, après la pile est vidée de l'élément du sommet. A partir de $[x]$, nous effectuons des tests sur la fonction d'inclusion $f_{[]}([x])$. Si le pavé $[x]$ est ambigu et la largeur $w([x])$ est supérieur à ε, nous procédons à une bissection de $[x]$. Les deux pavés générés $[x_1], [x_2]$ sont ensuite rangés au sommet de la pile $\mathcal{P}$. Sinon la pile $\mathcal{P}^0$ est augmentée de $[x]$.

```
3.1  Entrée(f, [x_0], 𝕐, ε)
3.2  Sortie(𝕏 = 𝒫)
3.3  begin
3.4      Initialisation : 𝒫^0 := ∅, 𝒫 := [x_0] ;
3.5      while (𝒫 ≠ ∅) do
3.6          [x] := ExtraireEntête(𝒫) ;
3.7          if (f_∏([x]) ⊂ 𝕐) then
3.8              𝒫^0 := 𝒫^0 ∪ [x], Aller à 3.5 ;
3.9          end
3.10         if (f_∏([x]) ∩ 𝕐 ≠ ∅) then
3.11             if (w([x]) < ε) then
3.12             𝒫^0 := 𝒫^0 ∪ [x], Aller à 3.5 ;
3.13             else
3.14             Bissection de [x] → ([x_1], [x_2]) ;
3.15             𝒫 := 𝒫 ∪ [x_1], 𝒫 := 𝒫 ∪ [x_1], Aller à 3.5 ;
3.16         end
3.17     end
3.18     Aller à 3.3
3.19 end
```

Algorithm 4: *SIVIA*

2.5 Conclusion

L'arithmétique par intervalles constitue une bonne approche pour répondre à l'exigence de fiabilité des calculs. En effet, elle repose sur le principe que tout calcul retourne un encadrement garanti de son résultat. De plus, elle peut être implantée efficacement sur ordinateur et une panoplie d'algorithmes numériques ont été développés spécifiquement pour tirer parti de cette arithmétique. Il serait naïf de croire que les algorithmes utilisés en arithmétique flottante donnent des résultats probants sur des intervalles, le plus souvent les encadrements obtenus sont trop pessimistes. En revanche, des algorithmes conçus pour l'arithmétique par intervalles, le plus souvent des algorithmes itératifs basés sur une itération contractante, fournissent des informations et des résultats inaccessibles en flottant. En utilisant l'arithmétique par intervalles, on peut non seulement déterminer toutes les solutions mais également prouver leur existence ou leur unicité.
Un dernier aspect qu'il est important de développer, ne serait-ce pour convaincre par l'exemple de futurs utilisateurs, est la résolution de problèmes pratiques. En France, on peut citer l'intérêt croissant des automaticiens pour le calcul ensembliste en général et le calcul par intervalles en particulier ; Le livre de Jaulin [L.Jaulin and E.Walter,] en fait foi, tout comme les travaux du groupe de travail *Méthodes ensemblistes pour l'automatique*,cf.
`http://www-lag.ensieg.inpg.fr/gt-ensembliste/.`

Chapitre 3

La planification des chemins via les méthodes ensemblistes

3.1 Introduction

Le principe de modélisation par cellules discrètes est certainement celui qui est actuellement le plus employé. Il consiste à transformer l'environnement, soit dans sa totalité c'est-à-dire espace libre et espace occupé, soit partiellement (espace libre) en un ensemble de cellules discrètes connexes. La différence entre les méthodes de discrétisation réside essentiellement dans la forme attribuée aux cellules et dans la manière de les représenter informatiquement. On a proposé des méthodes ensemblistes basées sur l'arithmétique par intervalles pour la discrétisation de l'espace de recherche en affectant au terme cellule la notion de pavés. Alors ces méthodes consistent à discrétiser l'espace en pavés de type $\mathbf{A} = [X_{min}, X_{max}][Y_{min}, Y_{max}]$ chaque dimension représente un intervalle. Les valeurs attribuées aux bornes des intervalles sont fonction de l'algorithme **SIVIA** vu dans le chapitre précédent consistant à partager chaque dimension en deux et de poursuivre la décomposition des cellules obtenues selon chaque dimension tant que les cellules ne sont pas d'un même type (occupé ou libre). Nous obtenons ainsi des cellules rectangulaires en 2D et des boites cubiques en 3D

Dans ce qui vient nous allons montrer comment résoudre le problème du déménageur de piano. En premier lieu on va appliquer des algorithmes ensemblistes pour la décomposition de l'espace de recherche en vue de séparer les configurations admissibles de celle où l'objet entre en collision avec les obstacles.

3.2 Décomposition de l'espace de recherche

Exemple

Considérons un objet représenté par un polygone non-convexe avec i_{max} sommets notés $\overrightarrow{s_i} \in \mathbb{R}^2, i \in \mathcal{I} = \{1, ...i_{max}\}$ qui se déplace dans un environnement bi-dimensionnel contenant j_{max} obstacles représentés par des segments, les extrémités du $j^{ème}$ obstacle seront notés $\overrightarrow{a_j}$, $\overrightarrow{b_j}$ avec $j \in \mathcal{J} = \{1...j_{max}\}$, nous traitons cet exemple avec $j_{max} = 2$ *et* $i_{max} = 14$.
Les x-coordonnées de l'objet sont données par :

$$\{0, 0, 14, 14, 10, 12, 12, 2, 2, 18, 18, 20, 20\}, \tag{3.1}$$

et les y-coordonnées sont données par :

$$\{0, 14, 14, 6, 6, 8, 8, 12, 12, 2, 2, 18, 18, 0\}. \tag{3.2}$$

Les coordonnées des segments des obstacles sont données par :

$$\overrightarrow{a_1} = (8, 10);\ \overrightarrow{b_1} = (9, 10); \overrightarrow{a_2} = (27, 10);\ \overrightarrow{b_2} = (28, 10); \tag{3.3}$$

le sommet s_1 doit rester sur l'axe horizontal de l'environnement avec l'équation $y = 0$ (voir figure 3.1). La configuration de l'objet sera représentée par un vecteur $\overrightarrow{p}(p_1, p_2)^T$, où le premier paramètre p_1 est

la x-coordonnée de s_1 sur l'axe horizontal, et $p_2$représente l'angle (en radian) de l'objet vis-à-vis de l'axe horizontal vers la droite (voir figure 3.2). L'objectif est de passer de la configuration $(0,0)^T$ à la configuration $(17,0)^T$ représentées sur la figure(3.1)

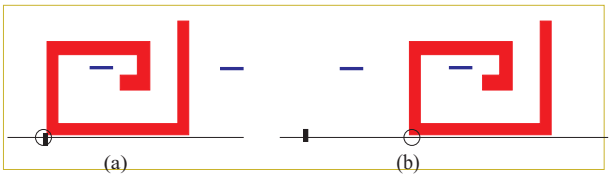


FIG. 3.1 – La figure représente les deux configurations initiale (a) et finale (b).

3.2.1 Test d'inclusion

Une configuration quelconque $\overrightarrow{p} = (p_1, p_2)$ est faisable si est seulement si
1. aucun des bords de l'objet n'intersecte un des segments obstacles ;
2. chaque segment obstacle est en dehors de l'objet.

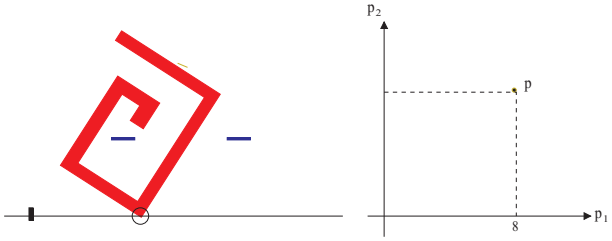


FIG. 3.2 – Une configuration donnée de l'objet sera représentée par un seul point dans l'espace des configuration CS_A, $p = (8, \pi/4)$ est une configuration admissible.

Pour caractériser ces tests nous introduisons les notations suivants :
un segment d'extrémités $\overrightarrow{a}$, $\overrightarrow{b}$ sera noté $segm(\overrightarrow{a}, \overrightarrow{b})$
$line(\overrightarrow{a}, \overrightarrow{b})$ est la ligne qui passe par $\overrightarrow{a}$ et $\overrightarrow{b}$;
$[\mathcal{A}]$ est le plus petit intervalle qui contient l'ensemble $\mathcal{A}$;
$[\overrightarrow{a} \cup \overrightarrow{b}]$ nommé l'union interne représente le plus petit pavé qui contient $\overrightarrow{a}$ et $\overrightarrow{b}$;
le sommet $\overrightarrow{s_i}_{max+1}$ représente le sommet $\overrightarrow{s_1}$
$\mathcal{S}$ est l'ensemble de toutes les configurations faisable

$$\overrightarrow{p} \in \mathcal{S} \Leftrightarrow \begin{cases} \forall i \in \mathcal{I}, \forall j \in \mathcal{J}, [\overrightarrow{s_i}, \overrightarrow{s_i}_{+1}] \cap \left[\overrightarrow{a_j}, \overrightarrow{b}_j\right] = \emptyset & \text{et} \\ \overrightarrow{a_j} \text{ et } \overrightarrow{b_j} \text{ en dehors de l'objet.} \end{cases} \tag{3.4}$$

Test d'intersection des segments de l'objet et de l'obstacle

Pour tester si

$$\forall i \in \mathcal{I}, \forall j \in \mathcal{J}, [\overrightarrow{s_i}, \overrightarrow{s_i}_{+1}] \cap \left[\overrightarrow{a_j}, \overrightarrow{b}_j\right] = \emptyset$$

on utilise l'équivalence suivante(voir annexe B)

$$[\overrightarrow{s_i}, \overrightarrow{s_i}_{+1}] \cap \left[\overrightarrow{a_j}, \overrightarrow{b}_j\right] \neq \emptyset \Leftrightarrow \begin{cases} line\left(\overrightarrow{s_i}, \overrightarrow{s_i}_{+1}\right) \cap segm\left(\overrightarrow{a_j}, \overrightarrow{b}_j\right) \neq \emptyset \\ segm\left(\overrightarrow{s_i}, \overrightarrow{s_i}_{+1}\right) \cap line\left(\overrightarrow{a_j}, \overrightarrow{b}_j\right) \neq \emptyset \\ [\overrightarrow{s_i} \cup \overrightarrow{s_i}_{+1}] \cap \left[\overrightarrow{a_j} \cup \overrightarrow{b_j}\right] \neq \emptyset \end{cases}$$

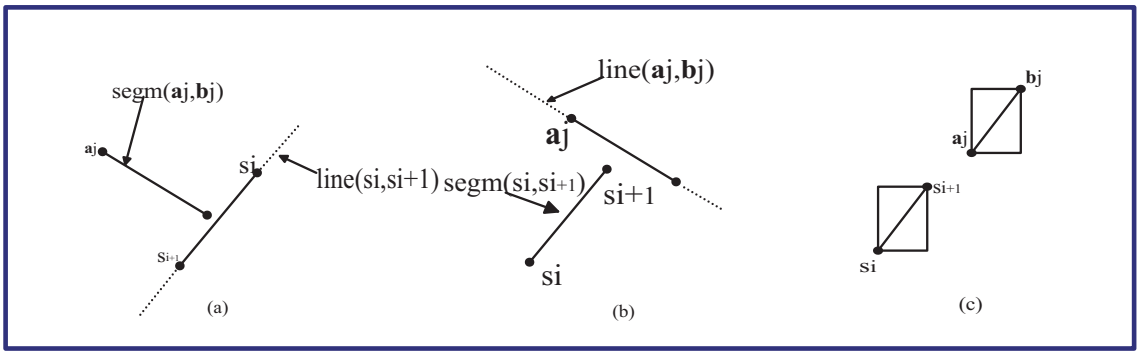


FIG. 3.3 – La figure montre les trois configurations possibles pour un intersection vide entre deux segments

la dernière condition est importante pour la situation produite où les segments sont alignés (voir figure 3.3 (c)). En ce cas, les deux premieres conditions sont toujours remplies et seule la dernière condition permet de determiner si l'intersection est non nulle.

Test pour savoir si le segment obstacle est hors de l'objet

Pour un $j^{ème}$ segment obstacle donné, $\tilde{a} = (\tilde{x_a}, \tilde{y_a})^T$ et $\tilde{b} = (\tilde{x_b}, \tilde{y_b})^T$ représentent les points extrêmes de ce segment dans le référentiel de l'objet, ces coordonnées sont données par :
$\tilde{x}_a = (x_a(j) - p_1)\cos p_2 + y_a(j)\sin p_2$
$\tilde{y}_a = -(x_a(j) - p_1)\sin p_2 + y_a(j)\cos p_2$
$\tilde{x}_b = (x_b(j) - p_1)\cos p_2 + y_b(j)\sin p_2$
$\tilde{y}_b = -(x_b(j) - p_1)\sin p_2 + y_b(j)\cos p_2$
$\tilde{a} = (\tilde{x}_a, \tilde{y}_a)^T, \tilde{b} = (\tilde{x}_b, \tilde{y}_b)^T$
pour montrer que $\tilde{a}$ est à l'intérieur de l'objet, il suffit de vérifier

$$\sum_{i=1}^{i_{max}} \arg(\overrightarrow{s_i} - \tilde{a}, \overrightarrow{s_i}_{+1} - \tilde{a}) \neq 0 \tag{3.5}$$

de même avec $\tilde{b}$ (voir Figure 3.4).

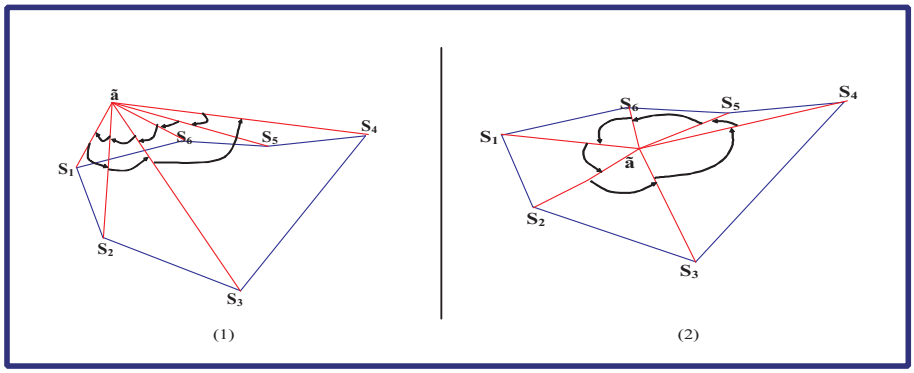


FIG. 3.4 – dans(1) le point $\tilde{a}$ est en dehors du polygone ce qui fait la somme de toutes les angles $(\widehat{s_i\tilde{a}, \tilde{a}s_{i+1}})$ est égale à 0, par contre dans (2) le point se trouve dans le polygone, d'où une somme égale à 2π.

Entrée($\overrightarrow{p}$)
Sortie($t(\overrightarrow{p})$)

begin
  for $j = 1$ *jusqu'à* j_{max} **do**
    $\tilde{x}_a = (x_a(j) - p_1)\cos p_2 + y_a(j)\sin p_2$
    $\tilde{y}_a = -(x_a(j) - p_1)\sin p_2 + y_a(j)\cos p_2$
    $\tilde{x}_b = (x_b(j) - p_1)\cos p_2 + y_b(j)\sin p_2$
    $\tilde{y}_b = -(x_b(j) - p_1)\sin p_2 + y_b(j)\cos p_2$
    $\tilde{a} = (\tilde{x}_a, \tilde{y}_a)^T, \tilde{b} = (\tilde{x}_b, \tilde{y}_b)^T$
    if $\tilde{a}$ *est dans l'objet* **then** t :=0 return ;
    if $\tilde{b}$ *est dans l'objet* **then** t :=0 return ;
    for $i = 1$ *jusqu'à* i_{max} **do**
      if $[\tilde{a} \cup \tilde{b}] \cap [\overrightarrow{s_i}_{+1} \cup \overrightarrow{s_i}] = \emptyset$ **then**
        t :=1 return ;
      end
      $d_1 = \det(\overrightarrow{s_i} - \tilde{b}, \overrightarrow{s_i} - \tilde{a}) \times \det(\overrightarrow{s_i}_{+1} - \tilde{b}, \overrightarrow{s_i}_{+1} - \tilde{a})$
      $d_2 = \det(\overrightarrow{s_i}_{+1} - \overrightarrow{s_i}, \overrightarrow{s_i}_{+1} - \tilde{a}) \times \det(\overrightarrow{s_i}_{+1} - \overrightarrow{s_i}, \overrightarrow{s_i}_{+1} - \tilde{b})$
      if $d_1 \leqslant 0$ *et* $d_2 \leqslant 0$ **then**
        t :=0 return ;
      end
    end
  end
  t :=1 return ;
end

Algorithm 5: test pour déterminer si un vecteur de configuration est faisable ou non.

Le test d'inclusion de $t(\overrightarrow{p})$ (*voir section* 2.4.1 *chapitre2*) est donné par l'**Algorithme** 6

Entrée($[\overrightarrow{p}]$)
Sortie($[t]([\overrightarrow{p}])$)

begin
  $[t] := 1$
  for $j = 1$ *jusqu'à* j_{max} **do**
    $[\tilde{x}_a] = (x_a(j) - p_1)\cos[p_2] + y_a(j)\sin[p_2]$
    $[\tilde{y}_a] = -(x_a(j) - p_1)\sin[p_2] + y_a(j)\cos[p_2]$
    $[\tilde{x}_b] = (x_b(j) - p_1)\cos[p_2] + y_b(j)\sin[p_2]$
    $[\tilde{y}_b] = -(x_b(j) - p_1)\sin[p_2] + y_b(j)\cos[p_2]$
    $[\tilde{a}] = ([\tilde{x}_a], [\tilde{y}_a])^T, [\tilde{b}] = ([\tilde{x}_b], [\tilde{y}_b])^T$
    for $i = 1$ *jusqu'à* i_{max} **do**
      $[d_1] = \det(\overrightarrow{s_i} - [\tilde{b}], \overrightarrow{s_i} - [\tilde{a}]) \times \det(\overrightarrow{s_i}_{+1} - [\tilde{b}], \overrightarrow{s_i}_{+1} - [\tilde{a}])$
      $[d_2] = \det(\overrightarrow{s_i}_{+1} - \overrightarrow{s_i}, \overrightarrow{s_i}_{+1} - [\tilde{a}]) \times \det(\overrightarrow{s_i}_{+1} - \overrightarrow{s_i}, \overrightarrow{s_i}_{+1} - [\tilde{b}])$
      if $[d_1] \leqslant 0$ *et* $[d_2] \leqslant 0$ **then** $[t] := 0$
      if $(0 \in [d_1]$ *ou* $0 \in [d_2])$ *et* $\left[[\tilde{a}] \cup [\tilde{b}]\right] \cap [\overrightarrow{s_i}_{+1} \cup \overrightarrow{s_i}] \neq \emptyset$ **then** $[t] := [0, 1]$;
    end
    if $[\tilde{a}]$ *est dans l'objet* **then** $[t] := 0$
    if $[\tilde{b}]$ *est dans l'objet* **then** $[t] := 0$
  end
end

Algorithm 6: Algorithme de test d'inclusion $[t]([p])$

3.2.2 Algorithme de décomposition

L'algorithme 3.2.2 est employé pour établir les sous-pavages R et ∂R, son fonctionnement est le suivant : Si un pavé $[P_0]$ est initialement rangé dans la file Q. Lors de la procédure de parcours de Q, nous récupérons

à chaque fois le pavé au sommet de Q dans un pavé temporaire $[x]$, la file est ensuite vidée de l'élément du sommet. Le test $[t]$ est ensuite effectué sur le pavé $[x]$. Si le test donne 1 alors $[x]$ est rangé en queue d'une file R servant à stocker les pavés acceptables (qui représentent les configurations faisables). Par contre si le pavé $[x]$ est ambigu ($[t] = [0, 1]$) et la longueur $width[x]$ est supérieure à ε, nous procédons à une bissection de $[x]$. Les deux pavés générés $[x_1], [x_2]$ sont ensuite rangés en queue de la file Q. Lorsque ($[t]([x]) = [0, 1]$ et $width[x] \leqslant \varepsilon$) le pavé $[x]$ est rangé dans ∂R servant à stocker les pavés incertains.

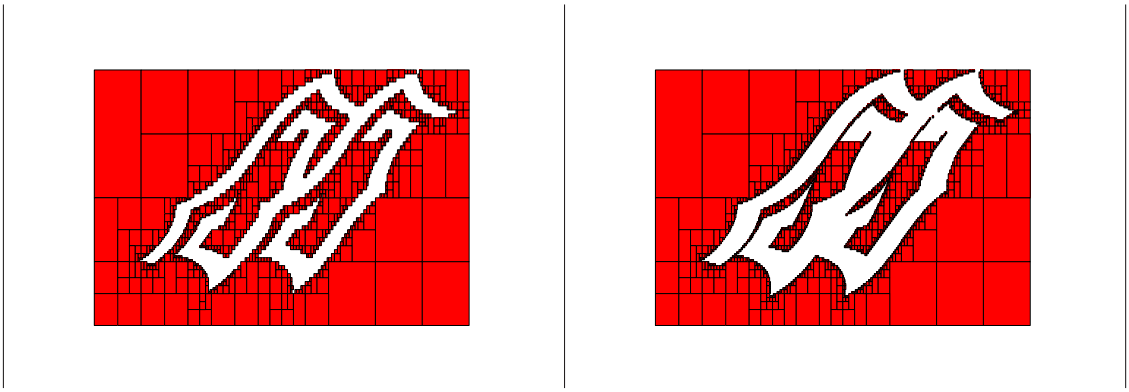


FIG. 3.5 – les deux figures présentent l'espace de configuration après la décomposition. La longueur des obstacles utilisés dans la première figure est plus petite que celle utilisée dans la deuxième figure

Entrée([t]([x]),$[P_0], \varepsilon$)
Sortie($R, \partial R$)

begin
  Initialiser la file Q *à* $[P_0]$
  Initialiser la file R *à* $\emptyset$
  Initialiser la file ∂R *à* $\emptyset$
  while $Q \neq \emptyset$ **do**
    $[x]$=ExtraireEnTête(Q)
    $Q := Q - [x]$
    if [t]($[x]$) =1 **then**
      $R := R \cup [x]$
    end
    if [t]($[x]$) =$[0, 1]$ *et* $width[x] \leqslant \varepsilon$ **then**
      $\partial R := \partial R \cup [x]$
    end
    if [t]($[x]$) =$[0, 1]$ *et* $width[x] > \varepsilon$ **then**
      $([x_1], [x_2]) = bisect([x])$
      $stack[x_1]$ *et* $[x_2]$*en queue de* Q
    end
  end
  return($R, \partial R$)
end

Algorithm 7: Algorithme de décomposition

3.3 La recherche d'un plus court chemin

Dans un problème de plus cours chemin, on possède en entrée un graphe pondéré $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ avec une fonction de pondération $w : \mathcal{A} \rightarrow \mathcal{R}$ qui fait correspondre à chaque arête un poids à valeur réelle.
Le poids du chemin $\ell =< \overrightarrow{v}_0, \overrightarrow{v}_1, \overrightarrow{v}_2, \cdots\cdots, \overrightarrow{v}_k >$ est la somme des poids des arêtes qui le constituent :

$$w(\ell) = \sum_{i=1}^{k} w(\overrightarrow{v}_{i-1}, \overrightarrow{v}_i) \tag{3.6}$$

On définit le poids du plus court chemin entre $\overrightarrow{u}$ et $\overrightarrow{v}$ par :

$$\partial(\overrightarrow{u}, \overrightarrow{v}) = \begin{cases} \min\{w(\ell) : \overrightarrow{u} \overset{\ell}{\rightsquigarrow} \overrightarrow{v}\} & \text{s'il existe un chemin de } \overrightarrow{u} \text{ à } \overrightarrow{v}, \\ \infty & \textit{sinon.} \end{cases} \tag{3.7}$$

On souhaite souvent calculer non seulement le poids des plus courts chemins, mais aussi les sommets présents sur ces plus courts chemins. Le problème de planification de trajectoires est traduit par le graphe d'adjacence des pavés extérieurs aux obstacles $\mathcal{G}(\mathcal{S}, \mathcal{A})$, tel que l'ensemble $\mathcal{S}$ représente les centres des pavés de l'ensemble R, et $\mathcal{A}$ représente l'ensemble des arêtes du graphes, le problème posé est comment définir l'ensemble $\mathcal{A}$.

3.3.1 Algorithme de recherche du plus court chemin

Dans un algorithme de recherche de plus court chemin généralement les arêtes ou bien les arcs du graphe de recherche sont organisés en *listes de voisinage*. Étant donné un graphe $\mathcal{G}$ la liste de voisinage $Adj([\overrightarrow{u}])$ d'un sommet $\overrightarrow{u} \in \mathcal{G}$ est une liste chaînée des voisins de $\overrightarrow{v}$ dans $\mathcal{G}$. Donc si l'arête $(\overrightarrow{u}, \overrightarrow{v}) \in \mathcal{A}$ le sommet $\overrightarrow{v}$ apparaît dans la liste $Adj([\overrightarrow{u}])$. Cette dernière contient les sommets $(\overrightarrow{v})$ pour lesquels il existe un chemin direct $(\overrightarrow{u}, \overrightarrow{v})$ inclus entièrement dans R. Ce chemin existe si et seulement s'il n'intersecte aucun pavé de la zone indéterminée présentée par ∂R.

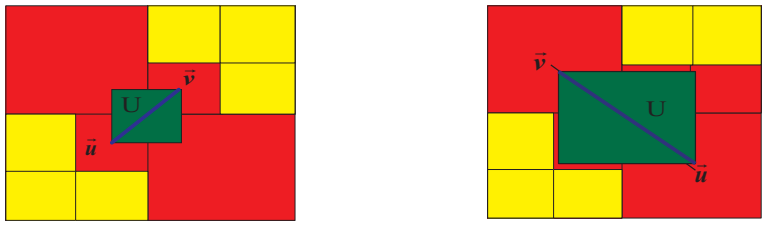
```
Entrée(D, G, ∂S)
Sortie(Adj[G], ℓ)
begin
    bool=0 ;
    while D ≠ ∅ do
        [x]=ExtraireEnTête(u)
        Adj[u] := ∅ D := D − [u]
        for all v ∈ D do
            U :=[u ∪ v]
            if [t](U) = 1 then Adj[u] := v
            else
                Γ := U ∩ ∂S
                if Γ = ∅ then Adj[u] := v
                else
                    for all [P] ∈ Γ do
                    if [P] ∩ segm(u, v) = ∅ then
                    Adj[u] := v
                end
            end
        end
    end
    return ℓ=Dijkstra(Adj[G])
end
```

Algorithm 8: Algorithme de recherche du plus court chemin passant par les centres des pavés de l'ensemble $\mathcal{S}$.

3.3.2 Présentation de l'algorithme

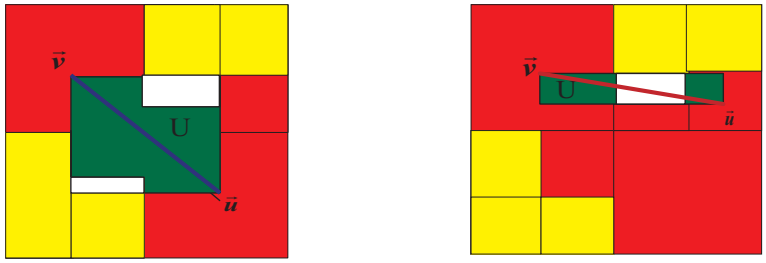
Préalablement tous les sommets de $\mathcal{G}$ sont stockés dans la liste $\mathcal{D}$, pour chaque sommet $\overrightarrow{u}$ de $\mathcal{D}$ une liste $Adj[\overrightarrow{u}]$ est initialisée, elle sert à stocker les sommets voisins de ce sommet, ensuite la liste $\mathcal{D}$ est vidée de ce sommet. Après cette opération pour chaque sommet $\overrightarrow{v}$, resté dans $\mathcal{D}$ nous calculons le plus petit pavé $\mathcal{U}$ qui contient les deux sommets $\overrightarrow{u}$, $\overrightarrow{v}$, et puis nous effectuons le test $[t]$ sur le pavé $\mathcal{U}$. Si le test donne 1 alors ce pavé est inclus strictement dans l'ensemble R qui assure que le chemin direct de $\overrightarrow{u}$ vers $\overrightarrow{v}$ est praticable, donc la liste $Adj[\overrightarrow{u}]$ reçoit le sommet $\overrightarrow{v}$ (voir figure 3.6,(1)). Sinon nous calculons l'ensemble Γ qui contient tous les pavés incertains inclus dans $\mathcal{U}$. Si cet ensemble est vide alors les deux sommets sont des voisins (voir figure 3.6,(2)), s'il n'est pas vide et si l'intersection entre l'ensemble Γ et le chemin direct de $\overrightarrow{u}$ vers $\overrightarrow{v}$ est vide alors les deux sommets sont des voisins (voir figure 3.6(3) et 3.6(4)) c'est-à-dire qu'ils

seront reliés par une arête dans le graphe $\mathcal{G}$.



(1) (2)

FIG. 3.6 – Les figures ci-dessus montrent les deux premiers cas traités par l'**Algorithme 8** la figure (1) présente le cas où le pavé $\mathcal{U}$ est inclus strictement dans l'espace libre ([t]($\mathcal{U}$)=1). La figure (2) montre le cas où [t]($\mathcal{U}$)=[0,1] et l'intersection de $\mathcal{U}$ et de l'ensemble des pavés incertains est vide. Pour les deux cas le chemin de u à v est faisable.



(3) (4)

FIG. 3.7 – Les figures ci-dessus montrent deux situations où l'intersection entre le pavé $\mathcal{U}$ et l'ensemble des pavés incertains $\partial\mathcal{S}$ n'est pas vide. Il s'agit des pavés blancs, ils sont regroupés dans l'ensemble Γ. Figure (3) : l'intersection entre le segment (u, v) et chaque pavé de Γ est vide il existe donc un chemin direct entre u et v. Figure(4) : cette intersection est non vide le chemin est impraticable.

Après la structuration de l'espace libre en un graphe de centres des pavés, l'exploration de ce graphe est basée sur l'algorithme de Dijkstra (voir chapitre 1 section 1.5.1 Algorithme 2). Le chemin trouvé passe par une série de pavés qui contient le plus court chemin (voir figure 3.8).

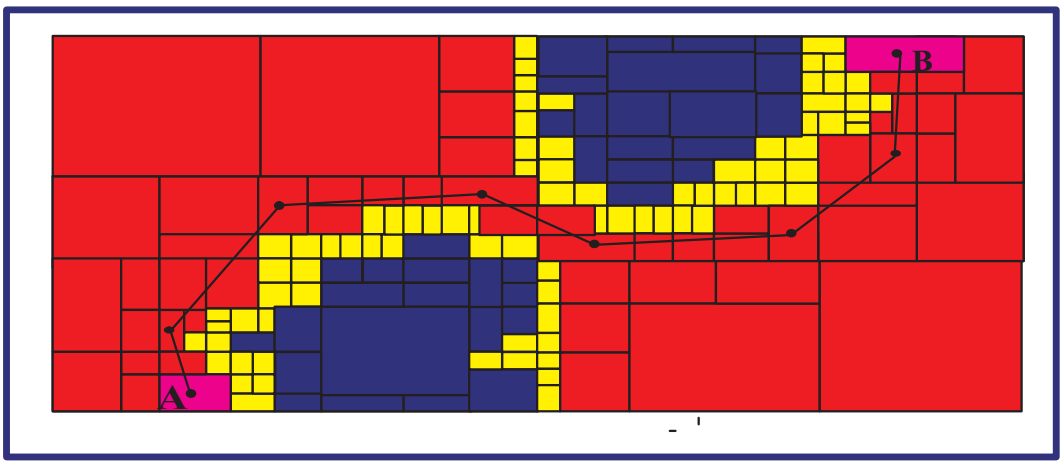


FIG. 3.8 – La figure montre le plus court chemin de **A** à **B** qui passe par les centres des pavés

Conclusion générale

Les problèmes de planification de chemins demandent des réponses fiables que ne peuvent fournir les méthodes numériques ou formelles classiques. Ceci est principalement lié à l'abondance de fonction trigonométriques et discontinues dans la formulation des problèmes, qui rend inefficace les méthodes locales. En revanche, les méthodes intervalles se montrent particulièrement adaptées. Nous avons proposé au cours de ce travail une technique de calcul de trajectoires efficace car elle repose sur des modèles simples. Nous avons alors essentiellement développé une technique de planification des chemins pour les robots mobiles via des méthodes ensemblistes.

Annexe A

Comparons les performances des fonctions d'inclusion établies à partir des formulations de la même fonction suivante :

$$f_1(x) = x(x+1);$$
$$f_2(x) = x \times x + x;$$
$$f_3(x) = x^2 + x;$$
$$f_4(x) = (x + \frac{1}{2})^2 - 1;$$

pour l'intervalle $[x] = [-1, 1]$, les évaluations intervalles des expressions précédentes sont :

$$f_{1[]}([x]) = [x]([x]+1) = [-2, 2];$$
$$f_{2[]}([x]) = [x] \times [x] + [x] = [-2, 2];$$
$$f_{3[]}([x]) = [x]^2 + [x] = [-1, 2];$$
$$f_{4[]}([x]) = ([x] + \frac{1}{2})^2 - \frac{1}{4} = [-\frac{1}{4}, 2];$$

Selon l'expression de la fonction, l'encadremment obtenu sera donc plus ou moins grossier (voir figure 3.9). Les deux expressions $[x] \times [x]$ et $[x]^2$ ne sont pas équivalentes, dans le premier cas, chacune des occurrences de x peut varier indépendamment. Précisons que l'ensemble image de la fonction initiale est $[-\frac{1}{4}, 2]$. $f_{4[]}$ est donc minimale

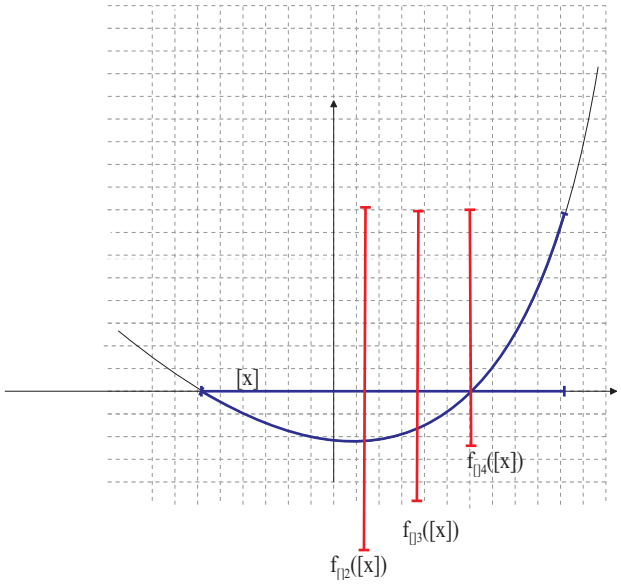


FIG. 3.9 – Comparaison de fonctions d'inclusion d'une même fonction

Annexe B

Position d'un point par rapport à une droite

Problème : Déterminer si un point A se situe d'un coté, de l'autre, ou sur une droite donnée, repérée par deux de ses points B_1 *et* B_2.

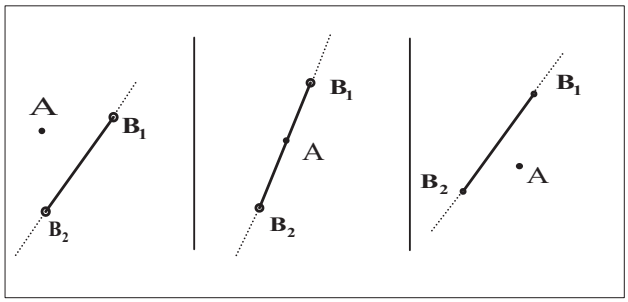


FIG. 3.10 – Position d'un point par rapport à une droite

Méthode : Etudier le signe du déterminant $DET(A - B_1, B_2 - B_1)$ comme critère de l'orientation de l'angle entre ces deux vecteurs.

Solution : renvoyer le signe de $\begin{vmatrix} A.x - B_1.x & B_2.x - B_1.x \\ A.y - B_1.y & B_2.y - B_1.y \end{vmatrix}$.

Le coté $(A, (B_1, B_2)) = signe(DET(A - B_1, B_2 - B_1))$
Remarque : Ce signe est fonction du fait que le système de coordonnées (x, y) est direct ou indirect.

Intersection de deux segments

Problème : Déterminer si deux segments $[A_1, A_2]$ et $[B_1, B_2]$ ont une intersection vide ou non.

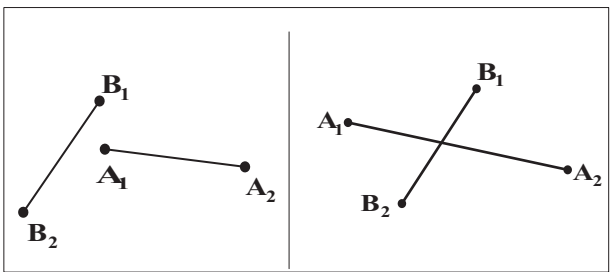


FIG. 3.11 – Intersection entre deux segments

Méthode : On vérifie que les points A_1 et A_2 sont de part et d'autre de la droite (B_1, B_2) et que les points B_1 et B_2 sont sont de part et d'autre de la droite (A_1, A_2).
Solution :

$$intersect((A_1, A_2), (B_1, B_2)) =$$

$$Max(cot(A_1, (B_1, B_2)) \times \ cot(A_2, (B_1, B_2)), cot(B_1, (A_1, A_2)) \times cot(B_2, (A_1, A_2)))$$

Remarque : On peut, en considérant de plus prés les cas d'égalités à 0, spécifier les cas d'intersection sur les extrémités des segments.

Annexe C

Nous fournissons dans cette annexe les scripts SCILAB qui permet la décomposition de l'espace de configuration par les méthode ensemblistes ainsi de trouver un plus court chemin passant par les centres des pavés

Remarque 1 *Toutes les fonctions qui serons utilisés dans la suite sont regrouper dans un script SCILAB.*

Fonctions utiles dans la librairie calcul ensembliste

```
////////////////////////////////////////////////////////////////////////////
//                                                                        //
//        ##########== INTERVAL ARITHMETIC==##########                    //
//                                                                        //
//                                                                        //
////////////////////////////////////////////////////////////////////////////


//--------- l'ensemble vide :

function z=i_empty()
           z=[%nan,%nan];
endfunction

//--------Test si un ensemble est vide :

function b=i_isempty(x)
                b=(isnan(x(1))|isnan(x(2)));
endfunction

//---------Le centre d'un intervalle:

function c=i_centre(x)
         c=x(1)+(x(2)-x(1))/2;
endfunction

//------Test ci une nombre est dans l'intervalle :

function b=i_in(a,x)
           b=(a>=x(1))&(a<=x(2));
            if i_isempty(x) then b=%F
            end
endfunction

//-----------Inclusion :
```

```
function b=i_subset(x,y)
           if i_isempty(x) then b=%T
             elseif i_isempty(y) then b=%F
                 else b=(x(1)>=y(1))&(x(2)<=y(2))
         end
endfunction

//-----------Intersection :

function z=i_inter(x,y)
           z=[max(x(1),y(1)),min(x(2),y(2))];
           if i_isempty(x)|i_isempty(y)|(z(1)>z(2)) then
              z=i_empty()
           end;
endfunction

//--------Union :

function z=i_union(x,y)
          z=[min(x(1),y(1)),max(x(2),y(2))];
           if i_isempty(y)  then z=y
           end;
           if i_isempty(y) then z=x
           end;
endfunction

//---------Somme :

function z=i_plus(x,y);
           z=x+y;
endfunction

//--------Soustraction :

function z=i_minus(x,y)
           z=[x(1)-y(2),x(2)-y(1)];
endfunction

//--------Multiplication :

function z=i_mult(x,y)
           v=[x(1)*y(1),x(1)*y(2),x(2)*y(1),x(2)*y(2)];
           z=[min(v),max(v)];
endfunction

//-------Inverse:

function y=i_inv(x)
           if  i_isempty(x)   then y=i_empty()
             elseif  i_in(0,x)    then y=[-%inf,%inf]
                 else y=[1/x(2),1/x(1)];
           end
endfunction

//-------Division :

function z=i_div(x,y)
```

```
           a=i_inv(y);
           z=i_mult(x,a);
endfunction

//--------Fonction Exp :

function z=i_exp(x)
         if (i_isempty(x)) then z=i_empty();
         else z=[exp(x(1)),exp(x(2))];
         end;
endfunction

//---------Fonction Log :

function z=i_log(x)
       if (i_isempty(x))|(x(2)<=0) then z=i_empty();
       elseif i_in(0,x) then z=[-%inf,log(x(2))];
       elseif (x(1)>0) then z=[log(x(1)),log(x(2))];
       end;
endfunction

//--------Carré :

function z=i_sqr(x)
        if  i_isempty(x) then   z=i_empty();
        elseif  x(2)<=0     then z=[(x(2))^2,(x(1))^2];
        elseif  x(1)>=0     then z=[(x(1))^2,(x(2))^2];
        elseif       i_in(0,x)  then z=[0,max(x(1),x(2))^2];
        end;
endfunction

//----------Racine carrée :

function z=i_sqrt(x)
        if  i_isempty(x) then   z=i_empty();
        elseif  x(2)>=0     then z=[-sqrt(x(2)),sqrt(x(1))];
        else z=i_empty();
        end;
endfunction

//-----------Carré positif :

function z=i_sqrtpos(a)
        if  i_isempty(a) then   z=i_empty();
        elseif a(2)<0 then z=i_empty();
        elseif a(1)<0 then z(1)=0;z(2)=sqrt(a(2));
        else z=[sqrt(a(1)),sqrt(a(2))]
        end;
endfunction

//-------Fonctin Modulo :

function z=i_fmod(x,m)
         if (x(1)>=0)&(x(2)<m) z=x;
         else k=floor(x(1)/m);
              offset=m*k;
              z(1)=x(1)-offset;
```

```
              z(2)=x(2)-offset;
          end;
endfunction

//-----------Fonction Sin :

function z=i_sin(x)
           y=i_fmod(x,(2*(%pi)));
           if i_isempty(x) then z=i_empty();
           else z(2)=max(sin(y(1)),sin(y(2))),
                z(1)=min(sin(y(1)),sin(y(2)));
           end;
           if  x(2)-x(1)>=2*%pi then z=[-1,1];
           end;
           if ((i_in(3*(%pi/2),y))|(i_in(7*(%pi/2),y)))then z(1)=-1;
           end;
           if ((i_in((%pi/2),y))|(i_in(5*(%pi/2),y)))then z(2)=1;
           end;
           z=[z(1),z(2)],
endfunction

//----------Fonction Cos :

function z=i_cos(x)
          y=i_plus(x,[%pi/2,%pi/2]);
          z=i_sin(y);
endfunction

//----------Distance Euclidienne :

function z=i_dEuc(p,q)
          z=sqrt(((p(1)-q(1))^2)+((p(2)-q(2))^2));
endfunction

//--------Determinant :

function z=determinant(A,B)
          z=i_minus((i_mult(A(1,:),B(2,:))),(i_mult(A(2,:),B(1,:))));
endfunction
//------------Le plus petit élément :

function z=Toreel(X)
          if size(X)==[1,1] then z=X
          else
              S=size(X(1,:));m=S(1);n=S(2);
              for j=1:m*n-1
                  d=X(j);
                  z=min(X(j+1),d);
              end;
          end;
endfunction

//----------Fonction Arccos :

function z=i_acossin(x,y)
          if y>0 then z=(acos(x));
          else z=(-acos(x));
```

```
         end;
endfunction
//-----------Ordre :

function z=re(x)
         if x(1)<x(2) then z=x;
         else z(2)=x(1),z(1)=x(2);
         end;
endfunction

//-----------Matricialisation :

function z=i_mat(x)
         if size(x)==[1,2]then z=[x;x];
         elseif size(x)==1 then z=[x,x];
         elseif size(x)==[2,1]then z=[x,x];
         elseif size(x)==[2,2]then z=x,
         end;
endfunction

////////////////////////////////////////////////////////////////////////////
//                                                                        //
//                  ###########==   BOXES   ==###########                 //
//                                                                        //
//                                                                        //
////////////////////////////////////////////////////////////////////////////

//---------Pavé vide :

function z=b_empty()
                  z=[%nan,%nan;%nan,%nan];
endfunction

//---------------------------------Largeur d'un pavé-----:

function w=b_width(X)
         w=max(X(:,2)-X(:,1));
endfunction

//------------Centre d'un Pavé :

function b=b_center(X)
         b(1)=X(1)+((X(3)-X(1))/2);
         b(2)=X(2)+((X(4)-X(2))/2);
         b=[b(1);b(2)];
endfunction

//------------Bissection :

function  [X1,X2]=b_bisect(XX)
          X=XX;
          X(2,2)=10*XX(2,2);
          X(2,1)=10*XX(2,1);
             [a,i]=max(X(:,2)-X(:,1));
             X1=XX; X2=XX;
             m=(XX(i,2)+XX(i,1))/2;
         X1(i,2)=m;
```

```
         X2(i,1)=m;
endfunction

//-----------Dessiner un pavé :

function   b_draw(X,color)
                xset("pattern",color);
                dx=X(:,2)-X(:,1);
                xfrect(X(1,1),X(2,2),dx(1),dx(2));
                xset("pattern",0);
                xrect(X(1,1),X(2,2),dx(1),dx(2));
endfunction

//--------Union de deux pavés :

function h=b_union(k,l)
    if(size(k)==[1,2]&size(l)==[1,2])then
          h=[min(k(1),l(1)),max(k(2),l(2));min(k(2),l(2)),max(k(2),l(2))];
    elseif(size(k)==[2,1]&size(l)==[2,1])then
          h=[min(k(1),l(1)),max(k(1),l(1));min(k(2),l(2)),max(k(2),l(2))];
    elseif(size(k)==[2,2]&size(l)==[2,2])then
          h=[i_union(k(1,:),l(1,:));i_union(k(2,:),l(2,:))];
    end;
endfunction

//-------Intersection de deux pavés :

function t=b_inter(f,w)
           dx=i_inter(f(1,:),w(1,:));
           dy=i_inter(f(2,:),w(2,:));
             if (i_isempty(dx)&i_isempty(dy)) then t=b_empty();
                else t=[dx;dy];
             end;
endfunction

//--------Produit scalaire :

function PR=produit_scalaire(w,x)
         somme=[0,0];
         for i=1:2
             somme=i_plus(somme,i_mult(w(i,:),x(i,:)));
         end;
         PR=somme;
endfunction

//----------Norme d'un vecteur :

function N=Norm(A)
         r=[0,0];
         for i=1:2
            r=i_plus(r,i_sqr(A(i,:)));
         end;
         N=i_sqrtpos(r)
endfunction

//---------Angle entre deux vecteurs :
```

```
function alpha=angle(p,q)
                n=Norm(p);
                m=Norm(q);
                w=i_mult(n,m);
                k=Toreel(w);
                if (i_isempty(w)| k==0) then
                   k=-9999;
                end;
                   costheta=Toreel(produit_scalaire(p,q)/k);
                   sintheta=Toreel(determinant(p,q))/k;
                   alpha=i_acossin(costheta,sintheta)
endfunction
```

Test d'inclusion

```
////////////////////////////////////////////////////////////////////////////
//                                                                        //
//              ##########==TEST D'INCLUSION==##########                  //
//                                                                        //
//                                                                        //
////////////////////////////////////////////////////////////////////////////

//-------------test si une point A est dans l'objet :

function b=in_test(A,objet)
           TOTAL=0;
           a=b_center(A);
           aa=i_mat(a);
      for  i=1:imax-1
            M1=i_mat(objet(:,i));
            M2=i_mat(objet(:,i+1));
            alpha=angle(M1-aa, M2-aa);
            TOTAL=(TOTAL+alpha);
      end;
          if ((TOTAL>1)|(TOTAL<-1)) then b=1;
          else b=0;
          end;
endfunction
//-------------test TT==[t]
function TT=b_test(x)
         TT=1;
          cosp2=i_cos(x(2,:));
          sinp2=i_sin(x(2,:));
          p1c=i_centre(x(1,:));
          p2c=i_centre(x(2,:));
          cosp12=cos(p2c);
          sinp12=sin(p2c);
     for j=1:jmax
         AX=i_plus(i_mult(AA(1,j)-x(1,:),cosp2),
         i_mult(i_mat(AA(2,j)),sinp2));

         fc=(AA(1,j)-p1c)*cosp12+AA(2,j)*sinp12;
         df1=-cosp2;

         df2=i_plus(i_mult(-(AA(1,j)-x(1,:)),sinp2),
         i_mult(i_mat(AA(2,j)),cosp2));

         A1c=i_plus(fc+(i_mult(df1,(x(1,:)-p1c))),i_mult(df2,x(2,:)-p2c));
         AX=i_inter(AX,A1c);

         AY=i_plus(i_mult(-(AA(1,j)-x(1,:)),sinp2),
         i_mult(i_mat(AA(2,j)),cosp2));

         fcc=-(AA(1,j)-p1c)*sinp12+AA(2,j)*cosp12;
         df11=sinp2;

         df22=i_minus(i_mult(-(AA(1,j)-x(1,:)),cosp2),
         i_mult(i_mat(AA(2,j)),sinp2));

         A2c=i_plus(fcc+(i_mult(df11,(x(1,:)-p1c))),i_mult(df22,x(2,:)-p2c));
```

```
        AY=i_inter(AY,A2c);

        BX=i_plus(i_mult(BB(1,j)-x(1,:),cosp2),
        i_mult(i_mat(BB(2,j)),sinp2));

        fbc=(BB(1,j)-p1c)*cosp12+BB(2,j)*sinp12;
        dfb1=-cosp2;

        dfb2=i_plus(i_mult(-(BB(1,j)-x(1,:)),sinp2),
        i_mult(i_mat(BB(2,j)),cosp2));

        B1c=i_plus(fbc+(i_mult(dfb1,(x(1,:)-p1c))),i_mult(dfb2,x(2,:)-p2c));
        BX=i_inter(BX,B1c);

        BY=i_plus(i_mult(-(BB(1,j)-x(1,:)),sinp2),
        i_mult(i_mat(BB(2,j)),cosp2));

        fbcc=(-(BB(1,j)-p1c)*sinp12+BB(2,j)*cosp12);
        dfb11=sinp2;

        dfb22=i_minus(i_mult(-(BB(1,j)-x(1,:)),cosp2),
        i_mult(i_mat(BB(2,j)),sinp2));

        B2c=i_plus(fbcc+(i_mult(dfb11,(x(1,:)-p1c))),
        i_mult(dfb22,x(2,:)-p2c));
        BY=i_inter(BY,B2c);

          A=[AX;AY];
          B=[BX;BY];
        if (in_test(A,objet)==1 )|(in_test(B,objet)==1) then
          TT=0;
        end;
    for i=1:imax-1
          M1=i_mat(objet(:,i));
          M2=i_mat(objet(:,i+1));
          G=i_empty();
          DET_1=i_mult(determinant(B-M1,A-M1),determinant(B-M2,A-M2));
          DET_2=i_mult(determinant(M2-M1,M2-A),determinant(M2-M1,M2-B));
          G=b_inter(b_union(A,B),b_union(objet(:,i+1),objet(:,i)));

          if (max(DET_1)<0 & max(DET_2)<0 )then TT=0 ;

          elseif (min(DET_1)<=0 |min(DET_2)<=0)&(i_isempty(G)==%F)==%T
                then TT=[0,1];
          end;
     end;
     end;

endfunction
```

Décomposition de l'espace de recherche

```
////////////////////////////////////////////////////////////////////////////
//                                                                        //
//     ##########== DECOMPOSITION DE L'ESPACE DE RECHRCHE==##########     //
//                                                                        //
//                                                                        //
////////////////////////////////////////////////////////////////////////////

//-----------Les coordonées de l'objet

xobjet=[0,0,14,14,10,10,12,12,2,2,18,18,20,20,0];

yobjet=[0,14,14,6,6,8,8,12,12,2,2,18,18,0,0];

objet=[xobjet;yobjet];

imax=15;    // le 15ème sommet représente le premier sommet

//  Les coordonnées des obstacles

jmax=2

AA=[8,25;10,10]

BB=[11,28;10,10]

//-------------------------------------------------------:

getf ('D:/interval.sci');

getf ('D:/box.sci');

xsetech([0 0 11],[-28 -1.4 57 2.7]); x=[-28,57;-1.4,2.7];

//-------------------------------------------------------:

L=list(x); R=list();

while (length(L)>0)
     x=L(1) ;
       L(1)=null();      //X=pull(L)

     TT=b_test(x);

     if TT==1           then b_draw(x,5);R($+1)=x;
     elseif TT==0 then b_draw(x,2);
     elseif b_width(x)<0.1 & TT==[0,1]  then  b_draw(x,32);   //32=jaune
         else   [x1,x2]=b_bisect(x);
                L($+1)=x1     //push(X1)
                L($+1)=x2     //push(X2)
     end;
end;
```

La recherche d'un plus court chemin

```
////////////////////////////////////////////////////////////////////////////
//                                                                        //
//       ##########== le plus court chemin==##########                    //
//                                                                        //
//                                                                        //
////////////////////////////////////////////////////////////////////////////

//-------------test de visibilité :

function z=b_CV(A,B,T)
          xa=A(1);ya=A(2);
          xb=B(1);yb=B(2);
          T1=T(:,1);
                xT1=T1(1);yT1=T1(2);
                T2=T(:,2);
                xT2=T2(1);yT2=T2(2);
          if (ya==yb)&(yT1<=ya)&(ya<=yT2)then z=%T;
             elseif
                   (xa==xb)&(xT1<=xa)&(xa<=xT2)then z=%T;
             elseif
                   ((ya<>yb)&(xa<>xb))&
                   (((xT1-xb)/(xa-xb))<=((yT1-yb)/(ya-yb)))&
                   (((xT2-xb)/(xa-xb))<=((yT2-yb)/(ya-yb)))
                   then z=%T;
          else
             z=%F
          end;
endfunction

//---------------Distances entre deus paves :

function  z=poid(P,Q)
                 p=b_center(P);
                 q=b_center(Q);
                 z=i_dEuc(p,q);
endfunction

//----------------le test de voisinage   :

function z=b_voisin(A,B)
             a=b_center(A);
             b=b_center(B);
             T=b_inter(A,B);
             aa=i_mat(a);
             bb=i_mat(b);
             U=b_union(aa,bb);
             L=list(U);
             test=list();
             B_T=%T;

       while(length(L)<11)
             x=L(1);
             [x1,x2]=b_bisect(x);
             L(1+$)=x1;
             L(1+$)=x2;
```

```
       end
             S=lstsize(L);
       while(length(L)>0)
             x=L(1); L(1)=null();
             test(1+$)=b_test(x);
       end;

            FG=lstsize(test);
       for i=1:FG
            B_T=(B_T)&(test(i));
       end;

       if ~i_isempty(T)& b_CV(a,b,T) then z=%T;
            elseif B_T==%T then z=%T;
            else
                z=%F;
       end;
endfunction



//---------------La matrice d'adjacence :

G=list(); // liste des sommets S=lstsize(R); T=[];
    for i=1:S
              x=R(i);
              c=b_center(x);
       for j=i:S
                y= R(j);
                d=b_center(y);
                    if b_voisin(x,y)==%T then
                        w=i_dEuc(c,d);
                        T(i,j)=w;
                    end;
        end
    end;
  for i=1:S
     for j=i:S
         T(j,i)=T(i,j);
     end
  end


//-------------------------Dijkstra--------------------------------
//**********************----------*****************************//

         for i=1:S
             x=b_center(R(i));
             G(1+$)=x
         end
         K=zeros(1,S)
         debut=input('debut=');
         nb=lstsize(G);
           for i=1:nb
              K(i)=+%inf;
            if G(i)==debut  then
                K(i)=0;
```

```
            end;
           end;

  II=list();
  SEL=list();
  KK=K;
  P=[]
   while (length(K)>11)
           x=min(K)
          nbKK=max(size(KK));
          nbK=max(size(K));
          I1=find(KK==x);
          nbII=max(size(II));
          nbI1=max(size(I1));
              if isempty(II) then
                 I=I1($);
              else
                  for j=1:nbII
                     f=II(j);
                     t=find(I1==f);
                     I1(t)=[];
                  end
              end
              I=I1($);
              J=find(K==x);
              J=J($);
              II(1+$)=I;
              SEL(1+$)=x;

             for j=1:nbKK
                  if T(I,j)<>0 then
                      if KK(j)>KK(I)+T(I,j) then
                         KK(j)=KK(I)+T(I,j);
                         P(j)=I;
                      end;
                  end;
              end
            K=KK;
            nbSEL=lstsize(SEL);
                for i=1:nbSEL
                    d=SEL(i);
                    n=find(K==d);
                    K(n)=[];
              end
            end;

//--------------------------plus court chemin :


          BUT=input('BUT=');
          for s=1:nbKK
                if G(s)==BUT then
                   I=s
                end;
          end;

             DISTANCE_MINIMALE=KK(I)
```

```
        L=list(BUT);
            while G(I)<>debut
                  j=P(I)
                  L(1+$)=G(j);
                  I=j;
            end;
            L(1+$)=debut;

//----------------representation graphique :

        xsetech([0 0 1 1],[-30 -1.5 50 2.5]);

         for i=1:S
             x=R(i)
             b_draw(x,5);
         end
            l=lstsize(L)
         for i=1:l-1
            x=L(i)
            y=L(i+1)
             plot2d([x(1);y(1)],[x(2);y(2)])
         end
```

Bibliographie

[B.Faverjon and P.Tournassoud, 1987] B.Faverjon and P.Tournassoud (1987). *The mixed appraoch for path planning of manipulators.* March.

[J-C.Latombe, 1991] J-C.Latombe (1991). *robot motion planning.* Kluwer Academic Press.

[J.T.Schwartz and M.sharir, 1983] J.T.Schwartz and M.sharir (1983). *On the piano movers'problem.*

[L.Jaulin and E.Walter,] L.Jaulin, M.Kieffer, O. and E.Walter. *Applied Interval Analysis.* Springer.

[N.J.Nelsson, 1969] N.J.Nelsson (1969). *A mobile automaton : an application of artificial intelligence.* Washington,DC(USA).

[O'Dùnlaing and N.Yap, 1982] O'Dùnlaing and N.Yap (1982). *Journal of Algorithms.*

[O.Khatib, 1986] O.Khatib (1986). *Real-time obstacle avoidance for manpulateurs and mobile robots.* Spring.

[P. Lacomme, 2003] P. Lacomme, C. P. e. M. S. (2003). *Algorithmes de graphes.* EYROLLES.

[P.Chrétienne, 1994] P.Chrétienne, C.Hanen, A. e. C. (1994). *Intrduction à l'algorithmique.* DUNOD.

[Pérez, 1983] Pérez, T. (1983). Spacial planning : A configuration space approach. *In IEEE transactions on computers, C-32 pages 108-120.*

[P.Tournassoud, 1988] P.Tournassoud (1988). *Géométrie et intelligences artificielle pour les robots, complexité du calcule de trajectoires,.* HERMES.

[P.Tournassoud, 1992] P.Tournassoud (1992). *Planification et controle en robotique,application aux robots mobiles et manipulateurs.* HERMES.

[R.E.Moore, 1966] R.E.Moore (1966). *Interval analysis.* Prentice Hall.

[R.Moore, 1962] R.Moore (1962). *Interval arithmitic and automatic error analysis in digital computing, PhD thesis,applied Math Statistics Lab. Report 25,.* Stanford.

Résumé

Planifier un chemin pour un robot mobile consiste à balayer un espace de recherche contenant une infinité non dénombrable de points et d'en extraire un chemin optimal. Les approches ponctuelles sont souvent utilisées mais ne permettent d'analyser qu'une infime portion de l'espace de recherche. Nous proposons dans ce rapport de décomposer ce dernier en un nombre fini de sous ensembles simples que l'on caractérise aisément à l'aide de méthodes ensembliste. Cela va nous permettre de réduire la taille des graphes de recherche ainsi réduire la complexité des algorithmes utilisés.

Mot-clés : Arithmétique par intervalle, calcul garanti, Algorithme à fixation d'étiquette, Algorithme à correction d'étiquette, Algorithme SIVIA, Fonction d'inclusion, Test d'inclusion, décomposition de l'espace de recherche.

Abstract

To plan a way for a mobile robot consists in sweeping a research space containing a not countable infinity of points and to extract an optimal way from it. The specific approaches are often used but allow to analyze only one negligible portion of the research space. We propose in this report to break up this later into a finished number of simple subsets characterized easily using methods ensemblist. That will enable us to reduce the size of the graphs of research thus to reduce the complexity of the used algorithms.

Key word : Arithmetic by interval, guaranteed calculation, Algorithm with fixing of label, Algorithm with correction of label, Algorithm SIVIA, Inclusion function, Inclusion test, decomposition of the research space.