Université de Toulouse

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par **l'Institut Supérieur de l'Aéronautique et de l'Espace**
Spécialité : Systèmes embarqués

Présentée et soutenue par **Julien GUITTON**
le 31 mars 2010

Architecture hybride pour la planification d'actions et de déplacements

---

JURY

M. Rachid Alami, président
M. François Charpillet
M. Raja Chatila, directeur de thèse
M. Jean-Loup Farges, co-directeur de thèse
M. Christian Laugier, rapporteur
M. Abdel-Illah Mouaddib, rapporteur

École doctorale : **Systèmes**

Unité de recherche : **Équipe d'accueil ISAE-ONERA CSDV**

Directeur de thèse : **M. Raja Chatila**
Co-directeur de thèse : **M. Jean-Loup Farges**

À ma grand-mère,
qui n'aura jamais vu la fin de ce travail.
Tu resteras pour toujours dans mon cœur.

Remerciements

On dit toujours que l'on finit de rédiger sa thèse par l'introduction. Et bien non, on finit par les remerciements et c'est sans doute la partie la plus difficile. D'autant plus que je le fais depuis le hall de l'aéroport de Glasgow où je viens de donner mon premier "invited talk" (à l'université, pas à l'aéroport). Donc je vais essayer de faire court...

Je remercie Mr Christian Laugier et Mr Abdel-Illah Mouaddib d'avoir accepté de rapporter ma thèse. Je remercie également Mr Rachid Alami et Mr François Charpillet d'avoir accepté d'examiner cette thèse.

Je remercie mes directeurs de thèse, Mr Raja Chatila qui a su gardé un oeil sur mes travaux en plus de ses fonctions de directeur du LAAS et, Mr Jean-Loup farges pour son encadrement quotidien à l'ONERA. Merci Jean-Loup pour ton investissement, ta patience, ta confiance et ton ouverture d'esprit. Et fais moi plaisir : passe ton HDR !

Je remercie ma famille pour m'avoir soutenu moralement, financièrement et de toutes les façons possibles. Merci Maman, Papa, Laurent et William.

Concernant les autres, je vous remercie tous. Au cours de ces trois ans, à Toulouse et lors de mes déplacements, j'ai rencontré tellement de personnes intéressantes qui ont influencé mes recherches, mais également ma vie et ma façon de penser, que je ne saurais toutes les remercier. Donc, merci à tous, vous qui vous reconnaitrez.

Je me dois, quand même, de remercier toute l'équipe du DCSD pour son accueil et son aide, que ce soit d'un point de vue administratif et technique (Patrick, Valérie, Serge, Françoise et Michel), ou d'un point de vue scientifique (Catherine, Magalie, Gérard, Jean-François et tous les autres). Je remercie finalement tous les doctorants et stagiaires que j'ai été amené à cotoyer durant mon séjour à l'ONERA. Ils sont tellement nombreux que je ne les citerais pas (ainsi, je suis sûr de n'oublier personne).

Je ne peux finir ces remerciements sans quelques dédicaces spéciales :

Merci les filles (Caro, Aurélie, Laure et Lara) pour avoir féminisé un peu ce monde de brutes en cette dernière année. J'espère que vous retiendrez mes enseignements de coinche et je vous souhaite une bonne thèse.

Merci la Dream Team (Greg1, Greg2, Nico et Steph) pour tous les bons moments que nous avons passés (et j'espère que nous continuerons de passer) ensemble. Pour toutes les soirées. Pour toutes les parties de coinche. Pour toutes les discussions... et pour tout le reste.

Evidemment, merci Manu, non je ne t'ai pas oublié. Pour ton soutien et ton aide dans tous les domaines. Pour nos longues soirées autour de la bouteille de muscat. Pour les vacances en Crête. Pout tout quoi ! Tu sais que tu auras toujours une place sur le canapé. J'espère que tu réaliseras tes rêves, tu le mérites.

Enfin, merci Yoko. Pour nos longues promenades, nos soirées, nos vacances et ton soutien en cette stressante fin de thèse. Je te souhaite plein de bonheur et de réussite, avec tout mon amour.

Julien.

Table des matières

Table des figures

Liste des tableaux

Liste des algorithmes

Introduction

> *L'homme a la possibilité non seulement de penser, mais encore de savoir qu'il pense ! C'est ce qui le distinguera toujours du robot le plus perfectionné.* (Jean Delumeau)

Créer des êtres artificiels ayant pour mission de seconder l'homme dans ses tâches quotidiennes ou bien de le remplacer lors de travaux pénibles et dangereux, est un vieux rêve qui devient peu à peu réalité. On doit le terme *robot* à l'écrivain tchèque Karel Čapek qui l'utilisa dans sa pièce de théatre *Rossum's Universal Robots* en 1920. Le terme *robotique*, dénominant l'ensemble des études et techniques de conception d'un robot, a été introduit pour la première fois par Isaac Asymov dans sa nouvelle *Liar !* publiée dans *Astounding Science Fiction* en mai 1941.

De nos jours, les robots sont dotés de capacités de raisonnement primaires encore éloignées de l'intelligence que veulent leur attribuer les écrivains de science fiction. Néanmoins, ils envahissent notre quotidien dans la plupart des domaines. Les machines automatisées sont utilisées dans de nombreux secteurs de l'industrie, par exemple pour remplacer l'homme dans les chaines d'assemblage. La robotique ludique est en plein essor avec des robots jouets tels que le chien *Aibo*[1] ou encore le robot *Nao*[2]. De plus en plus de robots manipulés sont utilisés dans le secteur médical lors d'opérations chirurgicales de précision afin de limiter les risques. Le développement de ces robots est également un objectif prioritaire dans le domaine militaire, avec les projets de développement de drones de surveillance, tel que le projet européen nEUROn, ainsi que dans le domaine de l'exploration spatiale avec les robots d'exploration planétaire *Spirit* et *Opportunity*.

Un robot est un système mécanique, électronique et informatique, équipé de capteurs et d'effecteurs, développé dans l'optique d'interagir avec le monde qui l'entoure. L'autonomie d'un robot se caractérise par sa capacité à agir et se déplacer dans l'environnement avec une intervention humaine limitée voire nulle. Une pleine automie est nécessaire en l'absence de communication avec une

1. développé et commercialisé par Sony.
2. développé et commercialisé par Aldebaran Robotics.

station de commande. Dans ce cas, la capacité du robot à réagir par lui-même pour accomplir sa mission se traduit par la présence de fonctions de planification dans son architecture informatique. En effet, suivant les aléas de l'environnement, il peut être amené à calculer des déplacements différents de ceux qui étaient initialement planifiés ou à prévoir une nouvelle manière d'accomplir les objectifs de la mission.

En robotique mobile, deux types de planification sont identifiés : la planification de tâches et la planification de déplacement. La première, qui s'appuie sur un raisonnement symbolique, permet au robot de prévoir la manière de réaliser les tâches de la mission. La deuxième fait intervenir un raisonnement géométrique et est utilisée pour calculer les déplacements du robots durant la réalisation de la mission. Ces deux types de planification, bien qu'étant différents, sont liés par le fait que la réalisation d'une tâche peut impliquer un ou plusieurs déplacements du robot. Les recherches sur la planification hybride considèrent ce lien pour en déduire des méthodes intégrant les deux types de planification.

L'objectif de cette thèse est de proposer puis de valider une architecture de planification hybride pour la robotique mobile permettant d'intégrer harmonieusement un planificateur de tâches et un planificateur de déplacements. Cette architecture s'appuie sur le principe d'entrelacement de l'exécution des deux planificateurs.

Le premier chapitre permet de cadrer ce travail en posant son contexte robotique et en présentant l'état de l'art concernant les architectures robotiques, la planification de tâches, la planification de déplacements ainsi que la planification hybride.

Dans le second chapitre, des questions relatives à l'interaction entre les planificateurs sont abordées expérimentalement. Ces questions portent avant tout sur l'intérêt de la démarche consistant à ne pas essayer de traiter l'intégralité du problème avec uniquement un planificateur de tâches mais de déléguer les calculs géométriques à un module spécialisé. Elles portent également sur des choix concernant la chronologie et le contenu des interactions entre les deux planificateurs.

En se basant sur les résultats expérimentaux, les concepts relatifs à une architecture hybride sont proposés dans le troisième chapitre. La caractéristique principale de cette architecture est de considérer les déplacements comme des préconditions et des effets spécifiques des opérateurs de planification de tâches. Ceci conduit à une algorithmique particulière au niveau de l'application d'une action, à des échanges spécifiques entre les deux planificateurs et à des problèmes de prise en compte des préconditions géométriques par le planificateur de déplacements.

Le quatrière chapitre est consacré à une démonstration de la faisabilité du concept d'architecture. Cette démonstration passe par la mise en œuvre des principes développés au chapitre précédent à l'aide d'un planificateur de tâches hiérarchique et d'un planificateur de déplacements qui s'appuie sur un algorithme probabiliste incrémental appelé *Rapidly-exploring Random Trees*. Dans ce cadre, une amélioration de la méthode, par décomposition de l'environnement puis restriction à un corridor, est proposée.

Finalement, le cinquième chapitre présente une validation de la mise en œuvre de l'architecture développée. Cette validation porte, d'une part, sur des modules : planificateur de déplacements et prises en compte des préconditions spécifiques par celui-ci et, d'autre part, sur l'architecture complète : présentation de la manière de décrire des comportements particuliers et calculs de plans pour différentes missions.

Chapitre 1

Planification et robotique

La robotique est le domaine d'application par excellence de la planification. En effet, un robot est un mécanisme ayant des capteurs lui permettant de percevoir son environnement puis d'en construire une représentation et des effecteurs lui permettant d'interagir avec cet environnement. À un niveau abstrait,ce mécanisme met en correspondance les informations de l'environnement perçues à l'aide de ses capteurs avec les actions qu'il peut effectuer à l'aide de ses effecteurs. La planification permer de décider des actions que le robot doit entreprendre. La figure 1.1 indique qu'elle produit un ensemble d'actions à partir de connaissances dont certaines sont relatives aux observations.

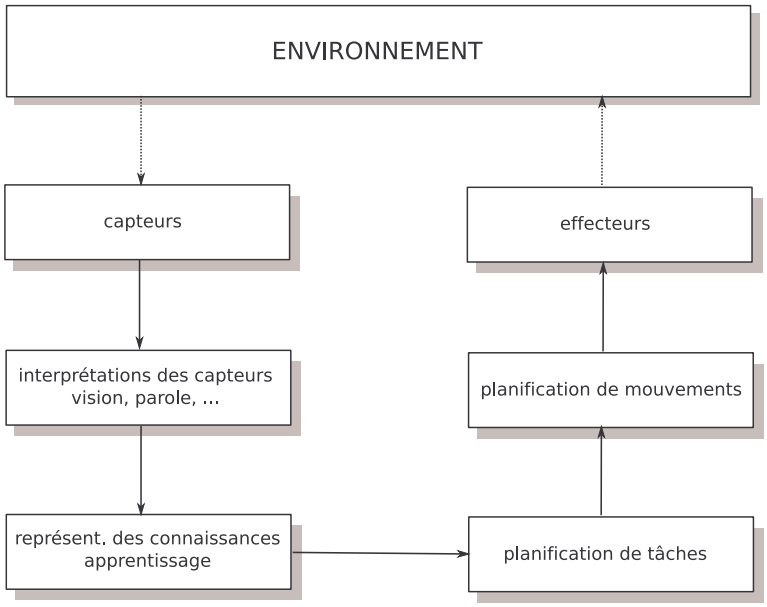


FIGURE 1.1 – Exemple d'architecture logicielle d'un robot

Dans ce chapitre, nous présentons différentes architectures robotiques pour lesquelles un module de planification est souvent nécessaire. Puis, après avoir défini le terme *planification*, nous abordons les travaux relatifs à ce domaine qui sont pertinents pour une application à la robotique mobile. Comme le montre la figure 1.1 deux types de planification peuvent intervenir : la planification de tâches et la planification de mouvements.

Nous présentons donc la planification de tâches qui correspond au choix des actions que l'agent devra entreprendre afin d'atteindre un but ou d'effectuer une tâche. Puis nous exposons les principales méthodes de planification de mouvements. Celles-ci interviennent dans le calcul des déplacements de l'agent dans l'environnement afin de réaliser les actions permettant d'accomplir sa mission. Finalement, nous nous intéressons aux recherches portant sur l'interaction entre ces deux types de planification.

1.1 Architectures robotiques

Les robots sont des systèmes complexes faisant intervenir différents traitements : traitement géométrique, gestion des capteurs, décision des actions à effectuer, boucles de commande... On appelle *architecture robotique* l'ensemble des modules réalisant ces traitements, ainsi que l'ensemble des modalités d'interaction reliant ces composants.

D'après Ingrand [2003], l'architecture logicielle mise en œuvre lors de la conception d'un robot autonome doit avoir les propriétés suivantes :

- **Programmabilité.** Elle doit permettre aux robots d'être des machines facilement programmables ;
- **Intégration.** Elle doit offrir des mécanismes de communication et d'échanges de données transparents afin de favoriser l'intégration de nouveaux modules. Cet ajout de modules ne doit pas remettre en cause le reste de l'architecture ;
- **Autonomie et cohérence.** Le comportement du robot doit être guidé par ses objectifs ;
- **Réactivité.** Les différents composants doivent être capable de réagir dans un délai approprié aux signaux qu'ils reçoivent ;
- **Robustesse.** Elle doit permettre d'exploiter la redondance des sources d'information et des modules de traitement ;
- **Sûreté.** Elle doit garantir certaines propriétés de sûreté pour les applications ou les situations critiques ;

1.1.1 Les différents types d'architecture

Les premières architectures de contrôle d'un robot étaient composées de trois éléments fonctionnels : un système de perception, un système de planification et un système d'exécution. On parlait alors d'approche *Sense-Plan-Act* (SPA) [Nilsson, 1982]. La figure 1.2 donne un exemple d'une telle architecture.

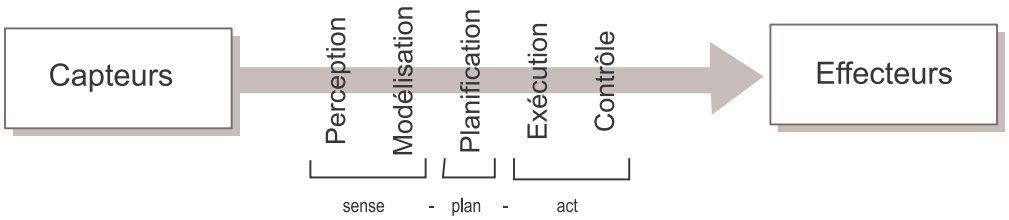


FIGURE 1.2 – Architecture de type SPA

Le rôle du système de perception est de capturer puis de transformer les données reçues par les capteurs en un modèle du monde. Puis, le planificateur génère un plan à partir de ce modèle et d'un objectif. Enfin, le système d'exécution génère les actions correspondantes au plan et contrôle leur exécution.

Cependant, un tel modèle d'architecture pose un problème de réactivité. En effet, l'intelligence du système repose uniquement sur le planificateur et non sur le système d'exécution. En cas d'échec du plan ou d'erreur d'exécution, il n'y a pas de remontée d'information. Un nouveau cycle perception - planification - exécution doit être entamé avant d'espérer palier à l'échec. La réactivité est donc limitée par le plus lent des modules de la chaîne, souvent le planificateur.

À partir des années 1985, du fait de la complexification des missions et de la représentation de l'environnement, ce type d'architectures a été abandonné. Depuis, la définition de nouvelles architectures robotiques a suivi deux approches radicalement différentes : l'approche réactive et l'approche délibérative.

L'approche réactive, ou approche ascendante, couple directement les capteurs aux effecteurs, c'est-à-dire qu'il n'y a pas de prise de décision à haut niveau. Ce type d'architecture correspond à un ensemble de comportements réactifs indépendants et permet une réponse en temps réel aux aléas et opportunités rencontrés. Cependant, du fait du manque d'information globale concernant l'environnement les actions produites peuvent être incohérentes entre elles.

L'approche délibérative, ou approche descendante, utilise un modèle global de l'environnement afin de générer la séquence d'actions la plus appropriée permettant au robot de réaliser un but spécifique. Cependant elle offre une

faible réactivité aux aléas et peut nécessiter une modélisation complexe de l'environnement. La plupart des architectures développées sont des architectures délibératives.

L'architecture de subsomption

Parmi les architectures réactives, la plus connue est l'architecture de subsomption [Brooks, 1986]. Elle a été présentée à l'origine comme une extension du modèle SPA concernant la notion de couches successives mais diffère radicalement car elle substitue à la notion de décomposition horizontale des traitements une décomposition verticale en niveaux de compétence.

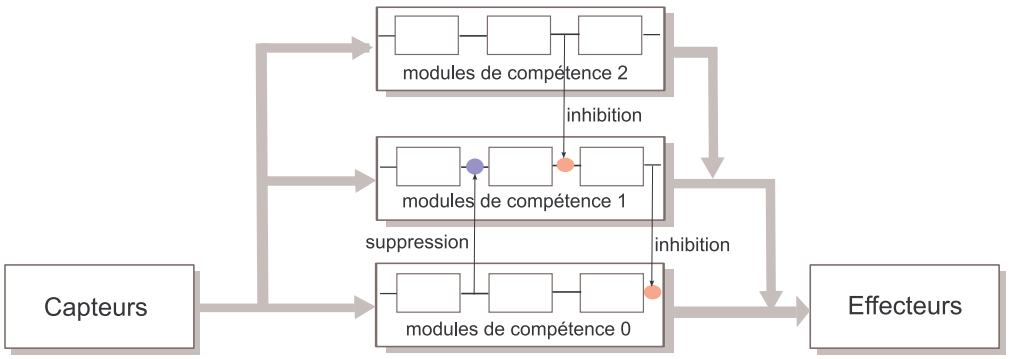


FIGURE 1.3 – Architecture de subsomption

La figure 1.3 présente l'architecture avec ces différents niveaux de commande qui sont gérés par des niveaux de priorité. Ainsi, le robot peut réagir très rapidement à un aléa de l'environnement. S'il dispose de plus de temps pour décider de la prochaine action à effectuer, alors les niveaux de priorité plus faible sont activés. La capacité à court-circuiter l'exécution d'un module d'un niveau par l'exécution d'un module d'un autre niveau est appelée *suppression* si elle a lieu en entrée et substitue les messages, et *inhibition* si elle a lieu en sortie de traitement et empêche la propagation des messages.

La simplicité de cette architecture facilite la mise au point des différents niveaux car ceux-ci sont relativement indépendants. Cependant, cette indépendance entre niveau est également le principal défaut de cette architecture : la modélisation effectuée à un niveau ne peut pas être réutilisée à d'autres niveaux. Par ailleurs, la mise en œuvre d'une telle architecture ne permet pas l'optimisation de la mission du fait de prises de décisions très locales. En effet, le robot a une vision de courte durée sur la résolution du problème et ne maintient pas une représentation globale de l'environnement, il ne peut pas toujours choisir la meilleure action à exécuter à un certain moment.

Dans l'application présentée par Brooks [Brooks, 1986] pour illustrer cette architecture, le niveau de comptétence le plus élevé comprend un module de planification de chemin simpliste. Ce module traduit un but, spécifié en termes de direction et de distance à parcourir, en une séquence de caps.

Les architectures trois-tiers

Les architectures trois-tiers [Gat, 1997] sont une réponse aux reproches faits aux architectures réactives. Elles ont pour objectif de favoriser l'autonomie des robots lors de l'exécution de tâches complexes nécessitant des capacités de raisonnement élévées. Elles sont composées de trois niveaux permettant de traiter le problème selon différentes granularités. Le niveau le plus haut est dédié aux capacités décisionnelles concernant la mission, le niveau le plus bas est chargé de l'exécution des actions et de faire le lien avec la couche physique du robot. Le niveau intermédiaire peut varier d'une architecture à une autre mais est généralement chargé de faire le lien entre les décisions prises à haut niveau et la réalisation de ces décisions, c'est-à-dire gérer et contrôler l'exécution du plan.

Parmi les premières architectures trois-tiers, on peut citer 3T [Bonasso and Kortenkamp, 1996], et ATLANTIS [Gat, 1992].

Ces deux architectures fonctionnent sur un modèle équivalent. Pour l'architecture 3T, les trois niveaux sont nommés, de bas en haut : niveau de connaissance, niveau de séquencement et niveau de planification. Pour l'architecture ATLANTIS, présentée sur la figure 1.4), les trois niveaux sont : le contrôleur, le séquenceur et le délibérateur.

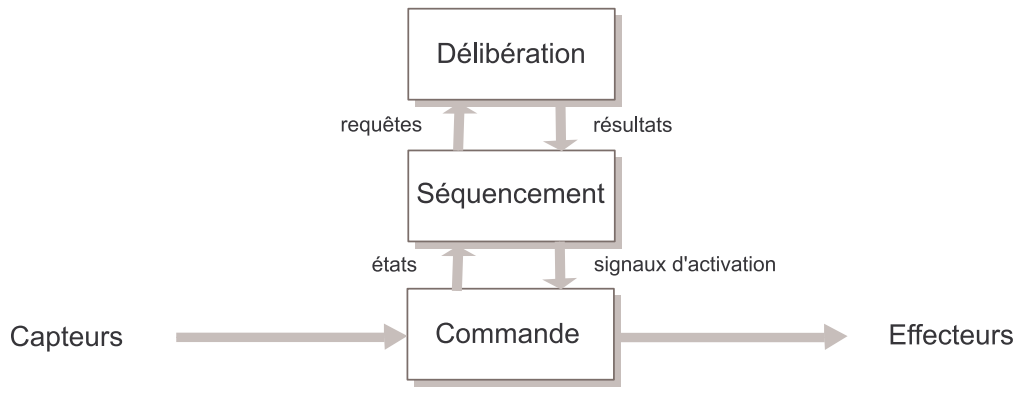


FIGURE 1.4 – L'architecture ATLANTIS

L'architecture LAAS [Alami *et al.*, 1998], du nom du laboratoire qui l'a développée, est une architecture trois-tiers originellement conçue afin d'offrir une

solution générique à la conception de robots autonomes. Les trois niveaux, présentés figure 1.5, sont nommés : niveau décisionnel, niveau contrôle d'exécution et niveau fonctionnel.

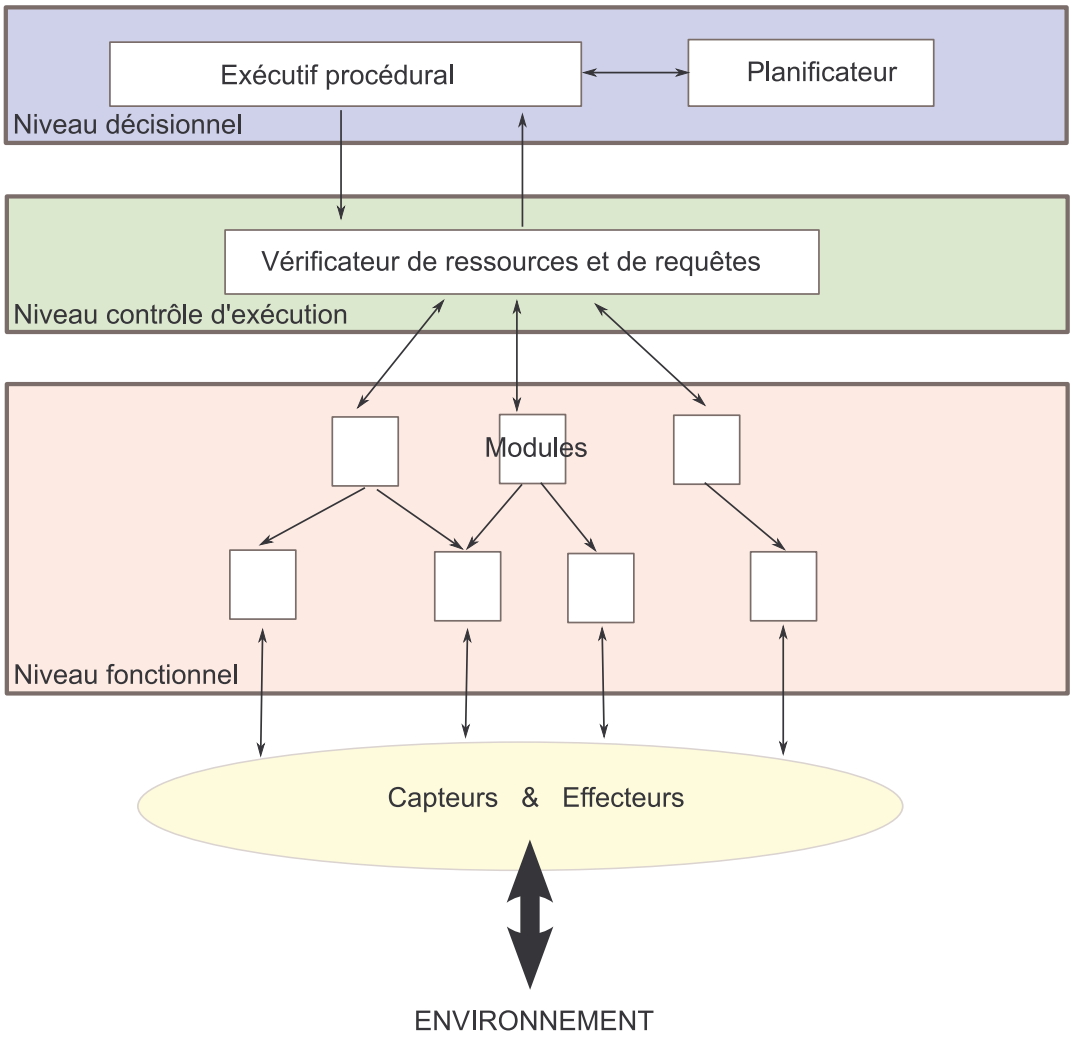


FIGURE 1.5 – L'architecture LAAS

La couche décisionnelle intègre les capacités délibératives du robot. Elle contient un planificateur chargé de produire des plans temporels qui prennent en compte les nouveaux buts et les échecs d'exécution renvoyés par un module exécutif chargé de s'assurer du bon déroulement de l'exécution du plan. Ce module exécutif est connecté au contrôleur d'exécution et lui envoie des requêtes d'exécution. Le module est également réactif aux évènements renvoyés par la couche inférieure ainsi qu'aux commandes de l'opérateur.

Le niveau contrôle d'exécution vérifie les requêtes envoyées aux modules fonctionnels et l'utilisation des ressources. Il a un rôle de filtre entre les décisions et les exécutions correspondantes.

La couche fonctionnelle, correspondant au niveau situé à la base de l'architecture, est l'interface entre les composants des couches supérieures et la partie physique du système. Ce niveau contient notamment les fonctions sensorimotrices, les fonctions de traitement, comme par exemple la planification de trajectoires et le traitement d'images, ainsi que les boucles de contrôles, comme par exemple la navigation et la gestion des capteurs.

Il est important de noter que ce type d'architecture s'accompagne d'une décomposition de la planification : planification de tâches au niveau décisionnel et planification de mouvements au niveau fonctionnel.

Ces architectures à trois niveaux ont certaines limitations [Estlin *et al.*, 2001] : Chaque niveau a sa propre représentation de l'environnement et du robot. Différents modèles doivent être créés et maintenus. De plus, chaque niveau est contraint à résoudre le problème à un degré de granularité spécifique. Cette structuration ne permet pas le choix des techniques de résolution les plus adaptées au problème courant. Finalement, ce type d'architectures ne permet pas la prise en compte des nouveaux travaux de recherche, dans le domaine de la planification et de l'exécution, visant à rapprocher ces deux niveaux.

CLARAty

L'architecture *Coupled Layer Architecture for Robotic Autonomy* (CLARAty) [Volpe *et al.*, 2001; Estlin *et al.*, 2001] est une alternative aux architectures à trois niveaux et a été développée dans le but d'offrir une solution aux reproches faits à celles-ci. Comme le montre la figure 1.6, CLARAty est une architecture à deux niveaux constituée d'une couche fonctionnelle en interaction directe avec une couche décisionnelle.

La couche fonctionnelle est une interface entre la couche décisionnelle et la couche physique du robot. Elle propose une décomposition modulaire et hiérarchique du système robotique à différents niveaux d'abstraction afin de permettre une facilité d'extension de ce système.

La couche décisionnelle comporte un planificateur ainsi qu'un exécutif faisant le lien entre le planificateur et la couche fonctionnelle. Elle contient également une sous-couche entre le planificateur et l'exécutif qui permet l'utilisation d'un langage commun et autorise ainsi l'envoi de requêtes d'exécution ayant différents niveaux de granularité.

Le planificateur global de la couche décisionnelle peut utiliser des planificateurs locaux appartenant à la couche fonctionnelle [Volpe *et al.*, 2000]. Par

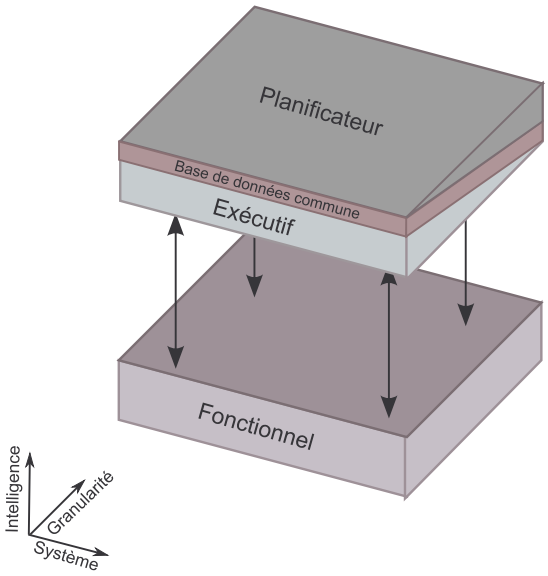


FIGURE 1.6 – L'architecture CLARAty

exemple, des planificateurs de chemins et de trajectoires attachés à des bras manipulateurs ou à des véhicules, qui donnent une capacité de planification précise des mouvements mais négligent l'optimalité globale. La décision d'utiliser ces planificateurs ou de traiter le problème lui-même est laissée au planificateur global.

Architectures multi-agent

Une autre manière de concevoir une architecture robotique est de considérer que les différents modules sont des entités autonomes qui coopèrent pour faire émerger le comportement désiré du robot : c'est l'approche multi-agent. La figure 1.7 illustre une possibilité de mise en œuvre d'une telle architecture.

Dans l'architecture *Distributed Architecture for Mobile Navigation* (DAMN) [Rosenblatt, 1995], chaque agent, appelé comportement, est responsable de la réalisation d'une tâche précise (éviter les obstacles, suivre la route, ...). Les agents expriment leur préférence sur chaque action possible. Un agent central, appelé arbitre, est en charge de la gestion de la communication entre les agents comportements et centralise les préférences pour décider de la meilleure action à entreprendre.

Dans l'architecture *Intelligent Distributed Execution Architecture* (IDEA) [Muscettola *et al.*, 2002; Finzi *et al.*, 2004], chaque agent, représentant les modules fonctionnels et décisionnels de l'architecture, repose sur la mise en œuvre d'un couple planificateur / système réactif. L'ensemble des agents est géré par un système de contrôle central.

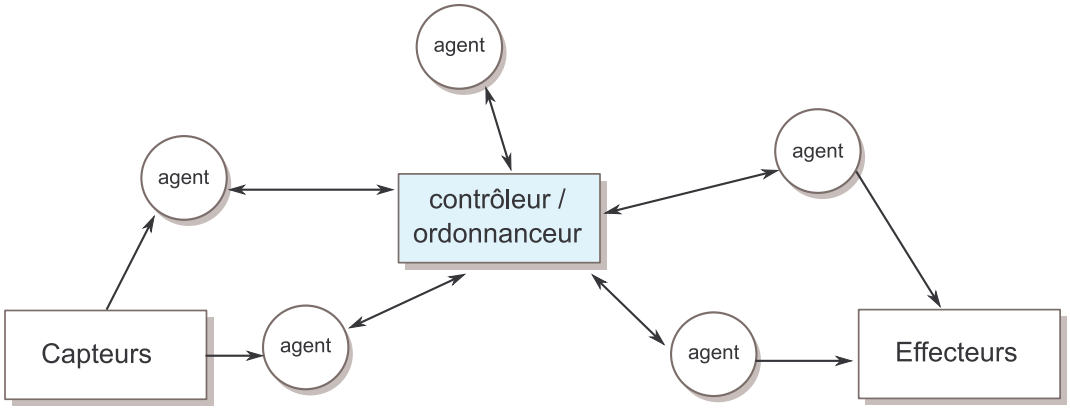


FIGURE 1.7 – Architecture multi-agent

1.1.2 Planification en robotique

La planification est un raisonnement basé sur un modèle visant à produire un plan permettant d'atteindre *a priori* un résultat désiré. Un plan est composé d'actions qui, lors de leur exécution, permettront d'obtenir le résultat désiré dans la mesure où les conditions d'exécution sont celles qui ont été modélisées.

Origine de la planification

Les travaux à l'origine de la planification, au sens de la définition ci-dessus, ont été initiés dans les années 1960 sous la forme de programmes dont l'objectif était de simuler les capacités de raisonnement de l'être humain. Le *General Problem Solver* (GPS) [Ernst *et al.*, 1969] fut l'un des travaux précurseurs en ce sens. GPS fonctionne par recherche dans un espace d'états et est guidé par le calcul des différences entre l'état courant et l'état final du problème. À la fin des années 1960, Green propose l'utilisation de prouveurs de théorèmes [Green, 1969]. Cependant, du fait de l'immaturité des techniques de preuves de théorèmes à cette époque, cette approche fut abandonnée au profit d'algorithmes de planification plus spécialisés. Ainsi en 1971 apparait le planificateur et langage de planification STRIPS [Fikes and Nilsson, 1971] qui, par la suite, donnera naissance à de nombreux autres langages et algorithmes de planification.

Plusieurs types de planification

Le terme *planification* est très général et peut prendre de nombreux sens, même en robotique. En effet, comme il existe de nombreux types d'actions, il existe de nombreux types de planifications, parmi lesquels, la planification de tâches, de mouvements, ou encore la planification pour la manipulation, pour la perception, pour la navigation...

La planification de tâches est un raisonnement abstrait visant à définir un plan d'actions pour la mission. La planification de mouvements est nécessaire pour permettre au robot de se déplacer dans l'environnement. Elle fait intervenir un raisonnement géométrique et nécessite des modèles détaillés du robot et de son environnement. La planification pour la manipulation est une forme de planification de mouvements qui vise à permettre au robot d'interagir avec son environnement, par exemple, saisir un objet. Le système autonome peut également avoir besoin de planifier ses actions de perception afin de modéliser l'environnement, identifier des objets ou encore localiser un objectif. On parle de planification pour la navigation lorsque la planification pour la perception est liée à la planification de mouvements et permet de résoudre des problèmes tels que se localiser, localiser et atteindre un objectif ou explorer une zone de l'environnement.

Afin de réaliser ces différentes planifications, deux approches sont envisageables : l'approche spécifique au domaine et l'approche indépendante du domaine. La première consiste à traiter chaque problème avec une représentation spécifique et des techniques adaptées au problème considéré. Cependant, les points communs entre les différents domaines de planification ne sont pas mis en commun. De plus, développer un planificateur pour chaque domaine à traiter devient vite couteux. Par ailleurs, l'utilisation de cette approche lors de la conception de l'architecture informatique d'un robot limite les capacités délibératives de celui-ci aux seules situations prises en compte par le planificateur utilisé. La seconde approche, l'approche indépendante du domaine, a pour but de fournir un cadre général à la résolution de problème. Pour résoudre un problème particulier, un planificateur prend en entrée les spécifications du problème ainsi que les connaissances concernant le domaine du problème.

Dans la suite de ce document, nous nous concentrons uniquement sur les aspects planification de tâches et planification de chemins pour un robot mobile.

1.1.3 Bilan

Les architectures robotiques intègrent systématiquement une fonctionnalité de planification. Cette fonctionnalité se réduit à la planification de mouvements pour les architectures basées sur une décomposition en compétences ou comportements. Pour les architectures s'appuyant sur la mise en évidence d'une couche fonctionnelle, on retrouve souvent une intégration de la planification de mouvements dans la couche fonctionnelle et une intégration de la planification de tâches dans la couche décisionnelle. Néanmoins, cette organisation n'est pas généralisée et la recherche d'une interaction efficace entre planification de tâches et de mouvements est un problème qui se pose pour un grand nombre d'architectures robotiques.

Cependant, les choix architecturaux en termes de lien entre la planification de tâches et la planification de mouvements dépendent également de la date de disponibilité de l'information sur l'environnement. Si celle-ci est postérieure à la date de calcul du plan de tâches, il est préférable que la planification de mouvements soit dans la couche fonctionnelle, en forte interaction avec les capteurs. À l'opposé, si au moment du calcul du plan de tâches les caractéristiques de navigabilité de l'environnement sont déjà connues, il peut être intéressant de connecter fortement les deux types de planification.

1.2 Planification de tâches

La planification de tâches est un raisonnement symbolique qui a pour objectif de construire un plan d'actions permettant, à partir d'un état initial, de résoudre un ensemble de buts en appliquant un ensemble d'actions.

1.2.1 Modèle conceptuel

Le modèle conceptuel de la planification peut être représenté comme un système de transition d'états, c'est-à-dire un quadruplets $\Sigma = (S, A, E, \gamma)$ où :

- $S = \{s_1, s_2, ...\}$ est un ensemble fini ou récursivement énumérable d'états ;
- $A = \{a_1, a_2, ...\}$ est un ensemble fini ou récursivement énumérable d'actions ;
- $E = \{e_1, e_2, ...\}$ est un ensemble fini ou récursivement énumérable d'évènements ;
- $\gamma : S \times A \times E \rightarrow 2^S$ est une fonction de transition entre états.

Ce système de transition d'états peut être représenté par un graphe direct dont les nœuds sont les états de S. Les arcs sont appelés transitions d'états.

La figure 1.8 montre un système de transition d'états impliquant un échantillon de roche ainsi qu'un *rover* qui peut transporter cet échantillon d'un endroit à un autre. Dans cet exemple :

- l'ensemble d'états est $S = \{s_0, s_1, s_2, s_3, s_4, s_5\}$;
- l'ensemble d'actions est $A = \{Take\_sample, Put\_sample, Goto\}$;
- l'ensemble d'évènements E est vide ;
- la fonction de transition γ est définie par : si a est une action et $\gamma(s, a)$ est non vide alors a est applicable à l'état s.

Le modèle conceptuel ainsi défini n'est pas directement opérationnel. En pratique, un certain nombre d'hypothèses restrictives sont émises, dont les principales sont :

- le système Σ a un ensemble fini d'états ;

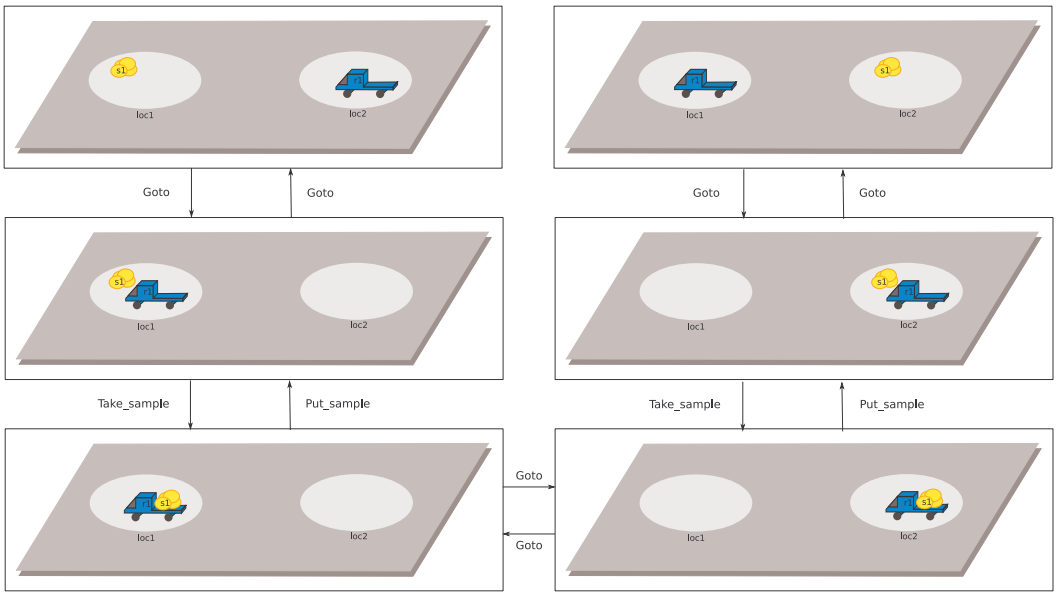


FIGURE 1.8 – Exemple de système de transition d'états

- le système Σ est totalement observable, *i.e.,* la connaissance concernant les états de Σ est complète ;
- le système Σ est déterministe, *i.e.,* si une action est applicable à un état, le résultat de cette application est un état unique. Autrement dit, $\gamma(s, a)$ est soit l'ensemble vide soit un singleton ;
- le système Σ est statique, *i.e.,* l'ensemble d'évènements E est vide. Σ n'a pas de dynamique interne ;
- les buts sont spécifiés sous la forme d'un ensemble d'états à atteindre ;
- un plan solution d'un problème de planification est une séquence d'actions linéaire, ordonnée et finie ;
- les actions n'ont pas de durée. Ce sont des transitions instantanées entre états ;
- la planification en cours n'est pas affectée par les changements pouvant intervenir dans le système de transition d'états.

1.2.2 Représentations et langages de planification

Une donnée nécessaire à tout algorithme de planification est la représentation du problème à résoudre. En pratique, il est impossible d'énumérer tous les états et les transitions entre états possibles : la description d'un problème peut être excessivement vaste, et la générer entièrement peut nécessiter plus de travail que la résolution du problème. Pour résoudre cette difficulté, il faut un langage de représentation du problème permettant de calculer ces états et ces transitions entre états à la volée.

Les principaux concepts du domaine de la planification sont introduits ici en utilisant le formalisme de la représentation classique qui utilise des notations dérivées de la logique du premier ordre : Un état est représenté sous la forme d'un ensemble d'atomes logiques qui sont vrais lorsque le système est dans l'état considéré. Les actions sont représentées par des opérateurs de planification qui changent la valeur de ces atomes.

Ainsi, soit L l'ensemble des atomes logiques pouvant être définis :

$$L = \{l_1, l_2, ..., l_{n_l}\}$$

On a S l'ensemble des états tel que chaque état est un sous-ensemble de L :

$$S \subseteq 2^L$$

Les états

Soit $\mathcal{L}$ un langage du premier ordre qui donc ne contient pas de symbole de fonctions. Un état est un ensemble d'atomes instantiés de $\mathcal{L}$, *i.e.,* qui ne contiennent pas de variable. Comme $\mathcal{L}$ ne contient pas de fonctions, l'ensemble S de tous les états est donc fini. Un prédicat p est vérifié dans un état s si et seulement si p peut être unifié avec des prédicats de s. On a alors $\sigma(p) \subseteq s$ où σ est la substitution résultant de l'unification de p avec le prédicat de s. Dans le cas contraire, la propriété du monde représentée par p est considérée comme fausse du fait de l'*hypothèse du monde clos* : un prédicat, qui n'est pas vérifié dans un état, est considéré comme faux dans cet état.

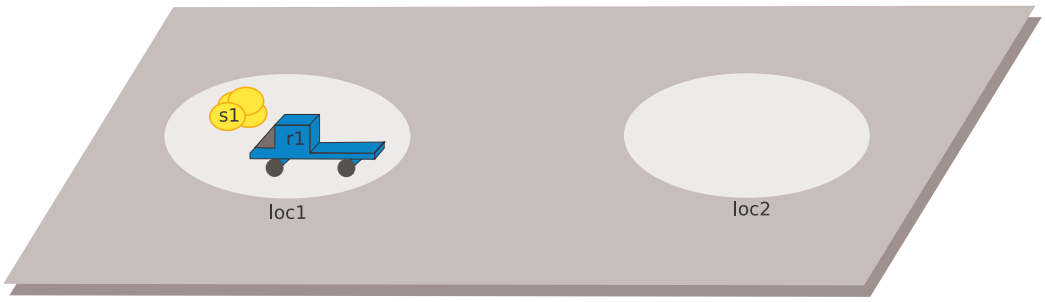


FIGURE 1.9 – Exemple de représentation d'un état : état s_1

$$s_1 = \left\{ \begin{array}{l} \mathsf{location}(loc1), \mathsf{location}(loc2), \\ \mathsf{rover}(r1), \mathsf{rock\_sample}(s1), \\ \mathsf{at}(s1, loc1), \mathsf{at}(r1, loc1), \mathsf{empty}(r1) \end{array} \right\}$$

La figure 1.9 illustre un exemple d'état.

Dans cet exemple, soient les prédicats p et p' :
$p : \exists r \mid \mathsf{rover}(r)$ est vérifié avec $\sigma : r1 \rightarrow r$, $\sigma\mathsf{rover}(r) = \mathit{preroverr1}$
$p' : \exists r \mid \mathsf{at}(r, \mathit{loc2})$ n'est pas vérifié.

Les opérateurs et actions

Une action est définie par l'application d'un opérateur de transformation. Le principe de la modélisation des actions par des opérateurs de transformation consiste à spécifier, à l'aide de prédicats, les propriétés du monde nécessaires à son application, *i.e.,* les préconditions de l'action, ainsi que les propriétés du monde engendrées par son exécution, *i.e.,* les effets de l'action. Les préconditions d'un opérateur représentent le domaine de définition d'une action, *i.e.,* l'ensemble des états dans lesquels l'action peut être appliquée. Ce calcul peut être réalisé en unifiant les préconditions positives ($\mathit{precond}^+$) de l'opérateur avec l'état courant et en vérifiant l'échec de l'unification des préconditions négatives ($\mathit{precond}^-$). Les préconditions positives expriment les propriétés qui doivent être vérifiées, les préconditions négatives expriment celles qui doivent être absentes de l'état pour que l'action soit appliquée. Les effets d'un opérateur spécifient les propriétés du monde modifiées par l'exécution d'une action. D'un point de vue formel, si une action est définie par un opérateur qui transforme un état s_i en un état s_{i+1}, les effets d'une action sont représentés par les prédicats ajoutées (effets^+) et les prédicats enlevées (effets^-) à s_i pour obtenir s_{i+1}.

Définition. 1.1 (Opérateur)
Un *opérateur* peut être défini par le triplet

$$o = (\mathit{nom}(o), \mathit{precond}(o), \mathit{effets}(o))$$

- $\mathit{nom}(o)$, le nom de l'opérateur, est défini par une expression de la forme $n(x_1, \ldots, x_k)$ où n est un *symbole d'opérateur* et $x_1, \ldots, x_k$ représentent les paramètres de l'opérateur.
- $\mathit{precond}(o)$ représente les préconditions de l'opérateur, *i.e.,* les propriétés du monde nécessaires à son exécution.
- $\mathit{effets}(o)$ définit l'ensemble des faits à ajouter et à supprimer de l'état après l'exécution de o.

Exemple d'opérateur. L'opérateur take_sample(r,s,l) peut être défini de la façon suivante :

;; take sample s at location l and load it into rover r
take_sample(r,s,l)
$\mathit{precond}$: rover(r), rock_sample(s), at(r,l), at(s,l), empty(r)
effets^+: loaded(s,r)
effets^-: empty(r), at(s,l)

Définition. 1.2 (Action)
Une *action* est une instanciation d'un opérateur. Si a est une action et s_i un état tel que $precond^+(a) \subseteq s_i$ et $precond^-(a) \cap s_i = \emptyset$, alors a est applicable à s_i, et le résultat de cette application est l'état :

$$s_{i+1} = \gamma(s_i, a) = (s_i - effets^-(a)) \cup effets^+(a)$$

Par exemple, comme le montre la figure 1.10, l'état résultant de l'application de l'action take_sample sur l'état s_1 est :

$$s_2 = \left\{ \begin{array}{l} \mathsf{location}(loc1), \mathsf{location}(loc2), \\ \mathsf{rover}(r1), \mathsf{rock\_sample}(s1), \\ \mathsf{at}(r1, loc1), \mathsf{loaded}(s1, r1) \end{array} \right\}$$

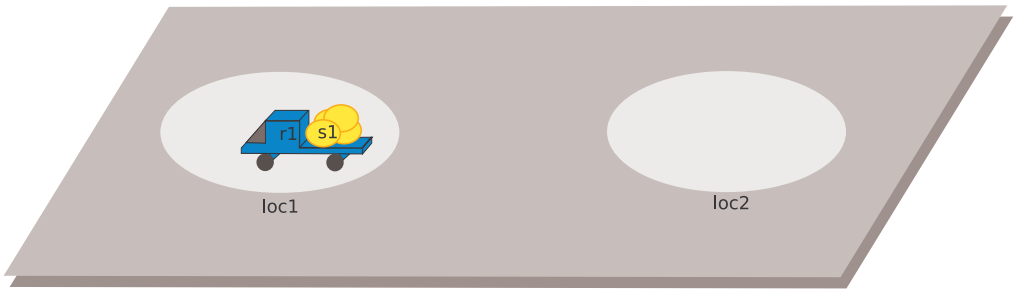


FIGURE 1.10 – Résultat de l'application de l'action take_sample($r1$,$s1$,$loc1$) sur l'état s_1

Les domaines et problèmes

En planification, un domaine décrit le cadre dans lequel va être résolu le problème. Il est défini par un ensemble d'opérateurs qui peuvent s'appliquer sur le monde. Un problème doit de plus spécifier l'état initial ainsi que les buts à résoudre. Un domaine peut être partagé par plusieurs problèmes.

Définition. 1.3 (Domaine de planification)
Un domaine $\mathcal{D}$ de L est un système de transition d'états restreint $\Sigma = (S, A, \gamma)$ tel que :

- $S = 2^{\{\text{les atomes instanciés de } L\}}$
- $A = \{$ l'ensemble des opérateurs instanciés de $O\}$ où O est l'ensemble des opérateurs
- $\gamma(s, a) = (s - effets^-(a)) \cup effets^+(a)$ si $a \in A$ et a est applicable à $s \in S$. La fontion γ étant la fonction de transition entre états.

Le domaine de planification est décrit par l'ensemble $\mathcal{O}$ des opérateurs applicables.

Définition. 1.4 (Problème de planification)
Un problème pour un domaine $\mathcal{D}$ est la paire (s_0, g) où :

- s_0, l'état initial, est un état quelconque de S ;
- g, le but, définit un ensemble cohérent de prédicats instanciés, *i.e.,* les propriétés du monde devant être atteintes ;

Ainsi, la spécification d'un problème de planification est $\mathcal{P} = (\mathcal{O}, s_0, g)$.

Exemple de but :

$$g = \{\ \mathsf{at}(r1, loc2), \mathsf{at}(s1, loc2)\ \}$$

Les plans.

Un plan solution se définit comme un chemin dans un espace d'états. Le passage d'un état à l'autre s'effectue en appliquant une action, *i.e.,* un opérateur instancié. Un plan solution est donc une séquence d'actions décrivant un chemin de l'état initial à l'état final.

Exemple de plan. Considérons le plan suivant :

$$\pi = \left\langle \begin{array}{l} \mathsf{Goto}(r1, loc1), \\ \mathsf{Take\_sample}(r1, s1, l1), \\ \mathsf{Goto}(r1, loc2), \\ \mathsf{Put\_sample}(r1, s1, l2) \end{array} \right\rangle$$

Ce plan est applicable à l'état s_0 et produit l'état s_4 (figure 1.11).
L'état initial s_0 avant exécution du plan est :

$$s_0 = \left\{ \begin{array}{l} \mathsf{location}(loc1), \mathsf{location}(loc2), \\ \mathsf{rover}(r1), \mathsf{rock\_sample}(s1), \\ \mathsf{at}(s1, loc1), \mathsf{at}(r1, loc1), \mathsf{empty}(r1) \end{array} \right\}$$

L'état final s_4 contenant le but g est :

$$s_4 = \left\{ \begin{array}{l} \mathsf{location}(loc1), \mathsf{location}(loc2), \\ \mathsf{rover}(r1), \mathsf{rock\_sample}(s1), \\ \mathsf{at}(s1, loc2), \mathsf{at}(r1, loc2), \mathsf{empty}(r1) \end{array} \right\}$$

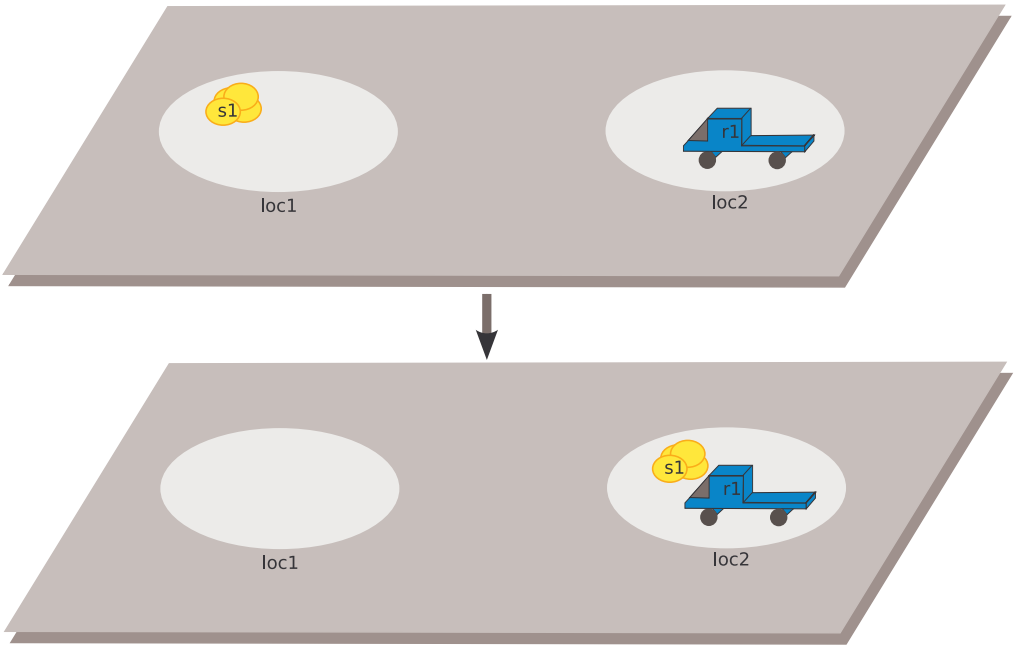


FIGURE 1.11 – Exécution du plan π permettant de passer de l'état s_0 à l'état final s_4

Langages de planification

Les trois principaux langages de planification permettant d'exprimer un domaine sont STRIPS [Fikes and Nilsson, 1971], ADL [Pednault, 1994] et PDDL [McDermott *et al.*, 1998].

Le langage STRIPS (*STanford Research Institute Problem Solver*) est considéré comme le premier langage de planification. C'est le langage utilisé en entrée du planificateur du même nom.

Le langage ADL (*Action Description Language*) est une extension du langage STRIPS. Il permet l'utilisation de quantificateurs et d'expressions conditionnelles dans la description des opérateurs.

Exemple de description d'un opérateur en ADL :

(**Action**
(Goto (r : rover, $loc1$: location, $loc2$: location))
(at(r, $loc1$))
(at(r, $loc2$), ∼at(r, $loc1$), ($\forall x$, loaded(x, r) ⇒ at(x, $loc2$), ∼at(x, $loc1$)))
)

Le langage PDDL (*Planning Domain Description Language*) a été développé pour servir de langage commun aux différents planificateurs lors de la

première compétition internationale de planification en 1998. PDDL propose certaines fonctionnalités telles que la déclaration préalable des prédicats utilisés ou encore le typage des paramètres.

Exemple de description d'un opérateur simple en PDDL :

(**:action** Goto
  (**:parameters** (?r - rover ?from ?to - location))
  (**:precondition** ((at ?r ?from))
  (**:effect** (and (at ?r ?to)(not (at ?r ?from))
)

Par la suite, de nombreuses extensions du langage ont été proposées, parmi lesquelles la prise en compte de la notion temporelle et des actions duratives [Fox and Long, 2003] ou encore la spécification de contraintes et préférences [Gerevini and Long, 2005]

1.2.3 Les ressources en planification

Une ressource est une donnée ayant un sens physique et qui est nécessaire à la réalisation d'une action. Les ressources font partie de l'état initial et sont produites ou consommées par les actions. On peut définir deux grandes familles de ressources : les ressources réutilisables et les ressources consommables.

Les ressources réutilisables sont des ressources dont l'état ne varie pas au cours de leur utilisation. Elles sont généralement identifiées par des prédicats booléens.

```
;; exemple de ressource réutilisable
   (has_camera rover1 cam1)
```

Les ressources consommables sont des ressources dont la quantité disponible varie au cours du temps. Elles sont représentées par des prédicats numériques et nécessitent une prise en compte de contraintes de capacités ainsi qu'une mise à jour de leur valeur suite à une utilisation.

```
;; exemple de ressource consommable
   (energy_level rover1 100)
```

Dans la littérature, différentes manières de gérer et de vérifier l'utilisation de ces ressources sont proposées. Le premier planificateur qui a été capable de gérer les ressources consommables est SIPE [Wilkins, 1988] mais il ne permet pas une représentation explicite des contraintes pour éviter les conflits. Dans le planificateur O-Plan [Currie and Tate, 1991], le respect des contraintes sur

les ressources est effectué en définissant des profils optimistes et pessimistes de consommation [Drabble and Tate, 1994]. Dans le planificateur IxTeT, la détection des conflits entre utilisations de ressources s'appuie sur un algorithme de recherche de cliques dans un graphe [Laborie and Gallab, 1995]. Cependant la plupart des travaux sur la gestion des ressources s'appuie sur des algorithmes de propagation de contraintes dans l'espace de plans ou d'actions [El-Kholy and Richards, 1996; Cesta and Stella, 1997; Laborie, 2003]. Du point de vue de l'expression des consommations et productions de ressources, [Koehler, 1998] étend le langage ADL afin d'identifier et d'exprimer les changements d'états des ressources. Actuellement, les travaux de recherche sur la gestion de ressources portent principalement sur le traitement des ressources continues [Biundo *et al.*, 2004].

1.2.4 Techniques et algorithmes de planification

Les algorithmes de planification peuvent être regroupés en deux grandes familles. D'un côté les algorithmes de planification dans un espace d'états qui sont les plus simples, de l'autre côté les algorithmes de planification dans un espace de plans.

Planification dans un espace d'états

Les algorithmes de planification dans un espace d'états sont des algorithmes de recherche dans lesquels l'espace de recherche est un sous-ensemble de l'espace d'états : chaque nœud correspond à un état du monde, chaque arc correspond à une transition entre deux états, et le plan courant correspond au chemin courant dans l'espace de recherche. Les principaux algorithmes de planification dans un espace d'états sont la recherche par chaînage avant, la recherche par chaînage arrière et l'algorithme STRIPS.

Dans un algorithme de recherche par chaînage avant, l'objectif est de trouver un état qui satisfait le but. L'algorithme prend en entrée un problème $\mathcal{P}$. Si le problème peut être résolu, alors un plan solution est renvoyé. Le plan produit à chaque invocation récursive est appelé plan partiel : c'est potentiellement une partie du plan solution qui sera produit par l'algorithme.

L'idée de la recherche par chaînage arrière est de partir du but et d'appliquer l'inverse des opérateurs de planification pour produire des sous-buts jusqu'à arriver à l'état initial. Si l'algorithme permet de remonter jusqu'à cet état initial, alors un plan solution est trouvé.

Le principal inconvénient des algorithmes de planification dans un espace d'états est la taille de l'espace de recherche. L'algorithme STRIPS développe un arbre dont la racine est l'état initial et le but. Les nœuds de l'arbre sont étiquetés par l'état courant et un ensemble de sous-buts à atteindre. Cet arbre permet de

mélanger chaînage arrière, par création de nœuds suivants comprenant le même état avec un sous-but supplémentaire et, chaînage avant, par création de nœuds suivants comprenant l'état résultat d'une action éliminant le premier sous-but de la liste, ainsi que cette liste réduite.

L'avantage de l'algorithme STRIPS réside dans sa capacité à réduire l'espace de recherche. Cette optimisation est caractérisée par les deux points suivants :

- à chaque appel récursif de l'algorithme, les sous-buts qui doivent être immédiatement satisfaits sont uniquement les préconditions de la dernière action considérée pour réduire l'écart entre le but et l'état courant. Ainsi le facteur de branchement de l'algorithme est réduit ;
- si l'état courant satisfait toutes les préconditions d'un opérateur, alors celui-ci est marqué et, en cas d'échec, le retour arrière se fera à partir de cet opérateur. Les actions précédemment planifiées ne sont pas remises en cause.

Planification dans un espace de plans

Les algorithmes de planification dans un espace de plans travaillent sur un espace de recherche plus sophistiqué : les nœuds représentent des plans partiellement spécifiés et les arcs sont des opérations de raffinement du plan, *i.e.,* ils permettent de réaliser un but ouvert ou d'enlever une inconsistance potentielle (par exemple lorsque deux actions sont en conflit). La représentation de plans partiellement spécifiés peut conduire à utiliser une structure de plan plus générale que la séquence. Un plan est alors défini comme étant un ensemble d'opérateurs reliés par des contraintes d'ordre et des contraintes d'instanciation. Une contrainte d'ordre est, par exemple, « l'action A doit s'exécuter avant l'action B ». Un exemple de contraintes d'instanciation est « le robot r qui intervient dans l'action A est le même que celui qui intervient dans l'action B ». Mais les contraintes d'ordre ne sont pas suffisantes. En effet, rien n'indique que les effets produits par l'action A soient toujours valables lors de l'exécution de l'action B. Par exemple, une action C s'exécutant parallèlement peut modifier l'état courant, ainsi les préconditions nécessaires à l'exécution de l'action B peuvent ne plus être disponibles. Pour garantir le plan, il est nécessaire d'ajouter des liens causaux, *i.e.,* des liens qui définissent les relations entre deux actions en termes de producteur / consommateur. Ainsi les prédicats de l'état courant qui ont été produits par l'action A ne pourront être consommés que par l'action B. Afin de limiter la complexité, la recherche dans un espace de plans peut être guidée par des heuristiques.

Parmi les planificateurs travaillant dans un espace de plans, on peut citer le planificateur IxTeT [Laborie, 1995; Laborie and Ghallab, 1995]. IxTeT génère un plan solution par raffinements successifs du problème initial. Il génère un plan partiel puis détecte les défauts de ce plan et calcule l'ensemble des contraintes qu'il faudrait rajouter au plan pour les résoudre. Ceci conduit à un nouveau

plan partiel pour lequel le processus est itéré. Le plan partiel courant est un plan solution lorsqu'il ne contient plus aucun défaut.

Autres techniques

Afin d'optimiser le processus de planification, des variantes des algorithmes précédents ont été proposées. Par exemple, la planification dans un graphe, la planification à partir de cas ou encore la méta-planification.

Parmi les planificateurs utilisant comme support un graphe, le plus connu est GraphPlan [Blum and Furst, 1997]. Le planificateur GraphPlan fonctionne en deux phases.

Tout d'abord, il construit un graphe appelé *graphe de planification* qui permet de déterminer les états atteignables à partir de l'état initial et des opérateurs disponibles. Ce graphe est structuré en niveaux qui alternent successivement des nœuds étiquetés par des propositions et des nœuds étiquetés par des actions (figure 1.12). La construction du graphe se termine lorsqu'un niveau contient tous les prédicats du but. L'algorithme conserve dans le graphe de planification le problème initial grâce à un mécanisme d'exclusion mutuelle entre certains nœuds du graphe.

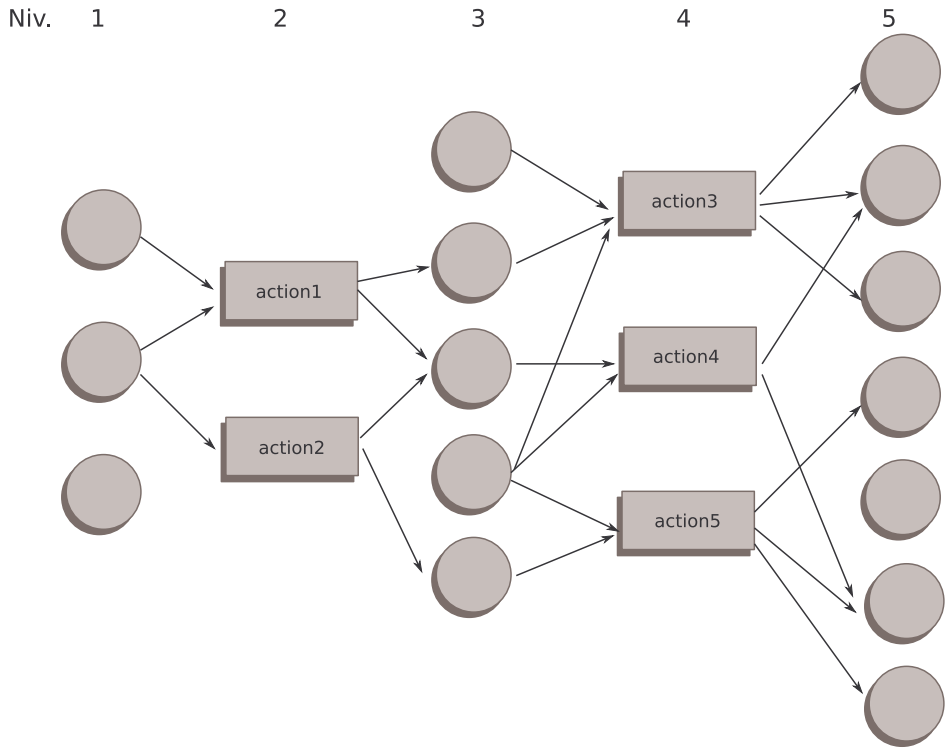


FIGURE 1.12 – Construction du graphe de planification

Dans la deuxième phase, appelée phase d'extraction, l'algorithme parcourt le graphe de planification à la recherche d'un plan solution. Il procède par chaînage arrière à partir de l'état final et remonte le graphe jusqu'à l'état initial en choisissant, à chaque niveau d'actions, une action compatible avec les actions précédemment sélectionnées.

La planification à partir de cas [Hammond, 1986] utilise les expériences passées pour construire de nouveaux plans. Cette technique consiste à mémoriser, pour chaque but rencontré, les plans permettant de les satisfaire ainsi que les échecs obtenus. Ainsi, le planificateur peut anticiper et éviter les échecs lors de la construction de nouveaux plans en tenant compte des plans précédemment obtenus.

La méta-planification [Stefik, 1981] revient à planifier la planification, c'est-à-dire qu'elle consiste à utiliser les connaissances sur la manière de planifier de la même façon que celles utilisées lors du processus de planification classique. Ce type de processus de planification est défini en trois niveaux. Le plus haut niveau, le niveau stratégique, détermine la stratégie de conception du plan. Le niveau intermédiaire, appelé niveau de conception, raffine le plan abstrait en résolvant les interactions entre buts. Enfin le niveau domaine fournit les opérateurs et les données du domaine de planification.

Dans les paragraphes précédents, nous avons représenté un problème de planification sous une forme dite "classique". Cependant, un tel problème peut être traduit dans un autre formalisme et résolu par une technique associée à ce formalisme. Par exemple, la satisfaction de contraintes ou encore la satisfaction de clauses propositionnelles.

Un problème de satisfaction de contraintes (CSP) est défini par n variables discrètes prenant leur valeur dans des domaines finis et par m contraintes spécifiant les relations entre ces variables. Une solution à un tel problème est une assignation, à chaque variable, d'une valeur satisfaisant toutes les contraintes. Pour effectuer la traduction vers ce formalisme, on peut par exemple associer une variable booléenne à chaque nœud du graphe de planification.

Un problème de satisfaction de clauses propositionnelles (SAT) est représenté à l'aide de formules propositionnelles sous forme normale conjonctive, c'est-à-dire une conjonction de clauses dont chacune est une dijonction de littéraux. L'objectif est de trouver une assignation de valeurs booléennes afin de satisfaire la formule.

1.2.5 Planification par recherche heuristique

Les heuristiques sont utilisées en planification de tâches pour accélérer le processus de recherche d'une solution. On parle de planification par recherche heuristique [Bonet and Geffner, 2001]. Dans ce type de planification, le problème

est souvent relaxé, *i.e.,* les effets négatifs des actions sont ignorés, puis un plan est recherché pour le problème relaxé. L'objectif est de calculer la distance au but depuis l'état initial, *i.e.,* connaitre a priori la longueur du plan solution. Parmi les planificateurs symboliques utilisant ce principe d'heuristiques, on peut citer le planificateur HSP [Bonet and Geffner, 2000] et le planificateur FF [Hoffmann and Nebel, 2001].

1.2.6 Planification hiérarchique

La notion de hiérarchie en planification remonte aux années 1970. Le planificateur ABSTRIPS [Sacerdoti, 1974] utilise pour la première fois plusieurs niveaux d'abstraction lors du processus de planification pour limiter l'espace de recherche. À chaque niveau d'abstraction, le planificateur effectue son choix parmi un nombre restreint d'opérateurs et de faits. Des sous-buts sont définis et résolus en fonction de l'importance qui leur est accordé. Dans le planificateur NOAH [Sacerdoti, 1975], ce sont les opérateurs de planification qui sont hiérarchisés. On parle alors de macro-opérateurs. À partir des années 1990, le terme planification hiérarchique fait référence à la planification HTN (*Hierarchical Task Network*) dans laquelle le domaine de planification spécifie la façon de réaliser les buts. Les planificateurs HTN les plus connus sont UCMP [Erol *et al.*, 1994], SHOP [Nau *et al.*, 1999] puis SHOP2 [Nau *et al.*, 2003].

La planification hiérarchique utilise un langage similaire à celui de la planification classique : chaque état du monde est représenté par un ensemble de prédicats instanciés et chaque action correspond à une transition déterministe entre deux états. Cependant la planification hiérarchique diffère de la planification classique sur l'objectif.

Dans un planificateur HTN, l'objectif n'est pas d'atteindre un ensemble de buts mais de résoudre un ensemble de tâches. En plus de l'ensemble d'opérateurs, le domaine est constitué d'un ensemble de méthodes.

Les méthodes

Une méthode peut être vue comme une *recette* permettant de décrire la manière de décomposer une tâche en un ensemble de sous-tâches. Le principe de la planification hiérarchique consiste à décomposer les tâches non primitives en sous-tâches de plus en plus petites jusqu'à obtenir des tâches primitives qui peuvent être résolues en appliquant les opérateurs de planification du domaine.

Définition. 1.5 (Méthode)
Une *méthode* m est définie par un ensemble de décompositions :

$$m = (name(m), < decomposition_1(m), ..., decomposition_n(m) >)$$

Une décomposition est une paire :

$$decomposition_d(m) = (precond(d), reduction(d))$$

- $name(m)$ est une expression de la forme $n(x_1, ..., x_k)$ où n représente le nom de la méthode et $x_1, ..., x_k$ ses paramètres.
- $precond(d)$ représente les préconditions (*i.e.,* un ensemble de prédicats) devant être vérifiées dans l'état courant pour que la décomposition d de m soit appliquée.
- $reduction(d)$ définit la séquence de sous-tâches à réaliser pour réaliser m. Chaque sous-tâches est soit une méthode, soit un opérateur.

Exemple de méthode. La méthode move_sample(s,x,y) peut être définie de la façon suivante :

;; move a sample s from location x to location y (with a rover r)
move_sample(s,x,y)
precond: at(s,x), at(r,x)
reduction: Take_sample(r,s,x), Goto(r,y), Put_sample(r,s,y)

Un problème de planification hiérarchique peut s'exprimer sous la forme :

$$\mathcal{P}_H = (\mathcal{O}, \mathcal{M}, s_0, < t_1, ..., t_n >)$$

Où $\mathcal{O}$ est l'ensemble des opérateurs, $\mathcal{M}$ est l'ensemble des méthodes, s_0 est l'état initial et $< t_1, ..., t_n >$ est la liste des tâches à décomposer.

Expressivité de la planification hiérarchique

Tout problème de planification classique peut être exprimé sous la forme d'un problème de planification hiérarchique en un temps polynomial. Par ailleurs, il existe des problèmes de planification hiérarchique qui ne peuvent pas être traduits en problème de planification classique [Erol *et al.*, 94].

1.2.7 Bilan

La planification de tâches peut être effectuée à l'aide d'une grande variété d'algorithmes dont certains construisent le plan par chaînage avant et donc dans le sens de l'exécution. Avec ces algorithmes et lorsque le plan implique des déplacements, ceux-ci peuvent être calculés durant le processus de planification de tâches.

1.3 Planification de déplacements

En robotique, on appelle planification de mouvements le problème du calcul préalable des mouvements nécessaires à un robot pour accomplir une tâche donnée. Dans sa forme la plus générale, la planification de mouvements se définit de la façon suivante : étant donné un modèle du système robotique et de son environnement, planifier un mouvement consiste à calculer le mouvement que doit effectuer le système pour atteindre un objectif fixé a priori.

La planification de chemins, appelée également planification de déplacements, est un sous-problème de la planification de mouvements. Elle prend uniquement en compte les aspects géométriques du système robotique et de son environnement. Planifier un chemin consiste à définir une séquence de configurations définissant un chemin sans collision entre une configuration initiale et une configuration finale en respectant les contraintes cinématiques du robot considéré.

1.3.1 Quelques définitions

Un problème de planification de chemins fait intervenir un certain nombre de notions définies ici.

Soit $\mathcal{A}$ un object non déformable, le robot, se déplaçant dans un environnement modélisé à l'aide d'un espace euclidien $\mathcal{W}$ défini sur $\mathbb{R}^N$ avec $N = 2$ ou 3 suivant que l'environnement est représenté en deux ou trois dimensions. Soit $\mathcal{B}_1, ..., \mathcal{B}_n$ des objets rigides et fixes distribués dans $\mathcal{W}$. Ces n objets, appelés obstacles, correspondent à des zones de l'environnement qui ne sont pas traversables.

Définition. 1.6 (Configuration)
Une configuration q d'un objet mobile est une spécification de tous les points de cet objet relativement à un référentiel fixé. De manière générale, la configuration correspond à une position et à une orientation de $\mathcal{A}$ dans $\mathcal{W}$. Ainsi une configuration q pour un robot terrestre $\mathcal{A}$ évoluant en deux dimensions est une spécification de la position (x, y) du centre d'inertie du robot et de son orientation θ.

Définition. 1.7 (Problème)
Un problème de planification de chemins est défini de la manière suivante : Connaissant une configuration initiale q_{init} de $\mathcal{A}$ et une configuration finale q_{goal}, définir un chemin paramétré par τ, spécifiant une séquence continue de configurations $q(\tau)$ permettant au robot $\mathcal{A}$ de se déplacer de la configuration initiale à la configuration finale en évitant les n obstacles.

Définition. 1.8 (Espace de configuration)
L'espace de configuration de $\mathcal{A}$ est l'espace $\mathcal{C}$ de toutes les configurations de $\mathcal{A}$. Raisonner dans $\mathcal{C}$ permet de transformer le problème de planification du déplacement d'un objet dans $\mathcal{W}$ en un problème de planification du déplacement d'un point.

Définition. 1.9 (C-Obstacle)
Chaque obstacle $\mathcal{B}_i$, $i = 1$ à n de l'espace $\mathcal{W}$ correspond dans $\mathcal{C}$ à une région

$$\mathcal{CB}_i \;=\; \{\; q \;\in\; \mathcal{C} \mid \mathcal{A}(q) \;\cap\; \mathcal{B}_i \;\neq\; \emptyset \;\} \tag{1.1}$$

appelée $\mathcal{C}$-Obstacle. L'union de tous les $\mathcal{C}$-Obstacles est appelée l'espace des $\mathcal{C}$-Obstacles et notée $\mathcal{C}obs$.

$$\mathcal{C}obs \;=\; \bigcup_{i=1}^{n} \mathcal{CB}_i \tag{1.2}$$

Définition. 1.10 (Zone libre)
Une zone libre est une zone dans laquelle le robot peut se déplacer, *ie.* une zone dans laquelle les configurations sont possibles. On note $\mathcal{C}free$ l'espace des configurations possibles :

$$\mathcal{C}free \;=\; \mathcal{C} \setminus \mathcal{C}obs \;=\; \mathcal{C} \setminus \bigcup_{i=1}^{n} \mathcal{CB}_i \tag{1.3}$$

Définition. 1.11 (Chemin solution)
Un chemin entre deux configurations q_{init} et q_{goal} est solution du problème de planification si et seulement si il existe une fonction continue :

$$\tau : [0,1] \xrightarrow{q(\tau)} \mathcal{C}free \tag{1.4}$$

avec $q(0) = q_{init}$ et $q(1) = q_{goal}$. C'est-à-dire qu'un chemin est solution s'il est sans collision.

Définition. 1.12 (Contrainte cinématique)
Une contrainte cinématique est une contrainte qui réduit la liberté de mouvement du robot considéré, c'est-à-dire qui fait que, même sans obstacle, les variables de configuration ne sont pas totalement libres et indépendantes. Une contrainte peut servir à exprimer, directement ou après intégration, un paramètre de configuration en fonction des autres et donc à réduire la dimension de l'espace de configuration. Par exemple, la contrainte

$$z \;=\; z_{sol}(x, y)$$

permet d'éliminer l'altitude z de q en exprimant que le robot est toujours en contact avec le sol. Dans ce cas, la contrainte est qualifiée de holonome. Une contrainte cinématique non-holonome est une relation non intégrable sur les paramètres de configuration du système ainsi que sur leurs dérivés. Ce type de contraintes survient généralement lorsque le système présente moins de paramètres de commande que de paramètres de configuration. Par exemple, la contrainte

$$\dot{x}sin\theta \; - \; \dot{y}cos\theta \; = \; 0$$

est une contrainte non-holonome qui signifie que le robot se déplace sans glissement. En planification de chemin, la non-holomie des contraintes oblige à calculer des chemins qui sont non seulement sans collision mais aussi cinématiquement admissibles, *i.e.,* qui vérifient les contraintes non-holonomes du robot.

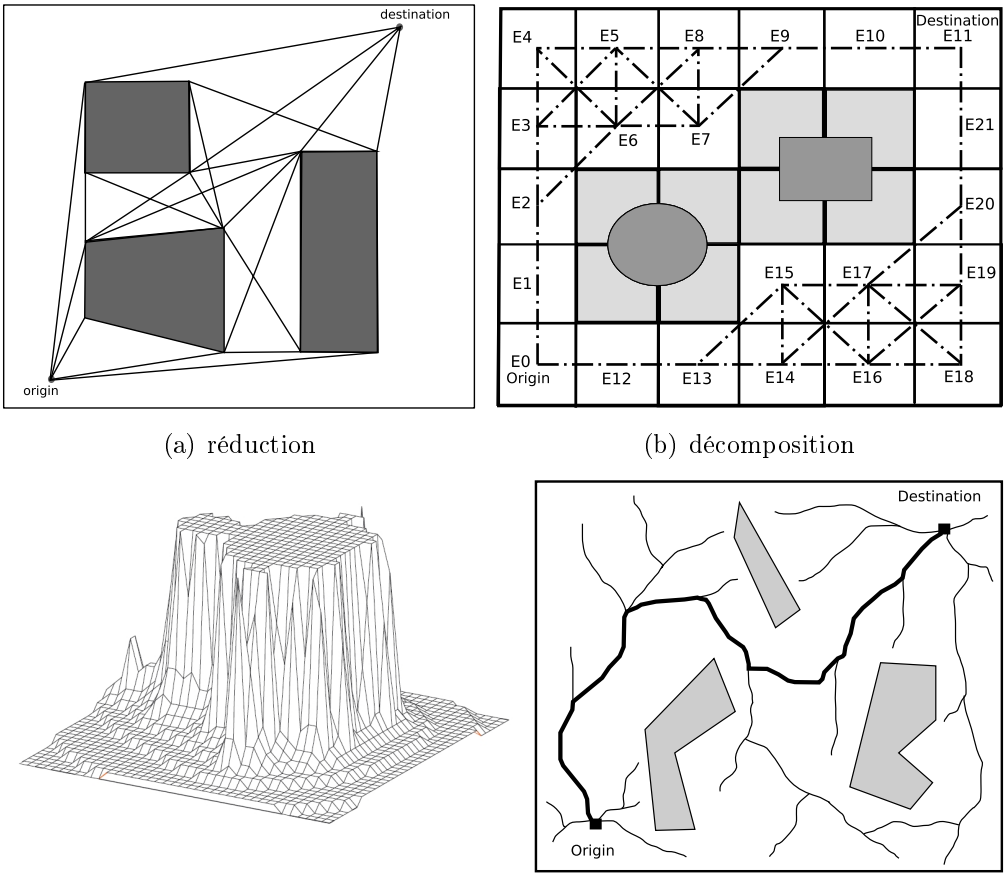
1.3.2 Différentes approches

Les algorithmes traditionnels de planification de déplacements utilisent généralement une structuration du modèle de l'environnement ou de l'espace de configuration sous la forme d'un graphe ou d'une grille. La plupart de ces méthodes peuvent être classées en quatre familles : les approches par réduction (fig. 1.13(a)), les approches par décomposition cellulaire (fig. 1.13(b)), les approches par champ de potentiels (fig. 1.13(c)) et les approches probabilistes (fig. 1.13(d)).

Les approches par réduction de l'environnement

L'objectif des approches par réduction est de capturer la connectivité de l'espace de configuration libre $\mathcal{C}free$ du robot sous la forme d'un graphe. Une fois construit, ce graphe est utilisé en tant qu'ensemble de chemins standardisés. La planification de chemins est alors réduite à la connexion des configurations initiale et finale au graphe, puis à la recherche d'un chemin solution. Le problème de planification devient alors un problème de recherche dans un graphe qui peut être résolu par différents algorithmes : A* [Hart *et al.*, 1968], Dijkstra [Dijkstra, 1959] (*cf.* annexe A), programmation par contraintes... Les deux méthodes les plus développées sont le graphe de visibilité et le diagramme de Voronoï.

Le graphe de visibilité. Cette méthode s'applique essentiellement aux espaces de configuration en deux dimensions avec des $\mathcal{C}$-obstacles polygonaux. Le graphe de visibilité est construit en connectant à l'aide d'un segment de droite chaque paire de sommets des $\mathcal{C}$-obstacles mutuellement visibles, *i.e.,* le segment ne traverse pas l'intérieur d'un obstacle. De plus, on exclut les segments dont le prolongement conduirait à l'intérieur d'un des deux obstacles. Cette méthode



(a) réduction (b) décomposition

(c) potentiel (d) probabiliste

FIGURE 1.13 – Quatre familles de méthodes pour la planification de chemin

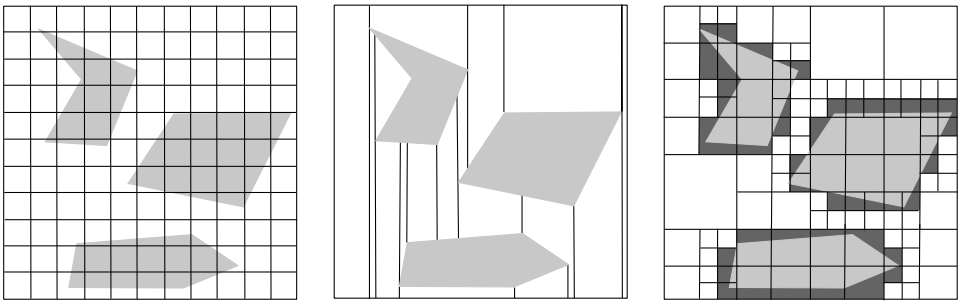
est mise en œuvre lorsque l'objectif du planificateur est d'optimiser la longueur totale du chemin.

Le diagramme de Voronoï. Si le robot doit rester éloigné des obstacles alors le diagramme de Voronoï est préféré. Ce graphe est défini à partir d'un ensemble de points qui sont équidistants de deux ou plusieurs obstacles. Le diagramme de Voronoï partitionne l'espace en régions qui contiennent au maximum un obstacle. Les frontières de ces régions, en deux dimensions, définissent les chemins que peut emprunter le robot. Le diagramme de Voronoï permet d'optimiser le critère de sécurité en maintenant le robot le plus éloigné possible des obstacles.

Les approches par décomposition de l'environnement

Ces méthodes permettent de décomposer l'espace de configuration en un ensemble de cellules et un ensemble de relations d'adjacence entre ces cellules. Chaque cellule correspond à une zone particulière de l'espace de configuration.

La méthode basique consiste à décomposer l'environnement en cellules régulières. Différentes décompositions sont possibles (figure 1.14) : décomposition en cellules convexes ou trapézoïdales, décomposition approximée ou décomposition hiérarchique (*e.g., quadtree*). La granularité de la décomposition influe fortement sur les performances de ce type d'approches en termes de temps de calcul. La recherche d'un chemin solution s'effectue en appliquant un algorithme de recherche entre la cellule initiale et la cellule destination sur le graphe de connectivité représentant les relations d'adjacence entre cellules. Les algorithmes de recherche dans un graphe évoqués pour les approches par réduction sont également utilisés pour les approches par décomposition de l'environnement.



(a) exacte (b) trapézoidale (c) hiérarchique

FIGURE 1.14 – Trois types de décomposition cellulaire

Les approches par champ de potentiels

Les approches par champ de potentiels sont des méthodes locales dans lesquelles le robot est assimilé à une particule sujette à l'influence d'un champ de potentiel artificiel qui reflète la topologie de l'environnement. La fonction de potentiel est définie sur l'espace libre comme la somme d'un potentiel attirant le robot vers la configuration finale et d'un potentiel répulsif permettant de tenir le robot à l'écart des obstacles. À chaque itération, la force induite par la fonction de potentiel permet de choisir la direction la plus prometteuse. Le principal reproche fait à ces méthodes est la possible existence de minima locaux piégeant le robot.

Les approches probabilistes

Les méthodes probabilistes sont des méthodes efficaces pour la résolution de problèmes impliquant un robot avec un grand nombre de degrés de liberté comme par exemple les robots articulés. Elles peuvent aussi être utilisées pour des robots mobiles. Cependant ce sont des méthodes incomplètes, *i.e.,* elles ne garantissent pas de trouver une solution en un temps fini même s'il existe

une solution. Leur complétude n'est que probabiliste : si une solution existe, la probabilité que l'algorithme la trouve tend vers un avec la durée d'exécution. Ces méthodes sont regroupées en deux sous-familles : les méthodes à requêtes multiples et les méthodes à requête unique.

Requêtes multiples. Parmi les méthodes à requêtes multiples, la plus connue et développée est la méthode *Probabilistic RoadMap* (PRM) [Kavraki and Latombe, 1994; Kavraki *et al.*, 1996]. Cet algorithme est performant dans le cadre d'environnements statiques. Il se déroule en deux phases : une phase de construction et une phase de requêtes. Durant la phase de construction, appelée également phase d'apprentissage, un graphe est construit à partir d'un ensemble de configurations exemptes de toute collision tirées aléatoirement. Les nœuds du graphe représentent ces configurations et les arêtes représentent les chemins permettant de rejoindre les configurations en évitant les obstacles. La phase de requêtes, ou phase de recherche, consiste à trouver un chemin entre une configuration initiale et une configuration finale en reliant les deux configurations à des nœuds du graphe et en construisant un chemin entre ces configurations à partir des segments précédemment calculés. On retrouve ici le même fonctionnement en deux phases que pour les approches par réduction et décomposition de l'environnement : première phase la construction d'un graphe, deuxième phase la recherche d'un chemin dans ce graphe.

Requête unique. L'algorithme à requête unique qui a fait l'objet du plus grand effort de recherche est l'algorithme *Rapidly-exploring Random Tree* (RRT) [LaValle, 1998; LaValle and Kuffner, 1999; Kuffner and LaValle, 2000]. RRT est un algorithme probabiliste incrémental qui explore rapidement l'environnement en cherchant à connecter itérativement des configurations tirées aléatoirement jusqu'à atteindre la configuration finale. Du fait de sa nature incrémentale, cette méthode permet de gérer les systèmes à forts degrés de liberté et permet de vérifier, à tout moment, des contraintes cinématiques complexes.

Approche mixte. Dans [Akinc *et al.*, 2003], la méthode *Probabilistic roadmap of Trees* (SRT) est présentée. L'algorithme fonctionne comme l'algorithme PRM mais les nœuds du graphe de recherche sont des arbres connectés entre-eux au lieu de connections entre simples configurations. La méthode combine les avantages des planificateurs à requête unique et ceux à requêtes multiples et a pour objectif de permettre la parallélisation du processus de recherche d'un chemin solution. Cette approche est améliorée dans [Plaku *et al.*, 2005] sous le nom de *Sampling-based Roadmap of Trees* (SRT) en proposant différentes méthodes de tirage des configurations s'appuyant sur des connaissances de l'environnement. En effet, les versions originales des méthodes PRM, RRT et PRT utilisent un tirage aléatoire uniforme des configurations.

Hybridisation de deux approches

Chaque méthode de représentation de l'environnement et de planification de chemins a ses avantages et ses inconvénients. Certains algorithmes font intervenir des techniques de différentes approches afin de combiner leurs avantages. C'est le cas de *Discrete Search Leading continuous eXploration* (DSLX) [Plaku *et al.*, 2007, 2008], un algorithme qui fonctionne sur le principe des RRTs et fait appel à une discrétisation de l'environnement afin de guider la construction de l'arbre dans les zones de l'environnement les plus susceptibles de contenir la solution finale.

1.3.3 Algorithme RRT

Dans cette partie, nous détaillons le fonctionnement de l'algorithme RRT [LaValle, 1998] qui sert de base à la mise en œuvre du planificateur Cell-RRT (Chapitre 5). Puis nous présentons les différentes améliorations que l'on peut rencontrer dans la littérature concernant, d'une part, le guidage de la construction de l'arbre et, d'autre part, l'application de l'algorithme aux environnement dynamiques.

Principe de RRT

L'objectif de la méthode RRT est de générer un arbre à partir de configurations tirées aléatoirement. Le cœur de la méthode est représenté par l'algorithme 1.1. À partir d'une configuration initiale q_{start}, l'objectif de l'algorithme est de trouver une séquence de configurations permettant au robot mobile d'atteindre la configuration finale q_{goal}. L'algorithme est constitué d'une boucle faisant appel à trois fonctions principales : chooseTarget() qui permet de sélectionner une configuration aléatoirement, nearestNeighbor() qui permet de connaitre la configuration présente dans l'arbre T la plus proche de celle sélectionnée et, extend() qui permet de créer une nouvelle configuration atteignable à partir de la configuration de l'arbre tout en respectant les contraintes cinématiques.

À chaque itération, une nouvelle configuration q_{target} est tirée aléatoirement. Puis, le nœud de l'arbre correspondant à la configuration la plus proche, suivant la métrique choisie, $q_{nearest}$ est sélectionné. Enfin, une nouvelle configuration q_{new} est créée en étendant la configuration sélectionnée en direction de la configuration tirée selon une distance préalablement fixée. L'algorithme se termine lorsque la nouvelle configuration peut être connectée au but, dans ce cas il existe un chemin solution, ou lorsque le nombre maximum K de tirages aléatoires a été atteint. Cette version de RRT est appelée RRT-Extend.

Une autre version, appelée RRT-Connect [LaValle and Kuffner, 1999; Kuffner and LaValle, 2000], consiste à essayer de connecter la configuration la plus

Algorithme 1.1 RRT-EXTEND(q_{start}, q_{goal})

```
1: T.init(q_start) ;
2: i ← 0 ;
3: q_new ← q_start ;
4: while (i < K and canConnect(q_new, q_goal) = null) do
5:    q_target ← chooseTarget() ;
6:    q_nearest ← T.nearestNeighbor(q_target) ;
7:    q_new ← extend(q_nearest, q_target) ;
8:    if (q_new ≠ null) then
9:       T.add(q_new) ;
10:   end if
11:   i ← i + 1 ;
12: end while
13: return T ;
```

proche à la configuration tirée. Son principe est représenté par l'algorithme 1.2. Dans ce cas, la configuration la plus proche est étendue vers la configuration aléatoire jusqu'à connection ou jusqu'à rencontrer un obstacle.

Algorithme 1.2 RRT-CONNECT(q_{start}, q_{goal})

```
1: T.init(q_start) ;
2: i ← 0 ;
3: q_new ← q_start ;
4: while (i < K and canConnect(q_new, q_goal) = null) do
5:    q_target ← chooseTarget() ;
6:    q_nearest ← T.nearestNeighbor(q_target) ;
7:    while (q_new ≠ null and canConnect(q_new, q_target) = null) do
8:       q_new ← extend(q_nearest, q_target) ;
9:       if (q_new ≠ null) then
10:         T.add(q_new) ;
11:         q_nearest ← q_new ;
12:      end if
13:   end while
14:   i ← i + 1 ;
15: end while
16: return T ;
```

Afin d'accélérer le processus de recherche d'un chemin solution, une amélioration des algorithmes suivants est l'utilisation de deux arbres au lieu d'un seul [LaValle and Kuffner, 1999; Kuffner and LaValle, 2000]. Le premier RRT est initialisé avec la configuration initiale q_{start} et le second avec la configuration finale q_{goal}. La recherche se termine lorsque les deux arbres se rejoignent. Les deux méthodes précédentes (RRT-Extend et RRT-Connect) peuvent être utili-

sées. Lorsque la méthode RRT-Extend est mise en œuvre sur les deux arbres, on parle de l'algorithme RRT-ExtExt. Lorsque la méthode RRT-Connect est mise en œuvre, l'algorithme est appelé RRT-ConCon. Les deux versions hybrides RRT-ExtCon et RRT-ConExt ont également été étudiées. En pratique, l'algorithme RRT-ExtCon est souvent préféré lorsque la recherche est effectuée en environnement moyennement contraint du fait de sa nature d'algorithme glouton.

Quelque soit la méthode utilisée, L'intérêt de RRT est de permettre une exploration rapide et non exhaustive de l'espace de configuration tout en favorisant les régions qui n'ont pas encore été explorées. En effet, avec l'utilisation d'un tirage uniforme, cette exploration peut être déterminée par le diagramme de Voronoï construit à partir des nœuds de l'arbre RRT : la probabilité qu'un nœud soit choisi comme voisin le plus proche est proportionnelle au volume de sa région de Voronoï.

1.3.4 Extensions de RRT

De nombreuses améliorations ont été proposées suite aux travaux de LaValle et Kuffner. L'objectif de la plupart de ces améliorations est de développer l'arbre de manière plus dirigée que lors de l'utilisation des algorithmes originaux. En effet, la recherche d'un chemin solution peut échouer si l'environnement contient des passages étroits comme par exemple des corridors. Un second axe de recherche développé est l'application des méthodes RRT à des environnements dynamiques ou partiellement connus.

Heuristiques de recherche et guidage de la construction de l'arbre

DDRRT [Yershova *et al.*, 2005], pour *Dynamic Domain* RRT, est une méthode qui a pour objectif de réduire le nombre d'itérations de l'algorithme en adaptant la métrique utilisée lors du choix du nœud à connecter de manière à favoriser les nœuds éloignés des obstacles. En effet, même si la région de Voronoï d'un nœud est importante, celui-ci peut éventuellement être proche d'un obstacle.

La méthode RRT-CT [Cheng and LaValle, 2001] permet de garder les informations sur les expansions passées qui ont échouées afin d'éviter à l'algorithme principal de chercher à étendre à nouveau ces nœuds. Elle utilise non seulement la métrique mais également l'information d'échec lors de la recherche du plus proche voisin.

[Strandberg, 2004] propose l'utilisation d'arbres locaux afin d'améliorer la qualité de la solution pour les zones de l'environnement peu accessibles. Dans cette approche, dès que l'algorithme principal rencontre une configuration aléatoire qui ne permet pas l'expansion de l'arbre, alors un arbre local est créé.

Lorsqu'il y a intersection entre les branches de deux arbres locaux, ceux-ci sont fusionnés. Cet algorithme revient à développer l'arbre RRT en priorité dans les zones critiques.

OBRRT [Rodriguez *et al.*, 2006] est une variante de RRT-Extend qui tient compte des informations de l'environnement pour choisir la direction dans laquelle étendre l'arbre. À proximité des obstacles, l'algorithme va fournir des configurations aléatoires permettant de contourner ces obstacles à partir de règles de comportement prédéfinies.

Une autre approche est la méthode RRT-Blossom [Kalisiak and von de Panne, 2006]. Avant d'étendre une configuration, l'algorithme étudie les différentes nouvelles configurations possibles et choisit celle qui n'entraine pas de régression, *i.e.,* celle qui favorise l'exploration de l'environnement. Le principe de cette méthode est assez similaire à l'évitement des minima locaux dans les approches de planification de chemins par champ de potentiels.

De la même façon, l'algorithme Util-RRT [Burns and Brock, 2007] permet de choisir une configuration à étendre en fonction d'un critère d'utilité. Ce critère est fonction d'un estimateur de l'utilité et de la probabilité de réussite de l'expansion. Cette méthode permet de privilégier les zones de l'environnement qui n'ont pas encore été explorées.

Enfin [Zhang and Manocha, 2008] propose un algorithme qui se concentre uniquement sur le problème des passages de corridors étroits. Ces travaux s'appuient sur la retraction : lorsque l'arbre ne peut pas être étendu vers la configuration tirée alors une phase de rétraction est appliquée à partir du nœud le plus proche. Cette phase, formulée comme un problème d'optimisation utilisant une métrique appropriée, permet de générer un ensemble de configurations possibles proches de la configuration impossible. Puis l'algorithme tente de connecter chacune de ces nouvelles configurations à l'arbre.

Application aux environnements dynamiques

L'algorithme ERRT [Bruce and Veloso, 2002], pour *Execution extended* RRT, propose un système de planification de chemins qui entrelace planification et exécution. L'objectif est d'améliorer l'efficacité de la replanification ainsi que la qualité des chemins générés. Pour cela, les auteurs proposent deux améliorations : un système de mise en cache des points de passage déjà planifiés et un système adaptatif de pénalisation du coût de recherche. Le système de mise en cache permet, lors de la replanification, de conserver les segments de chemin qui n'ont pas été concernés par la modification de l'environnement. La pénalité adaptative lors du calcul de la distance entre deux configurations permet d'envisager des alternatives afin d'explorer d'autres zones de l'environnement. Ainsi, la configuration la plus proche, au sens de la métrique utilisée, ne sera pas forcément choisie lors de l'extension de l'arbre.

Ferguson *et al.* [2006] proposent une alternative à ce système de replanification, appelée Dynamic RRT. Le principe est, lors de la replanification, de modifier les branches de l'arbre RRT uniquement à partir des nœuds invalidés. Lors de la détection d'un nouvel obstacle dans l'environnement, l'algorithme cherche les configurations qui sont devenues impossibles et crée de nouveaux arbres ayant pour racine la racine de ces configurations. L'avantage de cette méthode par rapport à la méthode ERRT est le faible nombre de nœuds ajoutés à l'arbre suite à la modification de l'environnement.

1.3.5 Bilan

Cette analyse des méthodes de planification de déplacements met en évidence deux grandes catégorie : Les méthodes qui construisent, préalablement à toutes requêtes, un graphe de l'environnement ; et celles qui effectuent des calculs uniquement en fonction des requêtes reçues. Il est intéressant de noter que, même si l'interaction entre un planificateur de déplacements et un planificateur de tâches peut impliquer un grand nombres de requêtes, il est peu vraisemblable que ce nombre dépasse le nombre d'arcs du graphe.

1.4 Planification hybride

Les planificateurs indépendants du domaine développés actuellement sont de plus en plus performants et permettent de résoudre des problèmes de plus en plus complexes [Hoffmann *et al.*, 2006]. Cependant, comme ils ne sont pas destinés à résoudre un type de problème particulier, leurs concepteurs tendent à les rendre le plus générique possible afin de pouvoir comparer leurs performances avec les planificateurs qui font autorité dans la communauté planification. Par conséquent, les problèmes utilisés sont souvent idéalisés ou sont des versions simplifiées de problèmes réels [Howe and Dahlman, 2002]. Par ailleurs, la représentation interne utilisée par les planificateurs indépendants du domaine ne leur permet pas de gérer les informations géométriques. Le couplage d'un planificateur de tâches avec un module de raisonnement géométrique spécialisé, comme par exemple un planificateur de mouvements, est souvent la seule manière réaliste de résoudre les problèmes de robotique.

1.4.1 Coopération entre planificateurs

Tâches puis déplacements

Comme nous l'avons vu précédemment, dans les architectures robotiques actuelles, le planificateur chargé du déroulement de la mission, c'est-à-dire du

choix des actions à réaliser, et le planificateur chargé de la gestion des mouvements du robot permettant de réaliser ces actions, sont très souvent découplés. Tout d'abord un plan abstrait permettant de réaliser la mission est défini. Puis, ce plan est raffiné, c'est-à-dire que les déplacements du robot sont calculés en tenant compte des informations géométriques de l'environnement et du modèle du robot. Certaines tâches à réaliser telles les tâches de manipulation [Alami *et al.*, 1995] impliquent de fortes contraintes sur la configuration initiale du mouvement. Ainsi de nombreux travaux visant à réduire le fossé entre planification de tâches et planification de mouvements s'intéressent essentiellement à la définition de cette configuration [Lozano-Perez *et al.*, 1989; Zacharias *et al.*, 2006], ou encore à limiter les différences de représentation entre le niveau symbolique et le niveau géométrique [Choi and Amir, 2009].

Cependant, le problème formé par le choix de la tâche et sa réalisation du point de vue géométrique peut nécessiter un raisonnement plus complexe. En effet, si une action ne peut être réalisée dans l'environnement alors une autre action devra être sélectionnée. Par ailleurs, le choix de cette action peut dépendre de la configuration du robot et de l'environnement à un instant donné.

Déplacements puis tâches

D'autres types d'architecture pour la résolution de problèmes spécifiques de planification de mission ont été proposés [Chanthery, 2005; Baltié *et al.*, 2008]. Ces architectures incluent des pré-calculs géométriques qui produisent un graphe. Les nœuds sont calculés à partir des objectifs courants et des contraintes géométriques sur les actions. Les arcs sont calculés en utilisant une méthode de planification de déplacements. Le principal inconvénient de cette approche est que les calculs géométriques sont effectués pour toutes les actions possibles, même celles qui ne seront pas sélectionnées dans la solution finale.

Tâches et déplacements entrelacées

Le terme *planification hybride* a été proposé pour la première fois par [Kambhampati *et al.*, 1993]. Il exprime l'utilisation de plusieurs raisonneurs afin de résoudre un unique problème. Dans le cadre de cette thèse, il s'agit du couplage entre raisonnement symbolique et raisonnement géométrique. Les auteurs proposent un modèle de planification hybride qui utilise un ensemble de modules spécialisés pour augmenter le pouvoir d'expression et les capacités de raisonnement du planificateur de tâches. Dans ces travaux, les différents raisonneurs, appelés spécialistes, et le planificateur de tâches utilisent un modèle central qui contient les données symboliques ainsi que les données géométriques du problème à résoudre. Le système résout le problème incrémentalement : le planificateur de tâches produit un premier plan d'actions. Puis, ce plan est vérifié par les spécialistes qui vont ajouter un ensemble de contraintes entre les

actions en cas d'échec. À partir de ces contraintes, le planificateur de tâches révise le plan puis le resoumet à l'ensemble des spécialistes.

Cette idée d'utiliser des raisonneurs spécialisés a été reprise par [Lamare and Ghallab, 1998] pour les problèmes faisant intervenir un robot mobile qui doit se déplacer dans l'environnement. Dans ces travaux, le planificateur de tâches IxTeT [Laborie and Ghallab, 1995] est couplé avec un planificateur d'itinéraires qui s'appuie sur un graphe d'accessibilité préalablement calculé et met en œuvre l'algorithme de Moore-Dijkstra. Lors du processus de planification, lorsqu'un sous-problème de déplacement est identifié, celui-ci est envoyé au planificateur d'itinéraires. Ces sous-problèmes sont identifiés par le fait qu'il n'y ait plus que des défauts sur un attribut particulier relatif à la position du robot. Cet attribut est défini lors de la modélisation du domaine de planification.

Lors de la troisième compétition internationale de planification [Third International Planning Competition, 2002], le planificateur hiérarchique SHOP2 faisait appel à des routines externes codées en Lisp pour calculer les meilleurs chemins sur les problèmes du domaine des rovers. Cette manière d'améliorer les performances du planificateur peut être vue comme l'utilisation d'un modèle hybride dans lequel certains sous-problèmes sont résolus par des algorithmes spécifiques.

1.4.2 Planification pour la manipulation

Une des premières tentatives de développer un planificateur permettant de traiter simultanément les aspects symboliques et géométriques des problèmes de manipulation a été conduite dans le cadre de l'exploitation de la technique de résolution du planificateur ShaPer [Guéré and Alami, 2001]. L'algorithmique de ShaPer vise à accélérer le processus de planification en apprenant la topologie d'un graphe d'accessibilité entre états. Ce planificateur utilise le langage de planification STRIPS augmenté par des faits numériques, pour représenter les aspects géométriques du problème, et des contraintes d'inclusion et d'inégalité entre des fonctions sur les arguments de ces faits.

Par la suite, Gravot et Cambon ont développé le planificateur aSyMov [Gravot, 2004; Cambon, 2005]. L'objectif de ce planificateur est de permettre la production de plans pour les problèmes de manipulation dans lesquels l'exécution d'une action a une forte répercussion sur la représentation géométrique du problème. Par exemple, lorsqu'un robot doit transporter un objet, la forme de l'ensemble robot plus objet est différente de celle du robot à vide. Ceci peut avoir des conséquences sur l'exécution des actions suivantes [Gravot *et al.*, 2005]. Le planificateur aSyMov fait intervenir un ensemble de types de prédicats ainsi qu'un ensemble de prédicats particuliers pour représenter ces interactions entre représentation symbolique et contrepartie géométrique [Cambon *et al.*, 2009]. Du point de vue algorithmique, un plan relaxé est initialement

calculé, c'est-à-dire que tous les mouvements sont considérés comme étant valides, puis ce plan est raffiné progressivement en tenant compte des contraintes géométriques.

1.4.3 Bilan

Les travaux sur la planification pour la manipulation montrent l'intérêt d'utiliser des faits et fonctions numériques dans la planification de tâches pour considérer les aspects géométriques. Par ailleurs, il semble que, malgré les progrès des planificateurs de tâches, il est toujours intéressant de traiter les problèmes de déplacements à part. La question de savoir s'il est préférable de traiter les problèmes de déplacements préalablement, en cours ou après la planification de tâches reste une question ouverte.

1.5 Conclusion

Dans ce chapitre nous avons présenté les différentes possibilités d'architecture logicielle pour un robot mobile. Ces architectures sont classifiées en deux grandes familles : les architectures réactives et les architectures délibératives. Seules les architectures délibératives permettent la gestion de missions complexes nécessitant un raisonnement étendu sur les actions que le robot devra entreprendre pour réaliser la mission. Une architecture délibérative est généralement décomposée en trois niveaux hiérarchiques : un niveau délibératif, un niveau de contrôle de l'exécution et un niveau fonctionnel. Souvent, le raisonnement global sur la mission est effectué au niveau délibératif tandis que les raisonnements plus locaux tels que la recherche d'un chemin ont lieu au niveau fonctionnel. Néanmoins, pour certaines architectures et des contextes où la traversabilité de l'environnement est relativement bien connue à l'avance, la cohabitation entre planification de tâches et planification de mouvements dans la couche décisionnelle est possible.

Puis nous avons présenté les différents types de planification pouvant intervenir lors de l'utilisation de ces architectures robotiques, en mettant l'accent sur la planification de tâches et la planification de mouvements.

La planification de tâches correspond au raisonnement symbolique global sur la mission. Nous avons présenté les différents formalismes ainsi que les différents algorithmes qui peuvent être mis en œuvre dans le cadre de la planification de mission pour la robotique mobile. Il semble que les algorithmes de planification par chaînage avant présentent plus d'opportunités d'interaction avec la planification de mouvements que les autres algorithmes.

La planification de mouvements correspond à un raisonnement géométrique dans l'environnement et permet au robot de se déplacer. Il existe différentes

familles d'algorithmes de planification de mouvements : les approches par décomposition, les approches par réduction, les approches par champs de potentiel ainsi que les approches probabilistes. Nous avons détaillé l'algorithme RRT ainsi que les différentes versions existantes. Cet algorithme planifie dans l'espace des configurations et semble en conséquence plus adapté pour l'intégration de contraintes géométriques fines que les algorithmes travaillant directement sur l'environnement.

Le raisonnement global couplant la planification de tâches et la planification de déplacements relève de la planification hybride. Les connaissances dans ce domaine de recherche demandent à être approfondies sur deux points principaux :

- L'interaction entre planificateur de tâches et planificateur de mouvements ;
- L'expression et la satisfaction des contraintes géométriques portant sur la planification de mouvements en adéquation avec la description des actions de la planification de tâches.

Chapitre 2

Vers une architecture de planification hybride

Afin d'accomplir sa mission, un robot mobile doit agir et se déplacer dans l'environnement. La planification de mission pour la robotique mobile fait intervenir différents types de raisonnement : un raisonnement symbolique permettant de sélectionner les actions que le robot devra entreprendre pour réaliser la mission et, un raisonnement géométrique dont le but est de calculer les mouvements du robot, ou plus généralement, de définir ses déplacements dans l'environnement.

L'objectif de ce chapitre est d'apporter une réponse expérimentale à trois questions relatives à la planification pour les robots mobiles :

- L'utilisation d'un planificateur spécialisé dédié à la gestion des problèmes de recherche de chemins améliore-t-elle le temps de calcul et la qualité du plan solution ?
- Dans l'affirmative, est-il préférable de construire un plan symbolique puis de calculer les chemins, ou bien de calculer les chemins au fur et à mesure de la construction du plan symbolique ?
- Toujours en supposant une réponse positive à la première question, le planificateur de chemin doit-il aider le planificateur de tâches en lui fournissant certaines informations sur l'environnement ?

Dans un premier temps, nous présentons la plateforme expérimentale mise en œuvre pour réaliser les expérimentations ainsi que la modélisation du domaine et des problèmes utilisés. Puis nous présentons les résultats de la première expérimentation qui consiste en une comparaison des temps de calcul et des qualités de solution lorsqu'un planificateur symbolique raisonne seul et lorsqu'un modèle de planification hybride est utilisé. Ensuite, nous proposons et étudions différentes méthodes permettant de coupler les deux planificateurs. Finalement, nous cherchons à savoir comment l'utilisation d'heuristiques, pour

la méthode hybride, permet d'améliorer la qualité de la solution lorsque l'ordre de réalisation des objectifs n'est pas fixé *a priori*.

2.1 Plateforme expérimentale

Afin de mener à bien les expérimentations, nous avons développé deux planificateurs : un planificateur de tâches et un planificateur de déplacements. Les expérimentations sont menées sur des problèmes inspirés du *domaine des rovers*.

2.1.1 Présentation du planificateur symbolique

Le planificateur symbolique mis en œuvre est un planificateur HTN offrant des fonctionnalités similaires à celles offertes par SHOP2 [Nau *et al.*, 2003]. Ces fonctionnalités sont décrites en annexe B.

Structures de données

Le langage de planification hiérarchique que nous mettons en œuvre est inspiré du langage PDDL. Il contient des opérateurs et des méthodes. La brique de base du langage est le terme. Un terme peut être une variable, une constante, ou encore une fonction. À partir de ces termes, il est alors possible de définir des formules atomiques appelées atomes. Ces formules atomiques sont décomposées en deux sous groupes : les littéraux permettant d'exprimer un prédicat p ; et les entêtes de tâches (*task atom*) qui permettent d'identifier les opérateurs et les méthodes, mais également les tâches à planifier. Les littéraux sont regroupés sous la forme de conjonctions et les entêtes sous la forme de listes de tâches.

Ainsi, à partir de ces structures de base, il est possible d'exprimer le problème et le domaine de planification. Par exemple, l'état initial est une conjonction de littéraux et le but à résoudre est représenté par une liste de tâches. Un opérateur est identifié par son entête, et ses préconditions et effets sont exprimés sous la forme de conjonctions de littéraux.

Principe de résolution

Le problème de planification peut se représenter comme la racine d'un arbre que nous appelerons *arbre de planification*. Un nœud de cet arbre est composé de l'état courant du monde et de la liste des tâches à exécuter. Ainsi, à l'initialisation, l'arbre est composé d'un unique nœud contenant l'état initial et la liste des tâches à planifier. L'objectif de l'algorithme de planification est alors de développer l'arbre jusqu'à obtenir un nœud feuille, *i.e.,* un nœud dont

la liste des tâches est vide. Le développement d'un nœud vers un ensemble de nœuds fils s'effectue en appliquant un opérateur ou une méthode.

Algorithmique du planificateur de tâches

Chercher à déterminer si et comment un opérateur ou une méthode est applicable revient à chercher l'ensemble des substitutions unifiant les variables de ses préconditions avec l'état courant.

Soit *preconds* les préconditions d'un opérateur ou d'une méthode, s l'état courant et σ un ensemble de substitutions, l'algorithme 2.1 cherche à déterminer récursivement les substitutions possibles afin d'unifier la liste de préconditititions *preconds* avec l'état courant s. L'algorithme présenté est une version simplifiée de l'algorithme mis en œuvre. La version implémentée permet de prendre en compte les prédicats imbriqués, les prédicats négatifs, les listes, les quantificateurs universels... Ces différents éléments du langage sont présentés en annexe B.

L'algorithme, dans un premier temps, teste si l'ensemble de préconditions est vide. Si c'est le cas, alors il s'arrête (ligne 3). Dans le cas contraire, il dépile la première précondition et essaie de l'unifier avec chaque prédicat de l'état s (ligne 8). Si l'unification est possible, c'est-à-dire qu'il existe une substitution unifiant p avec un prédicat p' de l'état s, alors l'algorithme cherche récursivement les substitutions possibles de l'ensemble R des préconditions non testées en prenant en compte les variables précédemment instanciées (ligne 10).

Algorithme 2.1 FindSubstitutions(*preconds*, s, σ)

```
 1: result ← ensemble vide de substitutions ;
 2: if (preconds = ∅) then
 3:    return σ ;
 4: end if
 5: p ← le premier prédicat de preconds ;
 6: R ← le reste des prédicats de preconds ;
 7: for each (predicat p' ∈ s) do
 8:    θ ← Unify(p, p', σ) ;
 9:    if (θ ≠ null) then
10:       Σ ← FindSubstitutions(R, s, σ + θ) ;
11:       for each (substitution γ ∈ Σ) do
12:          ajouter γ à result ;
13:       end for
14:    end if
15: end for
16: return result ;
```

L'algorithme 2.2 est l'algorithme central du planificateur. Il présente le développement de l'arbre de planification, *i.e.,* la décomposition récursive des tâches complexes en sous-tâches jusqu'à l'obtention d'une séquence de tâches primitives. Le développement de l'arbre se fait en profondeur d'abord et s'arrête lorsque la liste des tâches est vide.

Un nœud de l'arbre est défini par le tuple $(s, < t_0, ..., t_n >)$ où s est l'état courant du monde et $< t_0, ..., t_n >$ est la liste des tâches restantes à réaliser à cette étape de la décomposition. Les arêtes entre les nœuds de l'arbre représentent les transitions possibles entre les différentes étapes de résolution. Ces transitions résultent de l'application d'un opérateur ou d'une méthode sur l'état courant. Lorsque la transition résulte de l'application d'un opérateur, elle correspond à un changement de l'état du monde.

L'algorithme est initialisé avec le nœud représentant l'état initial ainsi que la liste des tâches donnée en entrée du planificateur. Dans un premier temps, la procédure teste si la liste des tâches est vide. Dans ce cas, une feuille a été atteinte, *i.e.,* une solution au problème a été trouvée et le nœud courant est renvoyé en tant que nœud solution. Dans le cas contraire, l'algorithme tente de réaliser la première tâche t_0.

Si la tâche t_0 est une tâche primitive alors, pour chaque opérateur o contenu dans $\mathcal{O}$, l'algorithme teste s'il peut réaliser la tâche t_0, *i.e.,* l'unification entre o et t_0 est possible (ligne 6). Si c'est le cas, la procédure calcule les substitutions résultant de l'unification des préconditions de l'opérateur avec l'état courant s (ligne 8). Puis, pour chaque substitution σ, la procédure crée un nouveau nœud contenant un nouvel état courant résultant de l'application des effets de l'opérateur o à l'état courant s, et de la liste des tâches privée de la tâche t_0, puis appelle récursivement l'algorithme sur ce nœud (ligne 12).

Si la tâche t_0 est une tâche complexe alors la procédure cherche à appliquer une méthode m contenue dans l'ensemble des méthodes $\mathcal{M}$ du domaine (ligne 22). Puis l'algorithme teste, pour chaque décomposition d de la méthode, les préconditions de la même façon que pour une tâche primitive (ligne 26). Pour chaque substitution σ trouvée, il crée un nouveau nœud en remplaçant la tâche t_0 de la liste des tâches par la réduction de la décomposition (ligne 30), *i.e.,* la liste des tâches spécifiée dans la décomposition d de la méthode.

2.1.2 Transmission de données géométriques

Afin de réaliser les expérimentations dans lesquelles le planificateur de tâches délègue les problèmes de recherche d'un chemin à un planificateur spécialisé, nous avons modifié la définition d'un opérateur de planification en y ajoutant un champ spécifique : les données géométriques. Un opérateur est décrit de la manière suivante :

Algorithme 2.2 Develop(n, $\mathcal{O}$, $\mathcal{M}$)

```
1: Soit n ← (s, < t_0, ..., t_n >) ;
2: if (< t_0, ..., t_n > = Nil) then
3:    return n ;
4: else if (t_0 est primitive) then
5:    for each (opérateur o ∈ O) do
6:       θ ← Unify(name(t_0), name(o), ∅) ;
7:       if (θ ≠ null) then
8:          Σ ← FindSubstitutions(preconditions(o), s, θ, ∅) ;
9:          if (Σ ≠ null) then
10:            for each (substitution σ ∈ Σ) do
11:               s' ← appliquer les effets de o à s ;
12:               n' ← Develop((s', < t_1, ..., t_n >), O, M) ;
13:               if (n' ≠ null) then
14:                  ajouter n' en tant que fils à n ;
15:                  return n ;
16:               end if
17:            end for
18:         end if
19:      end if
20:   end for
21: else if (t_0 est complexe) then
22:   for each (méthode m ∈ M) do
23:      θ ← Unify(name(t_0), name(m), ∅) ;
24:      if (θ ≠ null) then
25:         for each (décomposition d ∈ decompositions(m)) do
26:            Σ ← FindSubstitutions(preconditions(o), s, θ) ;
27:            if (Σ ≠ null) then
28:               for each (substitution σ ∈ Σ) do
29:                  < t_{n+1}, ..., t_{n+i} > ← Reduction(d)
30:                  n' ← (s, < t_{n+1}, ..., t_{n+i} > + < t_1, ..., t_n >) ;
31:                  n' ← Develop(n', O, M) ;
32:                  if (n' ≠ null) then
33:                     ajouter n' en tant que fils à n ;
34:                     return n ;
35:                  end if
36:               end for
37:            end if
38:         end for
39:      end if
40:   end for
41: end if
42: return null ;
```

```
(Operator (!navigate ?r ?x ?y)
 ;; preconditions
  (...)
 ;; données géométriques
  (...)
 ;; effects
  (...)
)
```

Lorsque le planificateur de tâches cherche à appliquer un opérateur comportant des données géométriques, il transmet le contenu de ce champ au planificateur spécialisé et attend la réponse de la recherche de chemin. Le résultat retourné est un booléen ayant la valeur vrai si un chemin permettant le déplacement du robot jusqu'à l'objectif est possible et la valeur faux le cas échéant. Ainsi, le résultat de la recherche d'un chemin peut être vu comme le test d'une précondition supplémentaire.

Le contenu de cette requête est un ensemble de prédicats spécifiant les coordonnées des points origine et destination du chemin à calculer.

Les demandes d'heuristiques sont représentées par le prédicat `(heuristic ...)`. Lorsque le planificateur de tâches rencontre un tel prédicat, alors il envoie une requête au planificateur de déplacements qui effectue le calcul correspondant et renvoie le résultat.

2.1.3 Présentation du raisonneur spécialisé

Le raisonneur spécialisé qui est utilisé est un planificateur de déplacement qui s'appuie sur un graphe de visibilité. Dans un souci de réalisme, les obstacles ont été augmentés à l'aide d'une somme de Minkowski [Ramkumar, 1996], ainsi le robot doit respecter une distance de sécurité à proximité des obstacles.

La somme de Minkowski de deux ensembles $E1, E2 \in \mathbb{R}^d$, notée $E1 \oplus E2$ est définie de la façon suivante :

$$E1 \oplus E2 = \{x + y \mid x \in E1, y \in E2\}.$$

Et plus particulièrement, pour un polygone P, l'ensemble des points dont la distance à P est inférieure à ρ est la somme de Minkowski $P \oplus R_\rho$, où R_ρ est un disque de rayon ρ (figure 2.1).

Dans un deuxième temps, le graphe de visibilité est construit (figure 2.2). Ce graphe contient l'ensemble des bitangentes possibles (des segments dont les prolongements ne sont pas en intersection avec les obstacles) des sommets des obstacles visibles deux à deux. Il existe quatre bitangentes entre deux sommets qui sont représentés par des cercles du fait de la somme de Minkowski. Mais toutes ne sont pas des bitangentes possibles (figure 2.3).

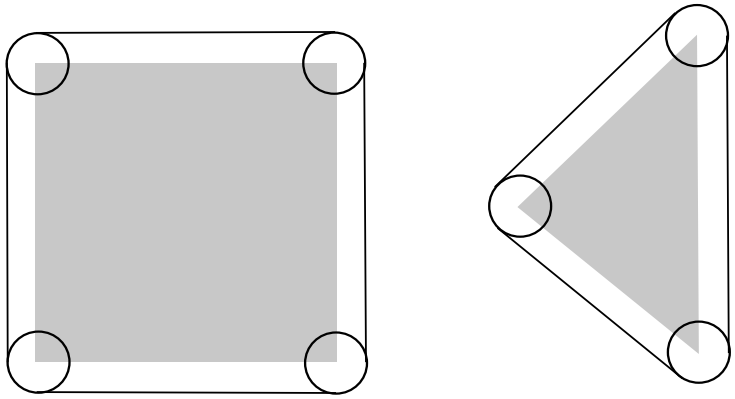


FIGURE 2.1 – Somme de Minkowski entre les obstacles et un disque de rayon ρ

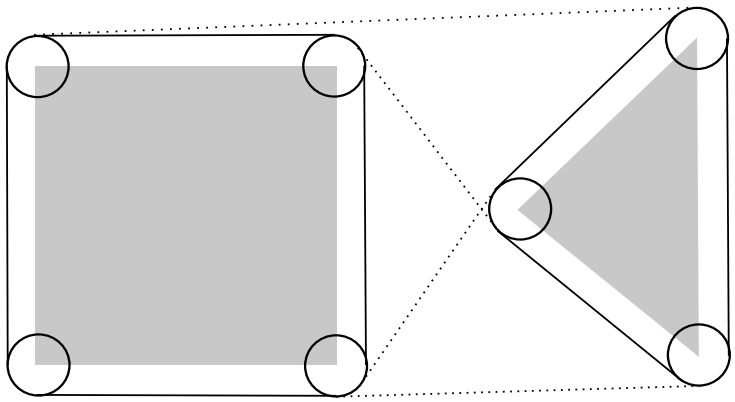


FIGURE 2.2 – Construction du graphe de visibilité

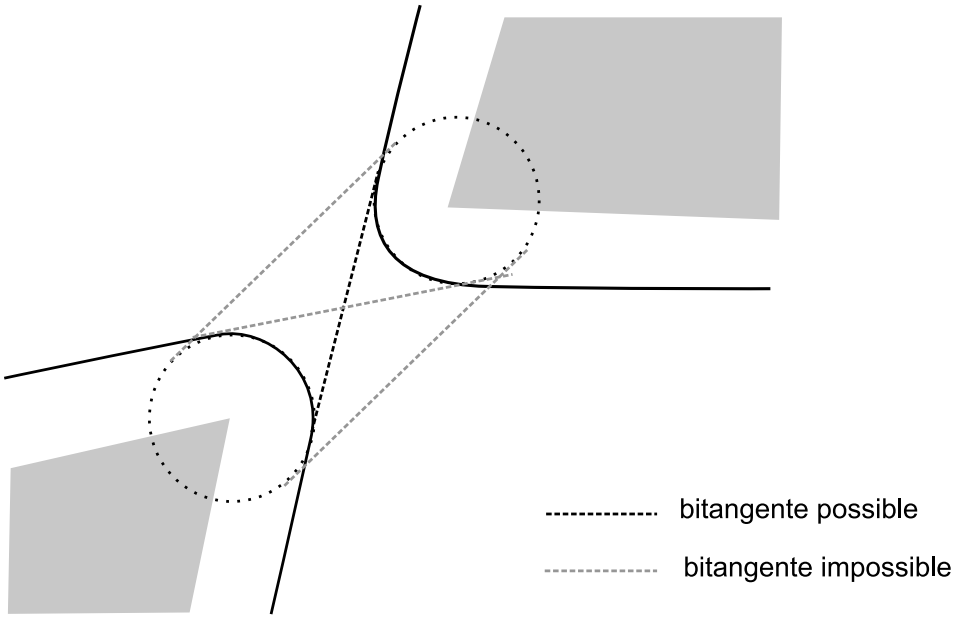


FIGURE 2.3 – Construction des bitangentes entre deux cercles

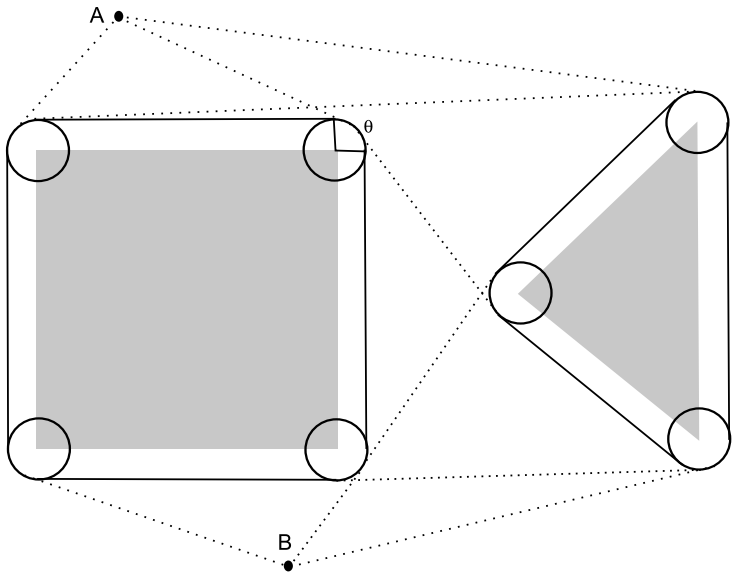


FIGURE 2.4 – Connexion des points origine et destination

Une fois le graphe de visibilité construit, le point origine A et le point destination B peuvent être connectés. La connexion de ces points se fait en calculant les tangentes aux différents sommets-cercles à partir des points (figure 2.4).

Le graphe obtenu contient des arêtes rectilignes qui correspondent aux bitangentes, et des arêtes curvilignes qui correspondent aux arcs de cercle entre les points de connexion des bitangentes sur les sommets augmentés. Les nœuds du graphe sont définis par ces points de connexion.

Enfin nous pouvons rechercher un chemin solution entre les deux points en utilisant un algorithme de recherche de plus court chemin. Le calcul des distances parcourues s'effectue de façon similaire au calcul dans le graphe de visibilité auquel on ajoute les longueurs parcourues sur les arcs de cercles au niveau des sommets des obstacles. Soit $d(.)$ la distance euclidienne correspondante au parcours sur une arête du graphe et $\rho\Theta$ la distance parcourue sur un arc de cercle au niveau d'un sommet en fonction de l'angle Θ (exprimé en gradient). La distance parcourue entre deux points est :

$$d_{TOT} = \sum_{i=1}^{M} d(m_i) + \sum_{j=1}^{N} \rho\Theta_j$$

où M est le nombre d'arêtes traversées, N est le nombre de sommets parcourus, m_i est la i^{eme} arête et Θ_j est l'angle parcouru sur le j^{eme} sommet.

L'algorithme de recherche d'un chemin mis en œuvre est l'algorithme de Dijsktra présenté en annexe A. La fonction de coût tient compte du type de l'arête parcouru : rectiligne ou curviligne.

2.2 Scénario d'évaluation

Dans le domaine des rovers, utilisé lors de la troisième compétition de planification [Third International Planning Competition, 2002] et inspiré par les problèmes de rovers destinés à l'exploration de planètes, un ensemble de robots doivent collecter des échantillons de roches, prendre des photos et transmettre leurs résultats à une station fixe. Les robots doivent en outre se déplacer dans l'environnement pour remplir leur mission.

Le problème que nous utilisons pour comparer les différentes approches de résolution du problème de planification est le suivant : un rover unique doit collecter des échantillons à des points de l'environnement préalablement définis. Le rover doit se déplacer dans l'environnment tout en évitant les obstacles. L'environnement, représenté sur la figure 2.5, est composé d'obstacles (en gris) et de zones sur lesquelles les échantillons doivent être prélevés (en jaune). La zone verte représente la position initiale du rover et la zone rouge correspond à la position qu'il doit rejoindre en fin de mission.

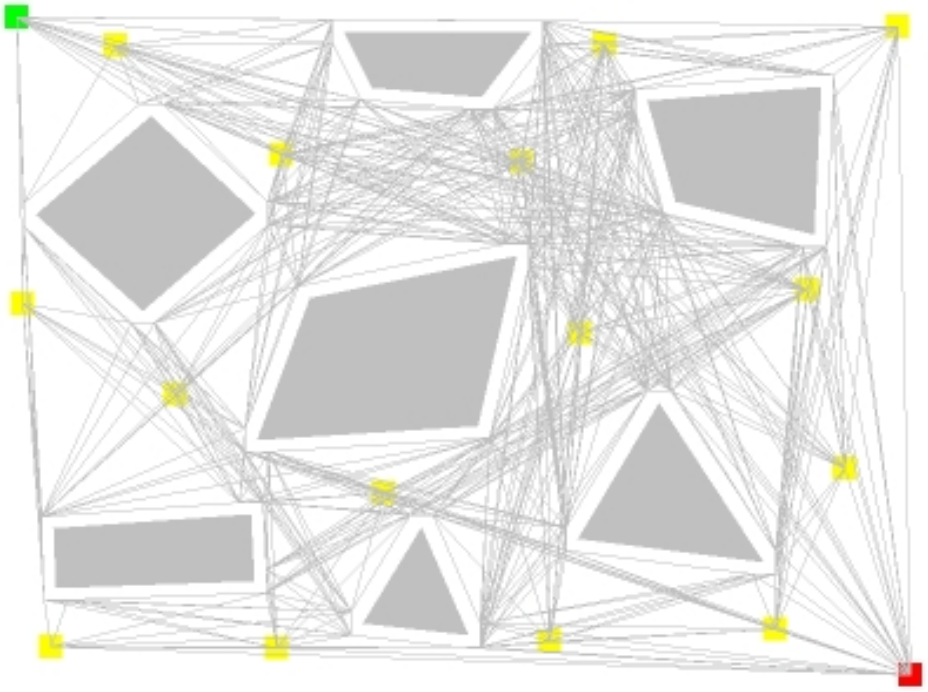


FIGURE 2.5 – La carte de l'environnement et le graphe de visibilité associé

Le domaine de planification et le problème sont formalisés avec le langage de planification décrit sous forme BNF en annexe B. L'état initial est composé de prédicats symboliques modélisant le rover et les ressources disponibles, ainsi que l'environnement sous la forme de prédicats représentant les points de passage et les chemins possibles entre ces points de passage.

Ces données géométriques sont extraites du graphe de visibilité associé à la carte de l'environnement (figure 2.5). Les sommets du graphe de visibilité sont transformés en points de passage symboliques et l'ensemble des arcs représentent les chemins possibles entre les points de passage. Ces chemins sont exprimés à l'aide du prédicat symbolique `can_traverse` tel que défini dans la version originale du problème :

```
(waypoint waypoint0)(waypoint waypoint1)
(can_traverse rover1 waypoint0 waypoint1)
```

Afin de permettre le choix du prochain objectif à réaliser selon différents critères, nous encodons dans le domaine un ensemble de méthodes permettant d'exprimer le but comme un ensemble de prédicats représentant l'état solution.

Ainsi, le problème à résoudre est spécifié par l'appel à une méthode ayant comme argument une liste de prédicats. Par exemple :

```
(achieve-goals (list
 (has_soil_data waypoint45)
 (has_soil_data waypoint29)
 (has_soil_data waypoint35)
 (has_soil_data waypoint14)
 (has_soil_data waypoint15)
))
```

Cette méthode transforme chaque prédicat en un prédicat "but" qui est inséré dans l'état initial sous la forme `(goal(has_soil_data waypoint..))`, puis la méthode `do-mission`, qui correspond à la méthode principale guidant la production du plan, est appelée.

```
(Method (achieve-goals ?goals)
 ()
 ((assert-goals ?goals nil)(do-mission))
)

(Method (assert-goals (list ?goal | ?goals) ?out)
 ()
 ((assert-goals ?goals (list (goal ?goal) | ?out)))
)

(Method (assert-goals nil ?but)
 ()
 ((!!assert ?but))
)
```

2.3 Différentes approches

Nous considérons différentes approches pour la planification de la navigation du robot, c'est-à-dire pour la recherche d'un chemin permettant au robot de réaliser les objectifs.

2.3.1 Méthodes de recherche classiques

Nous appelons méthode classique, la méthode consistant à rechercher un chemin en utilisant uniquement le planificateur de tâches. La recherche s'effectue en appliquant successivement des actions de navigation jusqu'à arriver dans un état qui contient le prédicat `(at rover0 ?y)` où la variable `?y` a comme valeur le point destination.

Cette recherche d'un chemin dans l'environnement permettant au rover d'accomplir sa mission est effectuée en unifiant les préconditions de l'opérateur `!navigate` avec l'état courant :

```
(Operator (!navigate ?r ?x ?y)
 ;; preconditions
 ((rover ?r)(waypoint ?x)(waypoint ?y)
  (at ?r ?x)(can_traverse ?r ?x ?y))
 ;; effects
 ((not(at ?r ?x))(at ?r ?y))
)
```

Il existe différentes méthodes pour effectuer le processus de sélection d'un ensemble d'actions permettant de définir un chemin. Nous avons encodé dans le domaine de planification trois de ces méthodes : une recherche aveugle, une recherche heuristique de type *plus proche d'abord* et une version symbolique de l'algorithme de Dijkstra.

Recherche aveugle

La recherche aveugle consiste à sélectionner la prochaine action en choisissant comme point de passage le premier point non visité défini dans l'état courant, c'est-à-dire que ce point est choisi non déterministiquement. Dans l'algorithme du planificateur de tâches que nous utilisons, le non déterminisme est résolu par la sélection du premier prédicat présent dans la liste représentant l'état courant.

L'opérateur `!navigate` est appelé par la méthode `navigate` dont le rôle est d'éviter au rover de revisiter des points de passage déjà empruntés durant le déplacement courant en retenant une liste des points de passage précédemment visités. Ceci est fait à l'aide d'opérateurs et de prédicats de service. Ces opérateurs ne donnent pas lieu à des actions exécutables par le robot et ne modifient pas l'état au sens du problème initial. Ils modifient uniquement les prédicats de service qui peuvent être interprétés comme un état de la recherche de chemin.

```
(Method (navigate ?rover ?to)
  ;; initialisation de la recherche d'un chemin
  ((at ?rover ?from))
  ((!!visit ?from)(navigate ?rover ?from ?to)(!!unvisit ?from))
)

(Method (navigate ?rover ?from ?to)
  ;; le rover est-il à destination ? dans ce cas, ne rien faire
  ((at ?rover ?to))
  ()
  ;; le rover peut-il arriver à destination en un déplacement ?
  ((can_traverse ?from ?to))
  ((!navigate ?rover ?from ?to))
  ;; sinon déplacer le rover vers un point intermédiaire
```

```
  ((waypoint ?m)(can_traverse ?from ?m)(not (visited ?m)))
  ((!navigate ?rover ?from ?m)(!!visit ?m)(navigate ?rover ?m ?to)
   (!!unvisit ?m))
)

;; opérateur de service : marquer un point de passage comme
;; déjà visité
(Operator (!!visit ?waypoint)
  ()
  ((visited ?waypoint))
)

;; Opérateur de service : démarquer un point de passage
(Operator (!!unvisit ?waypoint)
  ()
  ((not(visited ?waypoint)))
)
```

Recherche de type plus proche d'abord

La recherche du plus proche d'abord est une recherche heuristique qui consiste à sélectionner comme prochain point de passage le point étant le plus proche du point de passage courant.

Pour coder ce type de recherche, un ensemble de prédicats spécifiant les distances entre les points de passage a été ajouté à l'état initial. Ces prédicats ont la forme :

```
(distance ?wp1 ?wp2 ?value)
```

La recherche du plus proche d'abord est codée dans le domaine en utilisant la fonctionnalité de tri (`sort-by`) des susbstitutions possibles selon une variable et un ordre donné (*cf.* annexe B) :

```
(Method (navigate ?rover ?from ?to)
  ;; le rover est-il à destination ? dans ce cas, ne rien faire
  ((at ?rover ?to))
  ()
  ;; le rover peut-il arriver à destination en un déplacement ?
  ((can_traverse ?from ?to))
  ((!navigate ?rover ?from ?to))
  ;; sinon déplacer le rover vers un point intermédiaire (plus proche)
  (:sort-by ?d < ((waypoint ?m)(can_traverse ?from ?m)
   (not (visited ?m))(distance ?from ?m ?d))
  ((!navigate ?rover ?from ?m)(!!visit ?m)(navigate ?rover ?m ?to)
   (!!unvisit ?m))
)
```

Dans cette méthode `navigate`, le prochain point de passage est choisi en triant les susbstitutions possibles sur la variable représentant la distance entre deux points de passage selon un ordre croissant. Ainsi le point de passage le plus proche est sélectionné en premier.

Recherche par l'algorithme de Dijkstra

Dans le but de reproduire le processus de recherche de chemin effectué par le planificateur spécialisé et donc de chercher l'optimalité en termes de distance parcourue à l'aide du planificateur de tâches, nous avons encodé le fonctionnement de l'algorithme de Dijkstra à l'aide d'opérateurs et de méthodes. Cette transcription est présentée en annexe A. La méthode permettant de calculer un chemin est appelée `find_path`.

L'exécution de la méthode `find_path` produit un ensemble de prédicats de la forme `(next ?wp1 ?wp2)` symbolisant les segments qui constituent le chemin solution obtenu par l'algorithme de Dijkstra.

La méthode `navigate` consiste à parcourir dans l'ordre ces segments du point initial au point destination :

```
(Method (navigate ?rover ?from ?to)
  ;; le rover est-il à destination ? dans ce cas, ne rien faire
  ((at ?rover ?to))
  ()
  ;; le rover est à l'origine ? calculer le chemin et parcourir
  ((at ?rover ?from)(not(path_computed)))
  ((find_path ?rover ?from ?to)(navigate ?rover ?from ?to)(!!clean_path))
  ;; déplacer le rover le long d'un segment du chemin solution
  ((next ?from ?mid))
  ((!navigate ?rover ?from ?mid)(navigate ?rover ?mid ?to) )
)
```

2.3.2 Méthode de recherche hybride

Dans la méthode de planification hybride, les sous-problèmes de planification d'un chemin sont délégués au planificateur spécialisé au lieu d'être résolus par le planificateur de tâches.

Le planificateur HTN et le planificateur spécialisé sont liés grâce à l'utilisation de préconditions géométriques au niveau des opérateurs de navigation. Lorsque le planificateur de tâches rencontre ce type de précondition, il envoie une requête au planificateur de déplacement. Cette requête contient le point initial et le point final du déplacement. S'il existe un chemin entre ces deux

points, alors le planificateur spécialisé renvoie une réponse positive et le processus de planification continue. Dans le cas contraire, le planificateur HTN annule la dernière action planifiée et cherche une nouvelle solution.

```
(Operator (!navigate ?r ?x ?y)
 ;; preconditions
 ((rover ?r)(waypoint ?x)(waypoint ?y)
  (at ?r ?x)(posX ?x ?c1x)(posY ?x ?c1y)
  (posX ?y ?c2x)(posY ?y ?c2y))
 ;; geometrical data
 ((from ?c1x ?c1y)(to ?c2x ?c2y))
 ;; effects
 ((not(at ?r ?x))(at ?r ?y))
)
```

Dans cette version du problème, l'état initial ne contient pas les prédicats `can_traverse`. Cependant, nous avons ajouté des prédicats spécifiant les coordonnées des points de passage. Ce sont les données géométriques qui sont transmises au planificateur de déplacement lorsqu'une précondition géométrique est rencontrée :

```
(waypoint waypoint5)
(posX waypoint5 60)(posY waypoint5 99)
```

Cet ajout est effectué en utilisant les informations disponibles dans la carte de l'environnement avant le processus de planification.

Différentes méthodes de couplage

Nous pouvons considérer plusieurs façon de lier le planificateur de tâches et le planificateur de déplacement (figure 2.6). L'approche la plus répandue consiste à définir une hiérarchie de planificateurs. Cette approche hiérarchique est l'approche classiquement utilisée lors de la conception d'une architecture robotique [Latombe, 1991]. En effet, en règle générale, un plan symbolique est préalablement calculé. Puis ce plan est raffiné par un raisonneur spécialisé, *i.e.,* un chemin géométriquement faisable satisfaisant les actions symboliques `navigate` est recherché [Ghallab *et al.*, 2004]. S'il n'existe pas de chemin satisfaisant ces contraintes, alors le planificateur de tâches calcule un nouveau plan symbolique.

Cette approche peut être améliorée en enregistrant les actions produisant des contraintes géométriques, *i.e.,* nécessitant un déplacement. Ainsi, lorsque le planificateur de déplacement ne trouve pas de solution permettant de raffiner le plan symbolique, le planificateur de tâches effectue un retour arrière à partir de l'action irréalisable géométriquement.

Une autre solution pour coupler le planificateur principal avec le planificateur de déplacements est d'entrelacer leur exécution. Dans cette approche, les requêtes géométriques spécifiant un déplacement sont envoyées par le planificateur de tâches au planificateur spécialisé dès qu'elles sont rencontrées. Si le planificateur spécialisé trouve une solution permettant de déplacer le robot, alors il renvoie une réponse positive. Dans le cas contraire, *i.e.,* si les requêtes géométriques ne sont pas satisfiables, alors il renvoie un message d'erreur. Ces appels au planificateur spécialisé peuvent être vus comme des tests de satisfaction de préconditions.

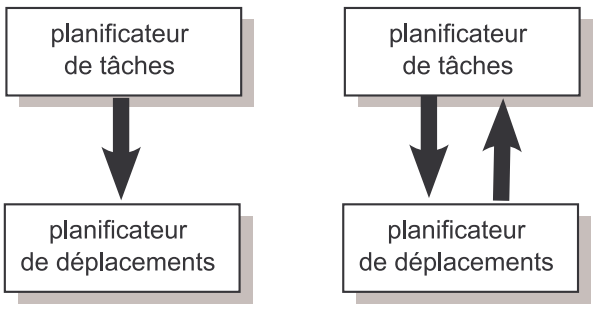


FIGURE 2.6 – couplage hiérarchique et entrelacé des planificateurs

Guidage géométrique pour le choix de l'objectif suivant

L'envoi de requêtes de calcul de chemin de manière entrelacée peut entrainer une sous-obtimalité de la distance parcourue totale lorsque l'ordre de réalisation des objectifs n'est pas fixé a priori. En effet, le planificateur spécialisé n'a pas une vue globale de l'ensemble des déplacements. Une manière d'améliorer cette dégradation de la qualité de la solution en termes de distance parcourue est de guider le choix des objectifs fait par le planificateur de tâches à l'aide d'une heuristique telle que celle permettant de sélectionner l'objectif le plus proche d'abord.

Méthode principale décrivant le choix du prochain objectif à réaliser afin d'accomplir la mission sans heuristique :

```
(Method (do-mission)
;; choix d'un objectif
 ((goal(has_soil_data ?wp)))
 ((realize-objective ?wp)(do-mission))
;; fin de la mission
 ((not(goal ?g))) ;; plus de buts à réaliser --> fin
 ()
)
```

Méthode principale décrivant le choix du prochain objectif à réaliser afin d'accomplir la mission avec heuristique :

```
(Method (do-mission)
;; choix d'un objectif
 (:sort-by ?distance < ( (goal(has_soil_data ?wp))(at ?r ?x)
   (heuristic(distance_from_waypoint ?r ?x ?wp ?distance))) )
 ((realize-objective ?wp)(do-mission))
;; fin de la mission
 ((not(goal ?g))) ;; plus de buts à réaliser --> fin
 ()
)
```

2.4 Délégation de sous-problèmes

Nous proposons de comparer les performances entre des approches pour laquelle un planificateur symbolique doit résoudre seul la totalité du problème et une approche hybride dans laquelle les sous-problèmes de recherche de chemins sont délégués au planificateur spécialisé.

2.4.1 Protocole expérimental

Nous comparons trois approches différentes. Tout d'abord une approche symbolique avec une recherche aveugle, c'est-à-dire que lors de la recherche d'un chemin, le premier prédicat modélisant un point de passage rencontré dans l'état courant est choisi. Puis une approche symbolique avec l'utilisation de l'heuristique *plus proche d'abord*. Le prédicat modélisant le point de passage le plus proche de la position actuelle du rover est choisi. Enfin, l'approche hybride dans laquelle le calcul des déplacements est délégué au planificateur spécialisé. L'approche hybride est également comparée à l'implémentation de l'algorithme de Dijkstra dans le planificateur de tâches.

Afin de comparer les méthodes classiques et la méthode de planification hybride, nous avons exécuté chacune d'elles sur quinze problèmes de complexité croissante. Dans le premier problème, le rover doit prélever un seul échantillon. Un nouvel objectif est ajouté pour chacun des problèmes suivants. Tous les objectifs sont identiques, c'est-à-dire, collecter un échantillon sur une zone spécifiée.

Comme nous utilisons un environnement statique et connu pour nos tests, le temps de génération de l'état initial n'est pas pris en compte quelque que soit l'approche utilisée.

2.4.2 Résultats

Le temps de calcul nécessaire pour trouver un plan solution pour chaque méthode est représenté sur la figure 2.7. Le modèle hybride permet de trouver une solution en un temps très inférieur à l'approche classique. De plus, cette différence de temps de calcul augmente fortement avec le nombre d'objectifs à atteindre : L'ajout d'un nouvel objectif augmente le temps de traitement d'approximativement 50 secondes pour les approches symboliques contre 4 secondes pour l'approche hybride.

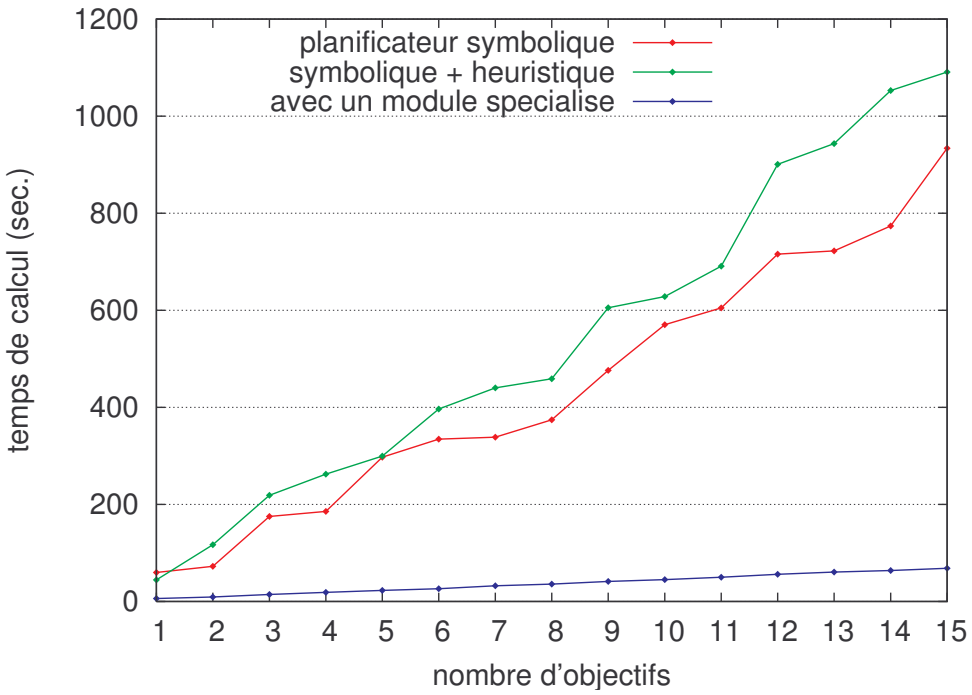


FIGURE 2.7 – Comparaison du temps de calcul pour les trois approches

La principale raison est que, pour les approches classiques, le planificateur de tâches manipule des prédicats symboliques qui ne sont pas tous en rapport avec le sous-problème de la recherche d'un chemin et raisonne donc sur un espace de recherche plus important. Dans l'approche hybride, le planificateur de déplacement est spécialisé dans la gestion de ce type de sous-problèmes et peut donc fournir une solution en un temps minimal.

La comparaison de la distance totale parcourue par le rover pour réaliser les objectifs pour les trois approches est représentée sur la figure 2.8. La distance parcourue par le robot, lorsque le modèle hybride est mis en œuvre, peut être utilisée pour juger la qualité de la solution fournie par les approches symboliques. En effet, dans ces tests, l'ordre de réalisation des objectifs est préalablement fixé. Ainsi, comme le planificateur spécialisé s'appuie sur l'algorithme de Dijkstra, la distance totale est minimale. Ces résultats corroborent

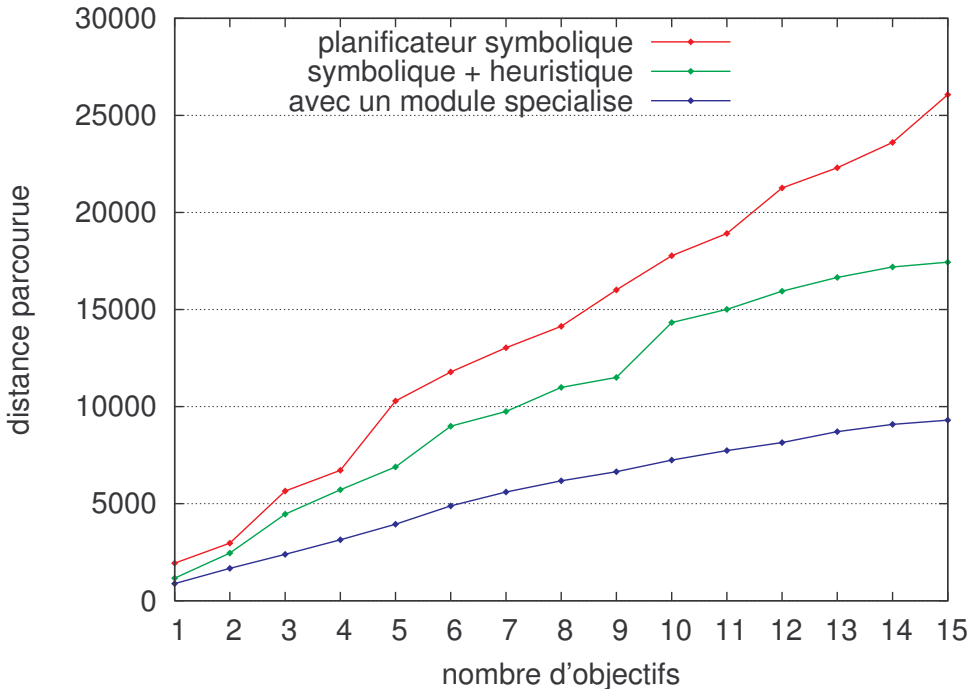


FIGURE 2.8 – Comparaison de la distance parcourue pour les trois approches

les résultats obtenus pour le temps de calcul : les sous-problèmes de recherche d'un chemin dans l'environnement pour un robot mobile ne sont pas facilement gérés par un planificateur de tâches. La principale raison est que, pour la recherche aveugle, le planificateur de tâches n'a pas de notion d'environnement et cherche un chemin en appliquant des opérateurs, c'est-à-dire en unifiant les préconditions de l'opérateur avec l'état courant du monde sans tenir compte de notions géométriques telles que la direction vers l'objectif ou encore la distance à l'objectif.

Lorsque les problèmes sont résolus en utilisant la version symbolique de l'algorithme de Dijkstra, la solution en termes de distance parcourue est optimale. Cependant, les temps de calcul des plans solutions ont été multipliés par dix en moyenne par rapport à la recherche aveugle.

2.5 Couplage de planificateurs

L'utilisation d'un planificateur spécialisé pour résoudre certains sous-problèmes bien identifiés permet de réduire le temps de calcul et d'améliorer la qualité de la solution. Cependant, la manière de coupler le planificateur spécialisé et le planificateur principal influe sur le temps de résolution du problème. Nous proposons d'étudier différentes méthodes de couplage entre un raisonnement symbolique et un raisonnement géométrique.

2.5.1 Protocole expérimental

Les trois approches pour le couplage hiérarchique, hiérarchique avec retour arrière et entrelacée sont testées sur onze exemples de complexité croissante dans lesquels un robot doit prendre des photos de dix objectifs. Le robot doit se déplacer dans l'environnement pour atteindre les objectifs. Pour chaque objectif, deux points de passage sur lesquels l'action peut être réalisée sont définis. Cependant, le premier de ces deux points, correspondant à la première action envisagée par le planificateur de tâches, peut être inaccessible. Par exemple, le point de passage est défini à l'intérieur d'un obstacle. Du point du vue du planificateur spécialisé, les coordonnées du point inaccessible ne correspondent à aucun nœud du graphe de visibilité. Dans le premier exemple, tous les points définis sont accessibles alors que, dans le second problème, le dernier objectif n'est réalisable par le robot qu'en se positionnant aux coordonnées du point alternatif fourni. Dans les problèmes suivants, un objectif supplémentaire, en partant du dernier, n'est pas accessible par le premier point. Dans le dernier problème, le robot doit se positionner au point alternatif de chaque objectif pour exécuter les actions afin de réaliser l'ensemble de la mission. Nous générons ainsi artificiellement un nombre d'objectifs non réalisables par la planification de chemins variant de 0 à 10.

2.5.2 Résultats

La figure 2.9 illustre le temps de calcul d'un plan solution pour les trois méthodes permettant de coupler le planificateur de tâches avec le planificateur de déplacements sur les onze problèmes. L'utilisation des résultats du planificateur de déplacements pour chercher un nouveau plan à partir de l'action infaisable offre de meilleurs résultats qu'une approche hiérarchique aveugle, dans laquelle un nouveau plan est calculé à partir de l'état initial. Cependant la méthode entrelaçant l'exécution du planificateur de tâches et du planificateur de déplacement est la plus performante des trois méthodes. En effet, si le point sur lequel le robot doit se positionner n'est pas atteignable, alors un déplacement vers la position alternative peut immédiatement être requis par le planificateur de tâches.

Le tableau 2.1 résume le nombre de requêtes générées par le planificateur de tâches puis envoyées au planificateur de déplacements. Ce nombre de requêtes est corrélé avec le temps de calcul. Cette corrélation entre le temps de calcul et le nombres de requêtes générées indique un effet de causalité : la principale raison de l'augmentation du temps de calcul est une augmentation du nombre de requêtes générées par le planificateur de tâches correspondant à une augmentation du nombre d'actions produites qui ne pourront pas être satisfaites géométriquement, *i.e.,* pour lesquelles il n'existe pas de chemin. La génération

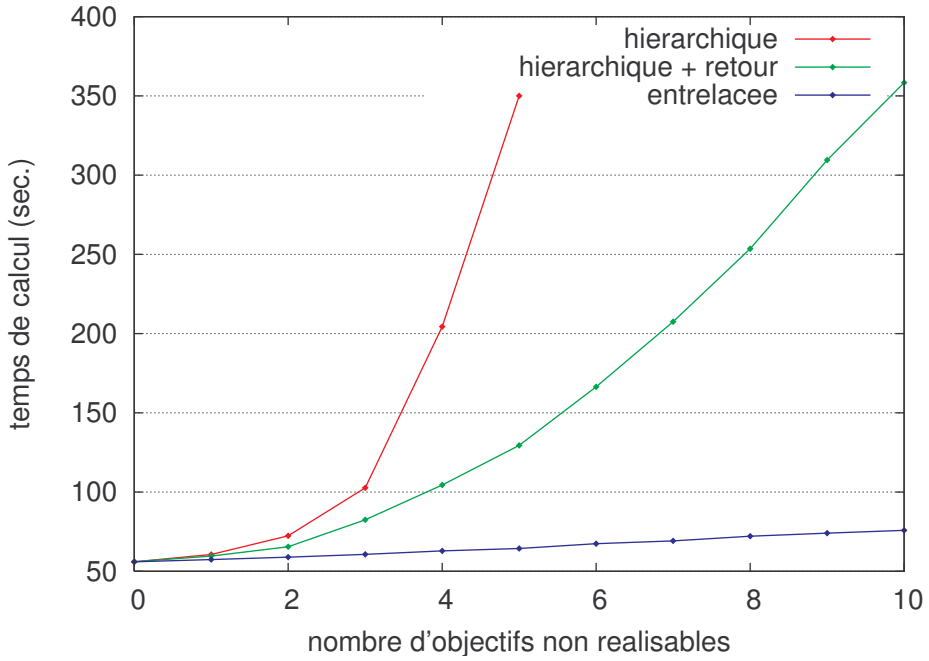


FIGURE 2.9 – Temps de calcul pour différentes méthodes de couplage

de ces actions entraine, d'une part une augmentation de l'espace de recherche symbolique et, d'autre part une augmentation du nombre de calculs de chemins.

Avec l'approche hybride entrelacée, le planificateur de tâches ne cherche pas à satisfaire le prochain objectif tant que les contraintes géométriques de l'objectif courant n'ont pas été satisfaites, c'est-à-dire tant qu'un chemin permettant de satisfaire l'objectif courant n'a pas été calculé. Alors que, avec les deux méthodes hiérarchiques, un plan symbolique satisfaisant l'ensemble des objectifs est calculé avant de rechercher les chemins permettant réellement de les satisfaire. Ainsi, avec l'approche hiérarchique classique, les mêmes contraintes géométriques, possiblement non satisfiables, sont envoyées au planificateur de déplacements tant que le planificateur de tâches n'a pas atteint la définition du déplacement alternatif à l'action impossible lors de son processus de retour en arrière.

2.5.3 Étude de complexité en fonction du nombre d'objectifs non réalisables

Nous proposons une analyse de complexité des trois méthodes pour la configuration des objectifs non réalisables du protocole expérimental. Les figures 2.10, 2.11 et 2.12 illustrent l'arbre de recherche pour un problème avec 3 objectifs non réalisables par le premier point défini. Les cercles noirs représentent les actions pour lesquelles le planificateur de déplacements peut calculer un chemin

# obj. non réalisables	hiérarchique	hiérarchique avec retour	entrelacée
0	10	10	10
1	11	11	11
2	14	13	12
3	21	16	13
4	36	20	14
5	67	25	15
6	130	31	16
7	257	38	17
8	512	46	18
9	1023	55	19
10	2046	65	20

TABLEAU 2.1 – Nombres de requêtes générées par le planificateur de tâches

et les cercles rouges représentent les actions pour lesquelles il n'existe pas de chemin. Afin de généraliser cette étude de complexité, nous définissons :

- n le nombre total d'objectifs ;
- m le nombre d'objectifs non réalisables.

Lorsque la méthode hiérarchique simple est mise en œuvre pour résoudre le problème, le planificateur de tâches n'a pas de retour sur l'action correspondante au calcul de chemin impossible. Il planifie récursivement les actions alternatives à partir de la dernière action (figure 2.10), reproduisant ainsi les mêmes actions non réalisables. Le nombre de requêtes générées (N_s) est dans ce cas :

$$N_s \;=\; n + 2^{(m+1)} - (m+2)$$

Ce qui correspond à une complexité exponentielle en fonction du nombre d'actions non réalisables.

Lorsque le retour arrière de l'approche hiérarchique a lieu sur la première action non réalisable, un nouveau sous-plan à partir de cette action est planifié (figure 2.11). Ainsi, les actions précédentes ne sont pas remises en cause. Les actions non réalisables sont remplacées successivement jusqu'à obtenir un plan solution. Le nombre de requêtes générées, N_r, est :

$$N_r \;=\; n + \frac{m.(m+1)}{2}$$

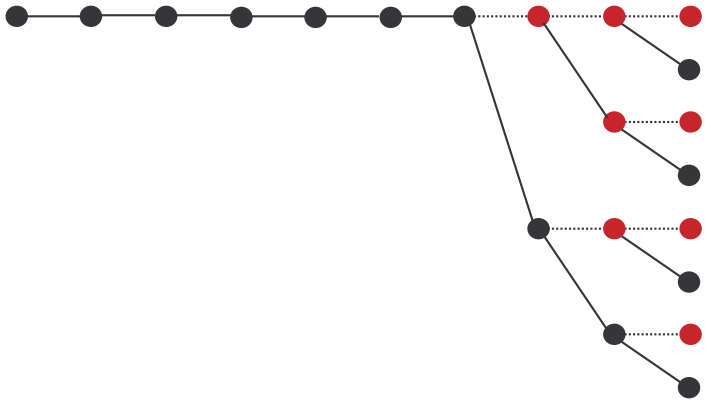


FIGURE 2.10 – Arbre de recherche pour l'approche hiérarchique simple

Ce qui correspond à une complexité quadratique en fonction du nombre d'actions non réalisables.

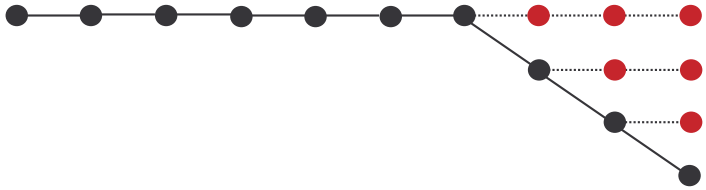


FIGURE 2.11 – Arbre de recherche pour l'approche hiérarchique avec retour arrière

Pour la méthode entrelacée, le planificateur de tâches demande le calcul d'un chemin dès qu'il cherche à planifier une action nécessitant ce calcul (figure 2.12). Ainsi, il y a au maximum une action non réalisable dans le plan à un instant donné. Le nombre total de requêtes géométriques, N_e, est donc :

$$N_e = n + m$$

C'est-à-dire que la complexité est linéaire en fonction du nombre d'actions non réalisables.

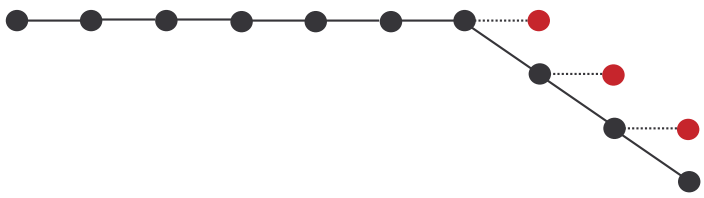


FIGURE 2.12 – Arbres de recherche pour l'approche entrelacée

2.6 Calcul heuristique pour l'amélioration de la qualité du plan

Dans ce paragraphe, nous étudions l'avantage que peut procurer au planificateur de tâches le retour d'information de la part du planificateur de mouvements. Ce retour est défini par des heuristiques qui ont pour but de guider le planificateur de tâches dans ses choix.

2.6.1 Protocole expérimental

Le problème utilisé est celui de la comparaison effectuée sur les méthodes de couplage pour l'approche hybride, sur cinq objectifs sans spécification de l'ordre de leur réalisation. Le choix de l'objectif à traiter à un moment donné est décidé en fonction des valeurs renvoyées par le planificateur de déplacements.

2.6.2 Résultat

L'utilisation d'une telle heuristique permet d'optimiser les déplacements du robot. Le gain obtenu en termes de distance parcourue en utilisant comme heuristique pour le choix du prochain objectif à traiter, l'objectif le plus proche de la position actuelle du robot, est présenté sur la figure 2.13.

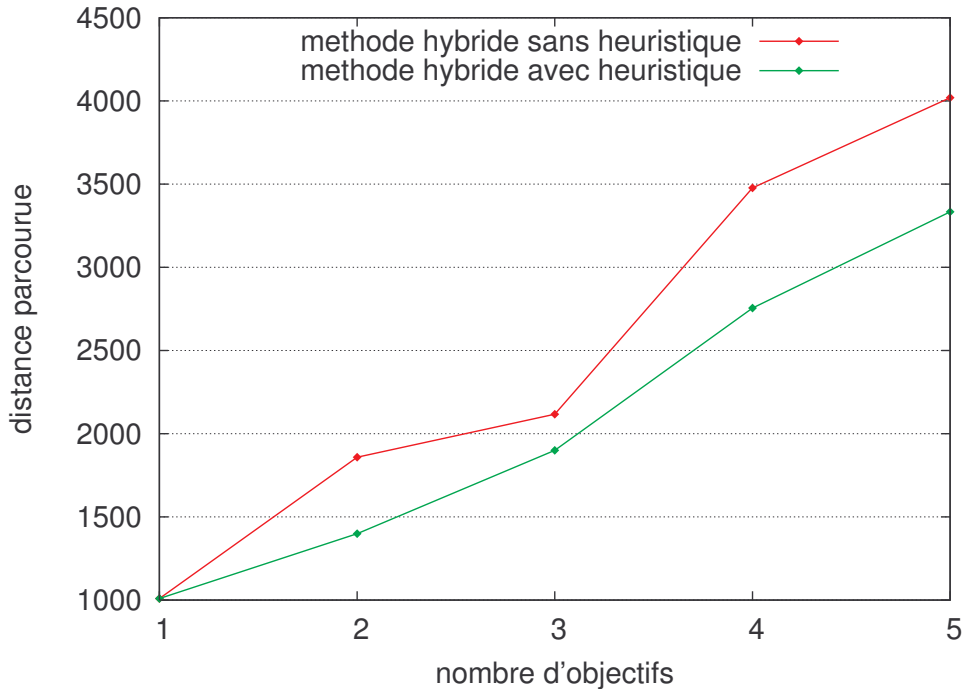


FIGURE 2.13 – Distance parcourue avec et sans heuristique

Les résultats indiquent que l'utilisation d'une heuristique lors du choix des actions à effectuer améliore la qualité de la solution en termes de distance parcourue.

2.7 Conclusion

Les expérimentations conduites dans ce chapitre indiquent que, dans le cadre de la planification de mission pour un robot mobile nécessitant le calcul de déplacements dans l'environnement, la délégation des sous-problèmes de déplacement à un raisonneur spécialisé permet de réduire le temps de calcul d'un plan solution et d'optimiser la distance parcourue. Ces premiers résultats, d'une part, justifient des pratiques en planification hiérarchique comme l'appel à des routines externes et, d'autre part, complètent les résultats similaires en planification dans l'espace des plans.

Les expérimentations indiquent également que la meilleure méthode du point de vue du temps de calcul pour coupler le planificateur de tâches avec le planificateur spécialisé est l'approche qui entrelace l'exécution des deux planificateurs. Finalement, elles montrent que la sous-optimalité de la solution qui peut survenir lorsque l'ordre de réalisation des objectifs n'est pas fixé peut être limitée par l'utilisation d'une heuristique guidant le choix de l'ordre de réalisation.

Ces constatations nous guident vers la définition d'une architecture de planification hybride pour la planification de mission pour un robot mobile faisant intervenir des aspects symboliques et des aspects géométriques. Dans ce cadre, les résultats expérimentaux conduisent à envisager :

- des requêtes, exprimées au niveau des actions, du planificateur symboliques vers le planificateur de mouvements ;
- des retours du planificateur spécialisé les plus riches possibles.

Chapitre 3

Une architecture de planification hybride

À partir des résultats des expérimentations et analyses du chapitre précédent, nous présentons dans ce chapitre les bases d'une architecture de planification dédiée au traitement des problèmes de planification pour la robotique mobile. Cette architecture intègre un planificateur de tâches et un module de raisonnement géométrique chargé de la gestion des déplacements du robot.

L'objectif de cette architecture hybride est de planifier en traitant de manière unifiée les aspects symboliques et géométriques du problème à résoudre et en permettant l'expression de contraintes géométriques qui spécifient la manière de réaliser une action.

Dans un premier temps, nous présentons un aperçu de l'architecture de planification hybride (paragraphe 3.1), puis une extension de la description des opérateurs de planification permettant d'exprimer des préconditions et effets géométriques (paragraphe 3.2). Dans un troisième paragraphe, nous présentons l'expression et la gestion des différents types de ressources qui peut être partagée par les deux modules de raisonnement (paragraphe 3.3). Puis nous proposons une algorithmique pour la gestion de la nouvelle description des opérateurs (paragraphe 3.4). Ensuite, nous proposons une méthode de résolution des contraintes géométriques, destinée au module de raisonnement spécialisé (paragraphes 3.5 et 3.6). Puis nous montrons comment le planificateur de tâches peut faire appel à l'expertise du planificateur de déplacement pour guider et optimiser la construction du plan (paragraphe 3.7). Finalement, nous introduisons le langage et les primitives d'interaction entre les deux planificateurs (paragraphe 3.8).

3.1 Aperçu de l'architecture hybride

Cette première partie présente une vue d'ensemble de l'architecture du point de vue des modules, du processus de planification et des interactions entre ces modules.

3.1.1 Les différents modules de l'architecture

L'architecture de planification hybride développée est présentée sur la figure 3.1. Elle s'articule autour de trois modules : un planificateur de tâches, un module de raisonnement géométrique et une interface.

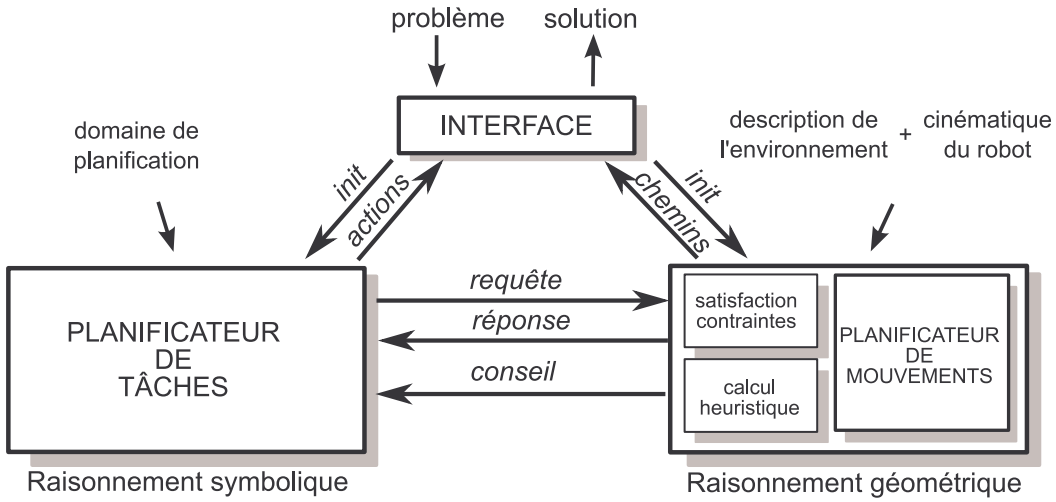


FIGURE 3.1 – Architecture de planification hybride proposée

Module de raisonnement symbolique

Le planificateur de tâches est chargé du raisonnement symbolique lors de la planification de mission. Il est paramètré par un domaine de planification qui exprime les actions que le robot peut réaliser et reçoit à l'initialisation un problème qui représente sous forme symbolique la mission à réaliser et son contexte. Les actions sont représentées par des opérateurs de planification augmentés par l'expression de contraintes géométriques. Ces contraintes ont pour objectif de définir les conditions géométriques nécessaires à la réalisation de l'action. Ce module de raisonnement symbolique peut être tout type de planificateur produisant des plans par chaînage avant. Par exemple, un planificateur HTN tel que celui présenté au chapitre précédent peut être utilisé. Dans ce cas, le domaine contient, en supplément, un ensemble de méthodes permettant de décrire l'enchainement de plusieurs actions afin de réaliser des tâches complexes.

Module de raisonnement géométrique

Ce module calcule les mouvements du robot. Il prend en entrée, lors de son initialisation, une description géométrique de l'environnement ainsi que les modèles cinématiques correspondant aux robots qui interviennent dans la réalisation de la mission. Le but principal de ce module spécialisé est de définir les mouvements nécessaires à la réalisation d'une action suite à une demande du planificateur de tâches. Il doit également être capable de fournir des conseils au planificateur symbolique afin de guider la construction du plan. Ce module de raisonnement géométrique est construit autour d'un planificateur de déplacements. Il comprend de plus un sous-module de traduction des contraintes géométriques en configurations initiales ou finales ou en méthodes d'extension des configurations lors de la recherche d'un chemin par le planificateur. Finalement, il inclut un sous-module de calcul heuristique permettant de conseiller le planificateur de tâches dans ses décisions et d'apporter son expertise lors des phases d'optimisation et de réparation du plan.

Interface

L'interface n'intervient pas directement dans le processus de planification mais a pour but d'initialiser ce processus en fournissant les données nécessaires aux deux modules de raisonnement et de fournir la solution finale en aggrégeant le plan d'actions retourné par le planificateur de tâches avec le plan des déplacements fourni par le module de raisonnement géométrique. C'est une interface entre le processus de planification et l'architecture robotique.

3.1.2 Communication entre modules

Ces différents modules de l'architecture communiquent au travers d'un formalisme commun. Ces communications peuvent être regroupées en deux classes de messages. Les messages système permettant d'assurer le bon fonctionnement de l'architecture et les messages internes au processus de planification.

Les échanges entre les deux modules de raisonnement durant le processus de planification s'appuient, d'une part, sur des **requêtes** permettant de transmettre les données géométriques et des **réponses** à ces requêtes et, d'autre part, des **conseils** qui sont des messages optionnels ayant pour objectif de guider le processus de planification vers une solution plus performante en termes de déplacements.

3.1.3 Processus de planification

Du point de vue symbolique, le processus de planification se déroule par chaînage avant. Pour appliquer une action, le planificateur de tâches vérifie

que les préconditions sont en accord avec l'état du monde relatif au nœud en développement. Cependant, certaines actions nécessitent une configuration ou un déplacement particulier du robot dans l'environnement. Lors de la sélection de ces actions, c'est-à-dire lorsqu'un opérateur à unifier contient des préconditions géométriques, le planificateur de tâches demande la satisfaction de ces préconditions au module de raisonnement géométrique. Celui-ci calcule alors les déplacements nécessaires et renvoie une réponse correspondant au résultat de ceux-ci. Si les déplacements sont possibles, alors l'action peut être appliquée. Dans le cas contraire, une autre action devra être trouvée pour réaliser la mission.

Le planificateur de tâches a la possibilité de demander des conseils au module de raisonnement géométrique lors du choix d'une action. Ce conseil doit permettre le choix de la meilleure action selon un critère géométrique issu d'un calcul heuristique.

Lorsque le plan calculé ne respecte pas les contraintes globales sur les ressources consommables du robot, les deux modules de raisonnement entrent dans une phase de réparation du plan. Cette phase a pour objectif de produire un plan alternatif, par permutation des actions, pour lequel ces contraintes sur les ressources seraient respectées.

Suite à la production d'un plan solution ou sur proposition du module de raisonnement géométrique, les deux modules de planification vont chercher à optimiser le plan solution ou le plan partiel. Cette phase d'optimisation, guidée par une fonction objectif, a pour but de fournir un plan solution globalement optimal selon un critère géométrique et nécessite un échange de propositions entre les deux planificateurs.

3.2 Description des opérateurs de planification

Le processus de planification hybride décrit dans le paragraphe précédent nécessite la redéfinition des opérateurs de planification afin de tenir compte des aspects géométriques du problème à résoudre.

Dans le but de permettre au planificateur de tâches de formuler des requêtes à l'attention du planificateur de mouvements, la description des opérateurs de planification est étendue par rapport à la définition 1.1 en ajoutant un nouveau type de préconditions : les préconditions géométriques et un nouveau type d'effets : les effets et références géométriques. Ainsi, pour être sélectionnée, une action devra respecter un ensemble de préconditions symboliques et un ensemble de préconditions géométriques. Les effets géométriques permettent de transmettre au niveau symbolique certaines valeurs qui ne peuvent être calculées que par le module de raisonnement spécialisé. Les références géométriques ont pour objectif de pointer sur une configuration géométrique particulière du

robot qui pourra intervenir dans les préconditions géométriques d'une action ultérieure à l'action courante.

La définition 3.1 présente ce nouveau formalisme d'expression d'un opérateur de planification.

Définition. 3.1 (Opérateur de planification étendu)
Un opérateur de planification étendu o est défini par le tuple :
$\langle\ name(o), symb\_pre(o), geom\_pre(o),\ geom\_\ eff(o),\ symb\_\ eff(o)\ \ \rangle$
où :
– $name(o)$ est le nom de l'opérateur ;
– $symb\_pre(o)$ est l'ensemble des préconditions symboliques ;
– $geom\_pre(o)$ est l'ensemble des préconditions géométriques ;
– $geom\_\ eff(o)$ est l'ensemble des effets et références géométriques.
– $symb\_\ eff(o)$ est l'ensemble des effets symboliques ;

Les différents ensembles d'un opérateur de planification peuvent être vides. Par exemple, celui-ci peut avoir des préconditions géométriques mais pas de préconditions symboliques, ou encore il peut définir des effets uniquement symboliques.

3.2.1 Préconditions géométriques

Nous distinguons deux types de préconditions géométriques : les **préconditions d'attitude** et les **préconditions de comportement**. Les préconditions d'attitude ont pour objectif de spécifier l'attitude, c'est-à-dire la configuration que doit avoir le robot pour débuter l'exécution une action. Les préconditions de comportement permettent d'exprimer le comportement que doit avoir le robot, en termes de déplacements, durant la réalisation de l'action.

Ainsi pour qu'une action correspondant à une instanciation d'un opérateur puisse être sélectionnée, le module de raisonnement géométrique devra définir une configuration du robot permettant d'initier la réalisation de l'action et calculer un chemin permettant au robot de se déplacer de sa configuration courante à cette nouvelle configuration. Puis, il devra calculer le déplacement du robot durant la réalisation de l'action en respectant les contraintes fournies par le planificateur de tâches (figure 3.2).

En cas d'échec de satisfaction des préconditions de comportement, le planificateur de déplacements peut mettre en action un mécanisme de recherche d'une nouvelle configuration géométrique satisfaisant les préconditions d'attitude et permettre ainsi un nouvel essai de satisfaction des préconditions de comportement. Ce mécanisme est interne au planificateur géométrique et transparent du point de vue du planificateur de tâches.

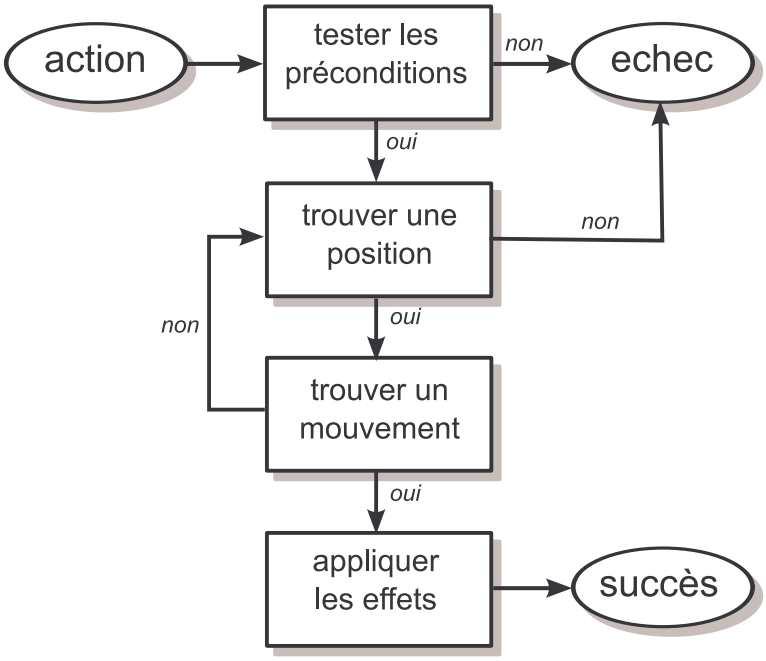


FIGURE 3.2 – Les différentes étapes de la sélection d'une action

L'opérateur de planification suivant définit la réalisation d'une action nécessitant la satisfaction de préconditions d'attitude et de comportement :

```
;; Opérateur !film_objective
(Operator (!film_objective ?r ?o ?c)
;; préconditions symboliques
  ((rover ?r)(objective ?o)
   (camera ?c)(has_camera ?r ?c) )
;; préconditions d'attitude
  ((agent ?r)(object ?o)
   (distance(?r, ?o) >= 10)
   (distance(?r, ?o) <= 20))
;; préconditions de comportement
  ((until(duration, 10))
   (constant(?r.speed)))
;; effets géométriques
  ()
;; effets symboliques
  ((has_film ?r ?o))
)
```

Cet opérateur de planification spécifie que, pour obtenir un film de l'objectif, le robot doit se placer à une distance comprise entre 10 et 20 unités de l'objectif, puis qu'il doit maintenir sa vitesse constante durant 10 unités de

temps. Cet exemple d'opérateur ne spécifie pas de comportement particulier quant au chemin que le robot doit suivre durant ces 10 unités de temps.

Les préconditions géométriques sont exprimées au niveau symbolique du planificateur de tâches et sont interprétées par le planificateur de mouvements. Ainsi, elles doivent être exprimées dans un langage interprétable par celui-ci. Le tableau 3.1 présente les concepts de base de ce langage sous une forme de Bacchus-Naur. [1]

<precondition>	: :	(<statement>* <body>)
<body>	: :	<command>* | <constraint>*
<statement>	: :	(<concept> <var_C>)
<command>	: :	c_ident(<arg>$_1$,...,<arg>$_n$)
<constraint>	: :	<elt_c> `comparator` <elt_c> | <p_function>
<concept>	: :	*agent* | *object* | *reference*
<property>	: :	<C_property> | <G_property>
<C_property>	: :	<var_C>.<parameter>
<G_property>	: :	*propriété globale*
<var_C>	: :	`variable` *symbolique du planificateur de tâches*
<parameter>	: :	*paramètre d'un agent ou objet*
<elt_c>	: :	<property> |<function> | <var_C> | `constant`
<function>	: :	f_ident(<arg>$_1$,...,<arg>$_n$)
<p_function>	: :	*constant*(<arg>) | *until*(<arg>, <arg>)
<f_ident>	: :	*entête de fonction*
<c_ident>	: :	*entête de commande*
<arg>	: :	<elt_c>

TABLEAU 3.1 – Description formelle de la syntaxe des préconditions géométriques.

Une précondition géométrique est une liste de déclarations (*statements*) liant les variables symboliques avec les éléments correspondants connus par le planificateur de déplacements, suivie du corps (*body*) de la précondition. Ce corps est :

- soit une liste de commandes d'affectation directe des composants du vecteur d'état du robot (*commands*) ;
- soit une liste de contraintes sur l'attitude relative que le robot doit rejoindre ou sur le comportement durant l'action (*constraints*).

1. La forme de Backus-Naur (BNF) est un méta-langage permettant de décrire formellement les règles syntaxiques des grammaires et langages de programmation. Voir par exemple [Naur, 1960].

Les déclarations

L'utilisation de ces déclarations permet de déléguer entièrement la gestion du robot et des données géométriques de l'environnement au planificateur de mouvements. Le planificateur de tâches ne manipule plus que des étiquettes identifiant les robots et objets. Ainsi, par exemple, il n'a pas besoin d'être informé sur la position des obstacles. Ces déclarations permettent d'établir un lien entre les étiquettes symboliques et les données géométriques.

```
;; Exemple de concepts
  (agent ?r)
  (object ?o)
```

Dans cet exemple, les déclarations sont formulées avec les prédicats `(agent ?r)` et `(object ?o)`. Le premier prédicat indique le robot qui réalisera l'action, c'est-à-dire le robot pour lequel un chemin doit être calculé. Ce prédicat n'est pas utile dans un contexte mono-agent mais apparaît dans les déclarations pour de futures extensions de l'architecture à la planification pour plusieurs agents. La seconde déclaration indique que la variable `?o` doit correspondre à un élément de la liste des objets de l'environnement connus du planificateur de mouvements. Les mots-clés `agent`, `object` et `reference` ont pour objectif de spécifier si une variable symbolique correspond à un robot mobile, à un objet statique de l'environnement ou à une configuration du robot issue d'un calcul précédent.

Ces déclarations font intervenir des **concepts**, formalisés par la définition 3.2, qui caractérisent les *agents* et *objets* qui sont manipulés par les deux planificateurs.

Définition. 3.2 (Concept)
Un concept est une étiquette sur une structure de données caractérisant un élément de l'environnement. Cette structure de données est modélisée sous la forme d'un tuple :
$\langle\ identifiant(C), type(C), \{propriétés(C)\}\ \rangle$
où :
- $identifiant(C)$ est l'identifiant du concept et permet de lier une variable symbolique à la structure de données ;
- $type(C)$ est le type du concept : `agent`, `object` ou `reference` ;
- $\{propriétés(C)\}$ est l'ensemble des champs de la structure de données caractérisant le concept.

Un ensemble de propriétés appelées **propriétés de concept** est associé à chaque concept. Ces propriétés de concept correspondent aux champs de la structure de données d'un agent ou d'un objet. Pour un agent, ce sont, d'une

part, les composantes géométriques de son vecteur d'état qui sont traitées uniquement par le planificateur de déplacements (*e.g.,* position) et, d'autre part, les ressources propres des agents (*e.g.,* énergie). Les propriétés de concepts permettent d'exprimer des contraintes sur les propriétés des robots et objets.

```
;; Exemples de propriétés de concept
  ?r.x            ;; <-- position en abscisse
  ?r.y            ;; <-- position en ordonnée
  ?r.heading      ;; <-- cap du robot
  ?r.energy_level ;; <-- niveau d'énergie du robot
```

Cet exemple exprime les propriétés de concept pour un robot dont les seules variables d'état géométriques sont sa position et son cap et, pour seule ressource, son niveau d'énergie.

Les **propriétés globales**, contrairement aux propriétés de concept, ne font pas référence directement aux propriétés d'un agent ou d'un objet, mais font référence aux propriétés du déplacement concerné. Elles permettent d'exprimer des mots-clés qui sont utilisés dans les contraintes pour restreindre le déplacement, par exemple `duration` fait référence à la durée du déplacement et `length` fait référence à sa longueur. Ces propriétés globales interviennent également au niveau des effets pour transmettre des données au module de raisonnement symbolique.

```
;; Exemples de propriétés globales
  energy_conso  ;; <-- énergie nécessaire pour le déplacement
  length        ;; <-- longueur de la trajectoire calculée
  duration      ;; <-- durée du déplacement
```

Les commandes d'affectation directe

L'objectif des commandes d'affectation directe, lorsqu'il s'agit d'une précondition d'attitude, est de permettre la gestion de l'état initial du robot par le planificateur de déplacement. Lorsque les commandes d'affectation directe interviennent dans une précondition de comportement, elles modifient la configuration du robot à la fin de l'action.

```
;; Opérateur !init_rover_attitude
(Operator (!init_rover_attitude ?r ?o)
;; préconditions symboliques
  ((location ?o)(not(initialized ?r)))
;; préconditions d'attitude
  ((agent ?r)(object ?o)
   (setProperty(?r.x,?o.x))
   (setProperty(?r.y,?o.y))
```

```
   (setProperty(?r.heading,0))
;; préconditions de comportement
  ()
;; effets géométriques
  ()
;; effets symboliques
  ( (initialized ?r) )
)
```

La description de l'opérateur `!init_rover_attitude` illustre l'utilisation d'une commande d'affectation directe du vecteur d'état du robot. Cette affectation est formulée par l'utilisation de la fonction `setProperty` dont le premier argument est une composante du vecteur d'état du robot (par exemple `?r.x`) et le second argument est une constante numérique ou une valeur déterminée par l'accès à un champ de la structure de données d'un objet du planificateur de mouvements (par exemple `?o.x`).

Dans cet exemple, le robot `r` est positionné aux coordonnées cartésiennes de l'objet `o` et son cap est fixé à 0 radian. Les caractéristiques de cet objet ont été transmises par l'interface lors de l'initialisation de l'architecture et sont donc connues par le planificateur de déplacements.

Les commandes d'affectation directe permettent également de mettre à jour les ressources d'un agent en cours de planification. L'exemple suivant illustre le rechargement de la batterie du robot, à raison d'une unité d'énergie par unité de temps, après avoir atteint la station de rechargement :

```
;; Opérateur !reload_energy
(Operator (!reload_energy ?r ?station ?qty)
;; préconditions symboliques
  ((location ?station))
;; préconditions d'attitude
  ((agent ?r)(object ?station)
   (distance(?r, ?station) = 0))
;; préconditions de comportement
  ((until(duration, ?qty))
   (r.speed = 0)
   (setProperty(?r.energy_level,?qty))
  )
;; effets géométriques
  ()
;; effets symboliques
  ()
)
```

Les contraintes géométriques

Les contraintes géométriques sont formulées par l'utilisation de comparateurs logiques spécifiant le type de la contrainte (égalité ou inégalité), par des constantes numériques ou des propriétés de concept, et par des éléments de contraintes correspondant aux fonctions géométriques. Ces fonctions sont traduites par le planificateur de mouvements en un ensemble de fonctions mathématiques.

La traduction est effectuée en sélectionnant les fonctions mathématiques correspondant aux éléments de contraintes dans une bibliothèque de fonctions. Des exemples d'éléments de contraintes et de fonctions associées sont donnés par le tableau 3.4 du paragraphe 3.5.

```
;; Opérateur !take_photo
(Operator (!take_photo ?r ?o ?c)
;; préconditions symboliques
  ((rover ?r)(available ?r)
   (location ?o)(need_photo_of ?o)
   (has_camera ?r ?c) )
;; préconditions d'attitude
  ((agent ?r)(object ?o)
   (distance(?r, ?o) >= 10)
   (distance(?r, ?o) <= 20)
   (rel_angle(?r, ?o) = cos-and-sin(pi/2)) )
;; préconditions de comportement
  ()
;; effets géométriques
  ()
;; effets symboliques
  ( (has_photo_of ?r ?o) )
)
```

L'opérateur `!take_photo`, représenté graphiquement par la figure 3.3, illustre, suite aux déclarations, une conjonction de contraintes géométriques sur l'attitude relative que le robot doit rejoindre afin de réaliser l'action `!take_photo`. Ces contraintes se réfèrent aux composantes du vecteur d'état du modèle du robot utilisé par le planificateur de déplacements et sont paramétrées par les caractéristiques des objets intervenant dans la définition de l'action. Elles sont exprimées en termes de fonctions (par exemple `distance` et `rel_angle`), de comparateurs logiques, de variables symboliques et de constantes numériques.

Le tableau 3.2 présente les éléments de contraintes de type fonction géométrique qui interviennent dans les exemples de ce paragraphe ainsi que leur signification.

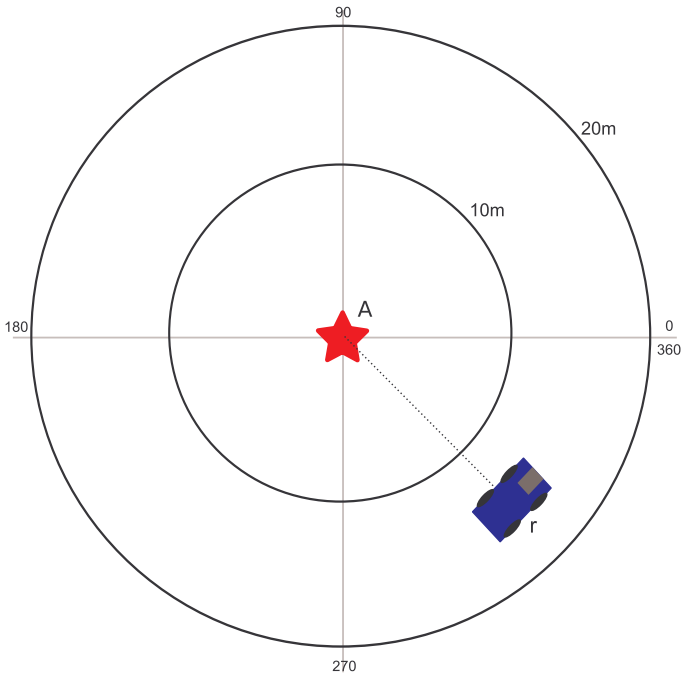


FIGURE 3.3 – Le robot `r` doit prendre une photo de l'objectif `A`

`distance(agent, objet)`	Distance euclidienne entre la position de l'agent et la position de l'objet
`rel_angle(agent, objet)`	cosinus et sinus de l'angle relatif entre le cap de l'agent et le segment de droite entre la position de l'agent et celle de l'objet
`cos-and-sin(constante)`	valeur du cosinus et du sinus de l'angle passé en argument de la fonction

TABLEAU 3.2 – Exemples d'éléments de contraintes de type fonction.

Fonctions particulières

Les préconditions de comportement font intervenir des fonctions particulières (*p_function*) qui ne sont pas présentes au niveau des préconditions d'attitude. C'est le cas des fonctions `until` et `constant`.

La fonction `until` permet de définir un critère d'arrêt pour le déplacement associé aux préconditions de comportement. Elle prend en argument une propriété globale identifiant le critère d'arrêt et une constante numérique ou une fonction géométrique représentant la valeur du critère.

```
;; Exemple d'utilisation de la fonction until
  until(duration, 10)
```

Dans cet exemple, la durée du déplacement associé aux contraintes de comportement est de 10 unités de temps.

La fonction `constant` prend en argument une propriété de concept ou un élément de contrainte de type fonction et permet de spécifier que cette propriété ou cette fonction doit rester constante durant le déplacement.

```
;; Exemples d'utilisation de la fonction constant
  constant(?r.heading)
  constant(distance(?r, ?o))
```

Le premier exemple signifie que, durant le déplacement, le cap du robot doit rester constant. Le deuxième exemple contraint le robot à rester à une distance fixe de l'objectif. Cette distance est le résultat de la satisfaction des préconditions d'attitude, *i.e.,* la distance entre la configuration de début de réalisation de l'action et l'objectif `?o`.

3.2.2 Effets et références géométriques

Le rôle du module de raisonnement géométrique est avant tout de calculer les déplacements du robot nécessaires à la réalisation des actions. Ce rôle implique la délégation de l'ensemble des aspects géométriques du problème au raisonneur spécialisé. Cependant, le planificateur de tâches peut avoir besoin de données relatives aux déplacements des robots. Par exemple, il peut avoir besoin de connaitre la quantité de carburant consommée lors d'un déplacement ou encore la distance parcourue durant ce déplacement. Par ailleurs, certaines actions peuvent faire référence à des configurations du robot précédemment calculées. Ainsi un opérateur de planification peut préciser les effets géométriques résultant de l'exécution d'une action et les références géométriques sur les configurations du robot durant cette exécution.

Le tableau 3.3 présente la syntaxe des effets et références géométriques.

<effect>	: :	<geom_effect>* <geom_ref>*
<geom_effect>	: :	(<keyword> <var_C> <var>)
<geom_ref>	: :	(<reference> <var>)
<reference>	: :	`@attitude` | `@behavior`
<keyword>	: :	*correspond à* <G_property> *des préconditions*
<var_C>	: :	*correspond à* <var_C> *des préconditions*
<var>	: :	`variable` *à unifier*

TABLEAU 3.3 – Description formelle de la syntaxe des effets géométriques.

Effets géométriques

Les effets géométriques permettent de transmettre des valeurs issues des calculs de déplacement au planificateur de tâches.

Ces effets sont des prédicats qui, à la différence des effets symboliques, sont unifiés par le module de raisonnement géométrique puis renvoyés au planificateur de tâches. Ils sont exprimés sous la forme de prédicats symboliques spécifiant la ressource et le concept, *i.e.,* l'agent concerné ; ainsi qu'une variable résultat, *i.e.,* la variable à unifier.

```
;; exemple d'effet géométrique
  (conso_energy ?r ?qty)
```

Les prédicats permettant d'exprimer ces effets géométriques sont des termes connus par le module de raisonnement géométrique : les propriétés globales. Par exemple, `conso_energy` fait référence à la consommation en carburant du robot `?r` durant l'action.

Références géométriques

Une référence géométrique permet de produire un prédicat symbolique qui est une référence sur la configuration initiale ou la configuration finale du robot lors de l'exécution d'une action. L'objectif de ces références est de pouvoir utiliser une configuration calculée pour une action antérieure lors de la réalisation d'une nouvelle action.

Comme il existe deux types de préconditions géométriques, les préconditions d'attitude et de comportement, les références géométriques peuvent faire référence soit à la configuration résultante des préconditions d'attitude, c'est-à-dire à la configuration du robot en début de réalisation d'action, soit à la configuration résultante des préconditions de comportement, c'est-à-dire à la configuration en fin d'exécution de l'action. Les références sont identifiées par le caractère `@` suivi de `attitude` pour la configuration associée aux préconditions d'attitude ou de `behavior` pour celle associée aux préconditions de comportement.

L'opérateur `!first_photo_4VP` définit une référence géométrique produite par les déplacements nécessaires à la réalisation de l'action.

```
;; Opérateur !first_photo_4VP
(Operator (!first_photo_4VP ?r ?o ?c)
;; préconditions symboliques
  ((rover ?r)(available ?r)
   (location ?o)(need_photo_of ?o)
```

```
   (has_camera ?r ?c) )
;; préconditions d'attitude
  ((agent ?r)(object ?o)
   (distance(?r, ?o) >= 50)
   (distance(?r, ?o) <= 100)
   (rel_angle(?r, ?o) = cos-and-sin(pi/2))
;; préconditions de comportement
  ()
;; effets géométriques
  ((@attitude ?refphoto))
;; effets symboliques
  ((ref_photo ?refphoto))
)
```

L'opérateur `!next_photo_4VP` utilise cette référence dans les préconditions d'attitude de la nouvelle action. Il produit également une nouvelle référence géométrique. Les contraintes géométriques font intervenir la fonction `rotate_right(?r ?o ?oldref)` qui prend en argument un robot, un objet et une référence. Cette fonction signifie que la nouvelle configuration du robot doit correspondre à la configuration obtenue après rotation dans le sens anti-trigonométrique ayant pour centre l'objet `?o` et pour rayon la distance entre `?oldref` et `?o` (figure 3.4). La contrainte stipule que l'angle de cette rotation doit être de 90 degrés.

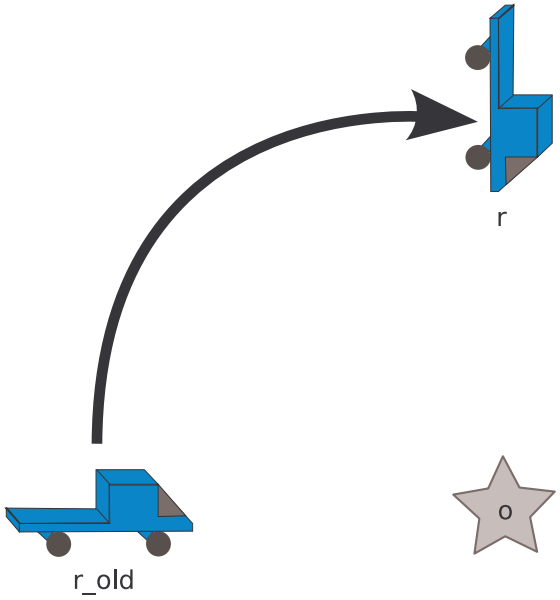


FIGURE 3.4 – La contrainte rotate_trigo(?r ?o ?oldref) = pi/2

```
;; Opérateur !next_photo_4VP
(Operator (!next_photo_4VP ?r ?o ?c)
;; préconditions symboliques
  ((rover ?r)(location ?o)
   (has_camera ?r ?c)
   (ref_photo ?oldref))
;; préconditions d'attitude
  ((agent ?r)(object ?o)(reference ?oldref)
   (rotate_right(?r, ?oldref, ?o) = pi/2))
;; préconditions de comportement
  ()
;; effets géométriques
  ((@attitude ?newref))
;; effets symboliques
  ((not(ref_photo ?oldref))
   (ref_photo ?newref))
)
```

Dans le cadre de la planification hiérarchique, lorsque la mission nécessite la réalisation de tâches complexes telles que la prise de photos selon différents angles, l'enchainement des actions peut être spécifiée à l'aide d'une méthode :

```
;; méthode take_photo_4VP
 (Method (take_photo_4VP ?r ?o ?c))
;; préconditions
  ((rover ?r)(location ?o)(camera ?c))
;; décomposition
  (:ordered
    (!first_photo_4VP ?r ?o ?c) ;; 1ere photo
    (!next_photo_4VP ?r ?o ?c)  ;; 2eme photo
    (!next_photo_4VP ?r ?o ?c)  ;; 3eme photo
    (!next_photo_4VP ?r ?o ?c)  ;; 4eme photo
  )
)
```

Utilisation des références

Lorsque les effets géométriques contiennent une référence, celle-ci n'est pas directement ajoutée à l'état résultant de l'action mais doit être utilisée dans les effets symboliques.

Cette gestion des références au niveau du planificateur de tâches permet au concepteur du domaine de planification d'utiliser des étiquettes sur les références pour produire des prédicats symboliques personnalisés qui seront ensuite insérés dans l'état courant lors de l'application des effets de l'action.

3.2.3 Récapitulatif

Afin de permettre l'expression des mouvements nécessaires à la réalisation d'une action au niveau symbolique, nous avons redéfini la notion d'opérateur de planification : en plus des préconditions et effets symboliques, l'utilisateur peut spécifier, d'une part, les préconditions géométriques nécessaires à la réalisation de l'action et, d'autre part, des effets et références géométriques engendrés par l'exécution de l'action du point de vue des déplacements du robot.

Les préconditions géométriques peuvent être de deux types : les préconditions d'attitude stipulant tout ou partie de la configuration que le robot doit rejoindre pour initier une action et, les préconditions de comportement permettant de restreindre les déplacements du robot durant l'action.

Ces préconditions géométriques sont constituées de déclarations identifiant les agents et objets intervenant dans les préconditions, suivies de commandes d'affectation directe permettant de spécifier directement le vecteur d'état du robot, et de contraintes géométriques permettant de restreindre les configurations et déplacements du robot à des sous-ensembles de l'environnement.

Les effets géométriques permettent de transmettre au module de raisonnement symboliques des données en rapport avec les calculs géométriques. Les références géométriques permettent de réutiliser la configuration de début ou de fin de réalisation d'une action précédemment planifiée. Ces références définissent ainsi un lien géométrique entre actions lors de la planification de tâches complexes dans lesquelles les sous-tâches s'appuient sur les configurations géométriques précédemment définies.

3.3 Gestion des ressources

Nous mettons en œuvre un modèle simplifié de gestion des ressources. Cependant, nous nous posons la question de la gestion des ressources intervenant dans le processus de planification. Nous distinguons trois types de ressources : les ressources symboliques gérées uniquement par le planificateur de tâches, les ressources géométriques gérées uniquement par le planificateur de déplacements et les ressources dont la gestion est partagée par les deux modules de raisonnement.

3.3.1 Les ressources symboliques

Les ressources symboliques sont des ressources qui interviennent uniquement au niveau symbolique. Elles peuvent être réutilisables ou bien consommables.

Pour la gestion des ressources consommables, nous ne définissons pas de traitement spécifique. Les contraintes sur le niveau de ressources nécessaires à la réalisation d'une action sont spécifiées à l'aide de prédicats logiques au niveau des préconditions symboliques des opérateurs.

```
;; Opérateur !process_data
(Operator (!process_data ?r)
;; préconditions symboliques
  (
   (rover ?r)
   (available_memory ?r ?qty_mem)
   (>= ?qty_mem 2048)
  )
;; préconditions d'attitude
  ...
;; préconditions de comportement
  ...
;; effets géométriques
  ...
;; effets symboliques
  ...
)
```

Dans cet exemple, pour réaliser l'action, la variable `?qty_mem` doit être supérieure ou égale à 2048.

3.3.2 Les ressources géométriques

Ces ressources correspondent aux données manipulées par le module de raisonnement géométrique. Certaines de ces ressources sont gérées uniquement par le planificateur de déplacements. C'est le cas des propriétés de concept dont les valeurs ne sont pas accessibles au planificateur de tâches.

3.3.3 Les ressources partagées

Les ressources partagées sont des ressources utilisées à la fois au niveau symbolique pour effectuer les actions et au niveau géométrique pour réaliser les déplacements du robot. Au niveau symbolique, la quantité de ressource nécessaire peut être spécifiée par les préconditions symboliques. Concernant l'utilisation de ces ressources pour les déplacements, la quantité de ressource nécessaire est connue uniquement après le calcul du déplacement, c'est-à-dire après avoir calculé des configurations vérifiant les préconditions géométriques.

Différentes solutions

Pour gérer les contraintes sur les ressources partagées, différentes méthodes sont envisageables :

- l'utilisation d'heuristiques pour définir des bornes minimales et maximales sur la consommation des ressources pendant les déplacements ;
- la mise en place d'un mécanisme de re-vérification des préconditions après les calculs géométriques ;
- la délégation de ces contraintes au module de raisonnement géométrique ;
- suivant le principe du moindre engagement, les contraintes sont vérifiées lors de la planification de l'action suivante.

L'utilisation d'heuristiques calculées par le module de raisonnement géométrique est une méthode facile à mettre en place du point de vue algorithmique. Cependant elle nécessite la définition d'heuristiques pour chaque ressource partagée. De plus, ces définitions ne sont pas triviales. Par exemple, il semble difficile d'estimer la consommation nécessaire au robot pour se déplacer d'un point à un autre de l'environnement. En effet, lorsque le planificateur de tâches demande ce calcul, le point d'arrivée n'est pas encore défini.

La deuxième méthode proposée, la re-vérification des préconditions, fait d'une part intervenir un mécanisme particulier afin de ne pas alourdir le processus de planification et, d'autre part, nécessite l'identification des ressources partagées au niveau des préconditions symboliques.

La délégation des contraintes sur les ressources au module de raisonnement géométrique n'est pas envisageable car elle sous-entend la gestion complète de ces ressources par le module même lors de la planification d'actions purement symboliques. Les conséquences au niveau des performances de l'architecture en termes de temps de résolution d'un problème pourraient être dégradées du fait d'une augmentation des messages échangés entre les deux modules.

La dernière méthode envisagée est de ne pas chercher à satisfaire les contraintes sur les ressources partagées au niveau d'une action mais durant l'action suivante qui fait appel à la ressource concernée. Dans le cas, le planificateur de tâches se "rend compte" que le niveau de la ressource est négatif et peut alors remettre en question l'action précédemment planifiée.

Nous avons fait le choix, pour l'instant, d'appliquer cette dernière méthode associée à un mécanisme de conseil de réparation du plan en cas de non respect des contraintes sur les ressources partagées (paragraphe 3.7).

Identification et utilisation des ressources partagées

La formulation d'un domaine de planification est très libre. Le concepteur peut utiliser n'importe quel terme pour représenter un objet, un concept ou encore une ressource, tant que le domaine ainsi défini et le problème associé sont cohérents. Au niveau du module de raisonnement géométrique, les ressources doivent être formellement identifiées pour être calculées. Or, ces calculs sont demandés par le planificateur de tâches. Ainsi, nous proposons de découpler les ressources partagées, c'est-à-dire qu'une même ressource physique peut avoir une étiquette différente au niveau symbolique et au niveau géométrique.

Au niveau géométrique, lors de la définition d'une action, ces ressources sont identifiées par des prédicats spécifiques qui sont des propriétés globales. Par exemple,

```
(conso_energy ?r ?conso)
```

fait référence à la consommation en énergie du robot lors des déplacements relatifs à l'action courante.

La réunion des deux consommations de ressource en une seule est effectuée au niveau des effets symboliques de l'action de manière explicite dans la définition de l'opérateur. Par exemple [2] :

```
;; Opérateur !make_sample
(Operator (!make_sample ?r ?o)
;; préconditions symboliques
  (
   (rover ?r)(location ?o)
   (level_energy ?r ?qty)
   (needed_energy_sample ?need_qty)
   (call <= ?need_qty ?qty)
  )
;; préconditions d'attitude
  (...)
;; préconditions de comportement
  (...)
;; effets géométriques
  (
   (conso_energy ?r ?conso)
  )
;; effets symboliques
  (
   (not(level_energy ?r ?qty))
```

2. La syntaxe du langage de planification et la définition des termes particuliers tels que call sont précisées en annexe B.

```
   (level_energy ?r (call - ?qty (call + ?conso ?need_qty)))
  )
)
```

Les variables `?conso` et `?need_qty` font toutes les deux référence à l'énergie consommée. La première exprime l'énergie consommée pour le déplacement et la deuxième l'énergie consommée pour le prélèvement.

3.4 Algorithme de planification associé

La définition proposée d'un opérateur de planification implique une modification des algorithmes de planification afin de prendre en compte les préconditions et effets géométriques. L'algorithme 3.1 permet d'appliquer un opérateur et produit un ensemble d'actions correspondantes.

Algorithme 3.1 InstantiateOperator(o, s, σ)

Input: o un opérateur de planification
Input: s l'état courant du monde
Input: σ une substitution
Output: A l'ensemble correspondant des actions applicables

1: $\theta \leftarrow$ FindSubstitution($symb\_pre(o), s, \sigma$) ;
2: **for each** ($\gamma \in \theta$) **do**
3: $\sigma_g \leftarrow$ substitute($geom\_pre(o), \gamma$) ;
4: sendPlanningRequest($geom\_pre(o), geom\_eff(o), \sigma_g$) ;
5: **if** (getMotionAnswer() = success) **then**
6: $eff_g \leftarrow$ getMotionEffects() ;
7: $m\_id \leftarrow$ getMotionIdentifier() ;
8: $a \leftarrow$ instantiateAction($symb\_pre(o), symb\_eff(o), eff_g, m\_id, \gamma$) ;
9: $A \leftarrow A + a$;
10: **end if**
11: **end for**
12: **return** A ;

Cet algorithme prend en entrée un opérateur de planification o à instancier, l'état courant du monde s et un ensemble de contraintes d'instanciation σ possiblement vide.

À la ligne 1, l'algorithme instancie les variables des prédicats présents dans les préconditions de l'opérateur et s'assure que les préconditions sont vérifiées dans l'état courant du monde s. Pour chaque substitution possible (ligne 2), c'est-à-dire pour chaque ensemble de couple (variable, valeur) satisfaisant les préconditions, l'algorithme instancie les variables des préconditions géométriques (ligne 3). Puis, il envoie au module de raisonnement spécialisé l'ensemble

des préconditions géométriques et l'ensemble des effets ainsi que les substitutions correspondantes (ligne 4). Si le planificateur de déplacements peut calculer un chemin respectant les contraintes géométriques (ligne 5), alors l'algorithme récupère les effets géométriques produits par le module de raisonnement spécialisé (ligne 6) et créé une nouvelle action (ligne 8).

L'application d'une action, c'est-à-dire son insertion dans le plan solution, nécessite une manipulation des effets de l'action. Les effets sont issus du processus symbolique et du processus géométrique et une simple union de ces deux types d'effets n'est pas possible. En effet, les effets géométriques ne modifient pas directement l'état courant du monde mais interviennent dans les effets symboliques. Les variables présentes dans ces effets géométriques sont utilisées pour mettre à jour les ressources partagées. L'algorithme 3.2 illustre l'application d'une action.

Algorithme 3.2 applyAction(a, s)

Input: a une action à appliquer
Input: s l'état courant du monde
Output: s' le nouvel état courant
1: $eff \leftarrow$ computeEffects($symb\_eff(a), geom\_eff(a)$) ;
2: $s' \leftarrow$ applyEffects(eff, s) ;
3: validateMotion($motion\_id(a)$) ;
4: **return** s'

Les effets de l'action sont calculés à partir des effets symboliques et des effets géométriques (ligne 1) et sont appliqués à l'état courant du monde produisant ainsi un nouvel état (ligne 2). Ce calcul est effectué en unifiant les variables libres des effets symboliques avec les subtitutions produites par le module de raisonnement géométrique. Puis les déplacements associés à l'action sont validés (ligne 3), c'est-à-dire que le planificateur de chemins est informé que ces déplacements font partis du plan solution.

Par exemple, l'exécution de l'action `!make_sample` implique une consommation d'énergie de 10 unités pour la réalisation du prélèvement et une consommation supplémentaire d'énergie pour les déplacements du robot afin de réaliser cette action. Par ailleurs, elle produit une référence géométrique sur l'attitude du robot lors du prélèvement.

```
;; Opérateur !make_sample
(Operator (!make_sample ?r ?o)
;; préconditions symboliques
  (
   (rover ?r)(location ?o)
   (level_energy ?r ?qty)
   (call <= 10 ?qty)
```

```
  )
;; préconditions d'attitude
  (...)
;; préconditions de comportement
  (...)
;; effets géométriques
  (
   (conso_energy ?conso)
   (@attitude ?refsample)
  )
;; effets symboliques
  (
   (not(level_energy ?r ?qty))
   (level_energy ?r (call - ?qty (call + ?conso 10)))
   (sample_at ?refsample)
  )
)
```

Pour cet exemple, soit s_i l'état courant du monde :

$$s_i = \left\{ \begin{array}{l} \mathsf{rover}(r1), \mathsf{location}(loc1), \\ \mathsf{level\_energy}(r1, 100) \end{array} \right\}$$

Si le module géométrique renvoie l'effet `(conso_energy 55)` et la référence `(@attitude rf01)`, alors l'état du monde devient :

$$s_{i+1} = \left\{ \begin{array}{l} \mathsf{rover}(r1), \mathsf{location}(loc1), \\ \mathsf{level\_energy}(r1, 35), \\ \mathsf{sample\_at}(rf01) \end{array} \right\}$$

Remarque. Dans ce paragraphe, nous avons présenté les algorithmes de production et d'application des actions comme découplés. Cependant lors de leur mise en œuvre, ceux-ci peuvent être regroupés ou modifiés afin d'optimiser le processus de planification en termes de requêtes échangées et de calculs du module de raisonnement géométrique. Par exemple, les requêtes peuvent être transmises uniquement au moment de l'application des actions.

3.5 Satisfaction des préconditions d'attitude

Pour trouver un vecteur d'état du robot satisfaisant les contraintes géométriques correspondantes aux préconditions d'attitude, plusieurs techniques de

résolution peuvent être appliquées. Par exemple, les algorithmes génétiques, les CSP sur domaines continus... Nous choisissons de représenter les contraintes géométriques en termes de fonctions de pénalisation. La méthode adoptée pour résoudre ces contraintes géométriques s'appuie sur une technique de programmation non linéaire classique et robuste : la descente de gradient.

3.5.1 Modèle formel de résolution

La recherche d'un état de destination satisfaisant les contraintes contenues dans la requête envoyée par le planificateur de tâches s'appuie sur la résolution d'un problème de programmation non linéaire dans lequel les variables d'optimisation sont les composantes géométriques du vecteur d'état r. Le critère à optimiser est composé de fonctions de pénalisation des contraintes.

Le problème de programmation non linéaire est défini de la façon suivante :

$$\min_{r} F(r) \tag{3.1}$$

avec :

$$F(r) = \sum_{i=1}^{i=n} G_i(f_i(r)) \tag{3.2}$$

où :

- n est le nombre de fonctions mathématiques dérivées des contraintes ;
- G_i est la fonction de pénalisation associée à la i^{me} contrainte ;
- $f_i(r)$ est une fonction déduite de la réécriture des contraintes.

Concernant la fonction de pénalité, il est nécessaire de distinguer les contraintes de type égalité des autres contraintes. Lorsque la i^{me} contrainte est une contrainte d'égalité, nous avons :

$$G_i(X) = \frac{1}{2}X^2 \tag{3.3}$$

Sinon, nous avons :

$$G_i(X) = \frac{1}{2}X^2\mathcal{H}(X) \tag{3.4}$$

où $\mathcal{H}(X)$ est la fonction de Heaviside qui prend une valeur de 0 lorsque X est négatif et 1 dans le cas contraire. Cette fonction permet de pénaliser les contraintes d'inégalité uniquement lorsqu'elles ne sont pas respectées.

La construction d'une fonction $f_i(r)$ par réécriture est faite par soustraction de la fonction associée à l'élément de contrainte `elt_c` présent à droite du comparateur, de la fonction associée à l'élément de contrainte présent à gauche. Si le comparateur est $\geq$, le signe de la fonction ainsi construite est inversé.

Par ailleurs, certaines contraintes peuvent être exprimées par plusieurs fonctions mathématiques. Par exemple, lorsque la contrainte reçue porte sur un angle, le modèle y fait correspondre deux contraintes numériques. L'une concerne le cosinus et l'autre le sinus.

3.5.2 Algorithme de résolution

La minimisation de la fonction F (équation 3.1) est effectuée par descente de gradient. L'évolution du vecteur d'état est :

$$r_{k+1} = r_k - \lambda_k \nabla F(r_k) \tag{3.5}$$

où :

- r_k est la valeur du vecteur d'état à l'itération k ;
- λ_k est le pas optimisé à chaque itération. À chaque itération, λ_k est initialisé à $\frac{F(r_k)}{\|\nabla F(r_k)\|}$ puis divisé par 2 tant que $F(r_{k+1}) > F(r_k)$
- $\nabla F(r_k)$ est la valeur du gradient de $F()$ pour le vecteur d'état à l'itération k.

Nous avons :

$$\nabla F(r) = \sum_{i=1}^{i=n} \nabla f_i(r) \nabla G_i(f_i(r)) \tag{3.6}$$

avec deux cas pour $\nabla G_i(X)$:

$$\nabla G_i(X) = X \tag{3.7}$$

lorsque $G_i(X)$ est exprimée par l'équation 3.3, et

$$\nabla G_i(X) = X\mathcal{H}(X) \tag{3.8}$$

lorsque $G_i(X)$ est exprimée par l'équation 3.4.

Le calcul de $\nabla f_i(r)$ est effectué par association de codes dérivés à chaque fonction mathématique des contraintes géométriques. En effet, les fonctions de la bibliothèque ont des codes dérivés associés (voir tableau 3.4) et la construction des fonctions $f_i(r)$ à partir des contraintes géométriques ne fait intervenir que des additions, soustractions et changements de signe. L'impact de ces opérations sur la fonction dérivée est facilement calculable. De plus, les constantes apparaissant dans les contraintes ont une dérivée nulle.

3.5.3 Exemples de fonctions et codes dérivés

Le tableau 3.4 présente les fonctions associées aux éléments de contrainte `distance` et `rel_angle` qui interviennent dans les préconditions géométriques

des exemples du paragraphe 3.2. La fonction `distance` correspond à la fonction distance euclidienne et ne fait intervenir que les positions du robot et de l'objet. La fonction `rel_angle` a pour objectif d'exprimer des contraintes d'angle entre le cap du robot et la direction du vecteur formé par le couple (position du robot, position de l'objet).

Pour ces exemples de fonctions, nous avons choisi de ne définir que les codes dérivés utilisés. Par exemple, la fonction associée à l'élément de contrainte `rel_angle` a des dérivées partielles correspondantes aux composantes du vecteur d'état du robot relatifs à la position, mais nous n'avons défini que la dérivée partielle correspondant au cap du robot, car un changement de cap peut être effectué par rotation du robot. Les codes dérivés non définis sont égaux à 0.

3.6 Satisfaction des préconditions de comportement

Contrairement aux préconditions d'attitude, les préconditions de comportement sont des préconditions qui ne définissent pas une attitude à un instant donné mais qui caractérisent une trajectoire durant la réalisation de l'action. Elles sont définies par un ensemble de contraintes restreignant le déplacement du robot et par un critère d'arrêt, défini par la fonction `until`, correspondant à la fin de réalisation de l'action.

3.6.1 Algorithmes de planification

Pour définir un chemin sans connaitre précisement la configuration terminale du mouvement, une solution possible est d'estimer cette configuration finale puis de calculer un chemin qui respecte les contraintes définies par les préconditions de comportement. Cependant, la seule information disponible pour définir cette configuration est la valeur du critère d'arrêt.

La famille d'algorithmes de planification de déplacements la plus simple à mettre en œuvre pour le calcul d'une trajectoire dont la configuration terminale n'est pas connue à l'avance sont les algorithmes par construction incrémentale dans le sens du déplacement. Le chemin est construit par étape, à partir de l'origine, en respectant les contraintes de comportement et en vérifiant que la valeur du critère d'arrêt n'est pas encore atteinte. Cette famille d'algorithmes permet de ne pas définir *a priori* une configuration terminale et de considérer les contraintes relatives à la fonction `until`.

elt_c	fonctions associées	dérivées partielles
distance(r, o)	$f_1 = \sqrt{(x_o - x_r)^2 + (y_o - y_r)^2}$	$\frac{\partial f_1}{\partial x_r} = \frac{x_r - x_o}{\sqrt{(x_o - x_r)^2 + (y_o - y_r)^2)}}$ $\frac{\partial f_1}{\partial y_r} = \frac{y_r - y_o}{\sqrt{(x_o - x_r)^2 + (y_o - y_r)^2)}}$
rel_angle(r, o)	$f_1 = \frac{(x_o - x_r) * cos(\theta_r) + (y_o - y_r) * sin(\theta_r)}{\sqrt{((x_o - x_r)^2 + (y_o - y_r)^2}}$ $f_2 = \frac{(x_o - x_r) * sin(\theta_r) - (y_o - y_r) * cos(r.\theta)}{\sqrt{((x_o - x_r)^2 + (o.y - r.y)^2}}$	$\frac{\partial f_1}{\partial \theta_r} = \frac{-(x_o - x_r) * sin(\theta_r) + (y_o - y_r) * cos(\theta_r)}{\sqrt{(x_o - x_r)^2 + (y_o - y_r)^2}}$ $\frac{\partial f_2}{\partial \theta_r} = \frac{(x_o - x_r) * cos(\theta_r) + (y_o - y_r) * sin(\theta_r)}{\sqrt{(x_o - x_r)^2 + (y_o - y_r)^2}}$

TABLEAU 3.4 – Exemples d'éléments de contrainte, fonctions associées et codes dérivés

3.6.2 Satisfaction des contraintes

Les contraintes de comportement qui ne définissent pas le critère d'arrêt sont de deux types :

- les contraintes d'égalité et d'inégalité ;
- et les contraintes faisant intervenir la fonction `constant`.

Contraintes d'égalité et d'inégalité

Les contraintes comportant un opérateur de comparaison peuvent être résolues de la même manière que les contraintes intervenant dans les préconditions d'attitude, *i.e.,* par réécriture en une fonction puis minimisation de cette fonction. Lors de la construction du chemin, chaque configuration intermédiaire est calculée à partir de cette fonction.

Contraintes constantes

Les contraintes constantes sont des contraintes spécifiant qu'une certaine propriété doit rester constante durant tout le déplacement du robot lors de la réalisation de l'action. Elles peuvent prendre en argument deux types de données : soit une propriété de concept, soit un élément de contrainte de type fonction.

Lorsque l'argument est une propriété de concept, la valeur de la variable d'état correspondante du robot est alors fixée. Les valeurs des autres paramètres du robot peuvent alors être calculées à partir des autres contraintes.

Lorsque l'argument est une fonction, le résultat de la fonction est calculée à partir du vecteur d'état défini par les contraintes d'attitude, puis la contrainte est transformée en contrainte d'égalité. Par exemple, soit la contrainte :

```
(constant(distance(?r, ?o)))
```

Cette contrainte indique que la distance entre le robot et l'objectif doit rester constante durant le déplacement. Si cette distance est, par exemple, égale à 20 unités au début de réalisation de l'action, alors cette contrainte est transformée en :

```
(distance(?r, ?o) = 20)
```

La contrainte devient donc une contrainte d'égalité et peut être résolue comme telle.

3.7 Calculs heuristiques, optimisation et réparation du plan

Des modules additionnels pour la construction d'un plan solution complètent le processus de planification décrit au paragraphe 3.2. Il s'agit d'une part de l'utilisation d'heuristiques pour guider la construction du plan et, d'autre part, d'une phase d'optimisation visant à améliorer la qualité du plan ainsi que d'une phase de réparation du plan.

3.7.1 Heuristiques pour la construction du plan

Bien que la méthode de couplage entrelacée soit plus performante que les méthodes hiérarchiques en termes de temps de calcul (*cf.* chapitre 2), la solution n'est pas globalement optimale pour les problèmes où l'ordre de réalisation des objectifs n'est pas préalablement fixé. En effet, le déplacement du robot est calculé par segments, *i.e.,* entre deux zones d'action. Ainsi, nous proposons l'utilisation d'heuristiques calculées par le module de raisonnement spécialisé permettant de guider le processus de construction du plan symbolique.

Ces calculs sont effectués par le module de raisonnement géométrique car, comme nous l'avons vu précédemment, le planificateur de tâches n'a pas de connaissance de l'environnement hormis des étiquettes sur les agents, objets et références de cet environnement.

Le planificateur de tâches envoie une **demande de conseil**. Cette requête-conseil ne modifie pas la configuration du robot mais déclenche des estimations au niveau du planificateur de déplacements dont les résultats permettent ensuite de guider le planificateur de tâches lors de la sélection d'une action.

Au niveau symbolique

Le planificateur de tâches, pour pouvoir profiter de l'expertise du module de raisonnement géométrique, doit avoir un mécanisme lui permettant de tenir compte de ces conseils lors de la sélection d'une action.

Dans le paragraphe 1.2.5 nous avons présenté la planification par recherche heuristique. Ici, nous ne cherchons pas à utiliser une heuristique pour tous les choix d'actions mais uniquement lorsque la prochaine action implique un calcul de chemin. La valeur de l'heuristique est calculée par le planificateur de déplacements puis envoyée au planificateur de tâches.

Contrairement aux préconditions géométriques, il n'est pas nécessaire de modifier les structures de représentation du langage de planification. Comme un conseil heuristique correspond à la réception d'une valeur numérique, la demande d'heuristique peut être définie par l'utilisation d'un prédicat symbolique

particulier. Ainsi, Cette demande est modélisée par l'utilisation d'un prédicat contenant le mot-clé `heuristic`. Lorsque le planificateur de tâches rencontre ce type de prédicats, il envoie un message au module de raisonnement géométrique.

Une solution possible pour permettre le choix de la prochaine action est d'affecter un coût de sélection à chaque action. Ainsi le planificateur de tâches choisira l'action de moindre coût. Ce coût peut être, par exemple, la distance approximative que doit parcourir le robot pour réaliser l'action. Par exemple :

```
(Operator (!make_sample ?r ?o)
;; préconditions symboliques
  ((rover ?r)(location ?o)
   (heuristic(distance_from_object ?r ?o ?distance)) ...
  )
;; préconditions d'attitude
  (...)
;; préconditions de comportement
  (...)
;; effets géométriques
  (...)
;; effets symboliques
  (...)
;; coût de l'action
  (cost ?distance)
)
```

Si nous faisons le choix de représenter le domaine de planification symbolique de manière hiérarchique (HTN), c'est-à-dire à l'aide de méthodes exprimant comment décomposer des tâches en sous-tâches et à l'aide d'opérateurs correspondant aux tâches élémentaires, l'appel à un calcul heuristique peut être défini au niveau des préconditions des méthodes. Ainsi, en plus de l'expertise humaine décrivant la manière de décomposer les tâches, le planificateur peut utiliser l'expertise du raisonneur spécialisé pour guider cette décomposition.

Par exemple, dans la description de la méthode suivante, l'heuristique `distance_from_object` est utilisée et permet de choisir comme prochain objectif, l'objectif le plus proche de la position actuelle du robot.

```
(Method (choose-objective)
;; choix d'un objectif
 (:sort-by ?distance < (
   (goal (has_soil_data ?wp))
   (heuristic (distance_from_object ?r ?wp ?distance)) )
 )
 ((realize-objective ?wp)(choose-objective))
)
```

Dans cet exemple, à chaque appel récursif de la méthode `choose-objective` et pour chaque unification de la variable `?wp`, une demande de calcul heuristique est envoyée. Suite à la réception de l'ensemble des réponses des calculs puis de l'unification des valeurs avec la variable `?distance` dans les prédicats `(distance_from_waypoint ?r ?wp ?distance)`, le planificateur de tâches trie ces prédicats selon la valeur des différentes distances puis appelle la méthode `realize-objective` avec comme argument l'objectif le plus proche de la position actuelle du robot.

Au niveau géométrique

Cette demande de calcul heuristique de la part du planificateur de tâches au module de raisonnement spécialisé nécessite un traitement géométrique. Ce traitement peut correspondre au calcul d'un chemin qui est ensuite annulé. Cette approche est la plus simple à mettre en œuvre lorsque le module spécialisé est uniquement composé d'un planificateur de mouvements standard, mais est coûteuse. Nous avons fait le choix de développer un sous-module responsable des calculs heuristiques. L'objectif de ce sous-module est de fournir une réponse au planificateur de tâches en un temps inférieur au temps nécessaire pour calculer un chemin.

Par exemple, dans le cas de l'heuristique `distance_from_object` un simple calcul de distance euclidienne entre la position du robot et la position de l'objet concerné est effectué.

3.7.2 Optimisation du plan

La phase d'optimisation du plan est une phase qui vise à améliorer la qualité du plan du point de vue des déplacements. C'est ainsi une phase optionnelle. Elle a lieu à la fin du processus de planification, lorsqu'un plan solution a été calculé.

Critère d'optimisation

Le critère d'optimisation du plan doit être un critère géométrique. En effet, la phase d'optimisation est effectuée par le module de raisonnement géométrique en proposant un nouvel ordre des déplacements entre les actions planifiées. Cette permutation des actions implique le calcul de nouveaux chemins entre la fin de réalisation des actions et le début des actions suivantes. Le déplacement du robot durant la réalisation d'une action n'est, quant à lui, pas modifié.

Le critère d'optimisation le plus évident est la distance totale parcourue. Cependant d'autres critères peuvent être envisagés tels que la quantité de carburant consommée ou encore le temps nécessaire à la réalisation de la mission.

Ces critères correspondent aux ressources partagées entre les deux planificateurs (paragraphe 3.3).

Soit f la fonction de coût associée à un déplacement selon le critère d'optimisation. Le coût total de la mission est :

$$\sum_i (f(q_{end_{i-1}}, q_{start_i}) + f(q_{start_i}, q_{end_i}))$$

où q_{start_i} est la configuration du robot au début de la réalisation de l'action i et q_{end_i} sa configuration en fin de réalisation. Ainsi le coût de la mission est la somme des déplacements entre actions plus la somme des déplacements durant la réalisation d'une action, avec q_{end_0} la configuration initiale du robot.

Durant le processus d'optimisation, il est trop coûteux de calculer une valeur exacte du critère entre chaque configuration. Ainsi, nous utilisons une heuristique. Soit h la fonction heuristique associée au critère choisi, la valeur du critère pour le déplacement entre deux configurations q_{i-1} et q_i est :

$$v = h(q_{end_{i-1}}, q_{start_i}) + f(q_{start_i}, q_{end_i})$$

Par exemple, si on cherche à optimiser la distance, alors la valeur du critère est :

$$v = h(q_{end_{i-1}}, q_{start_i}) + d(q_{start_i}, q_{end_i})$$

où h est la distance euclidienne entre deux configurations et d est la longueur du chemin parcouru durant la réalisation de l'action i.

Processus d'optimisation

Le problème d'optimisation de la mission est similaire au problème du voyageur de commerce. L'objectif est de "visiter" toutes les configurations associées aux actions une seule fois en minimisant la somme des coûts de trajet. Le temps mécessaire pour calculer toutes les permutations possibles entre les différentes actions est exponentiel. En effet, le problème du voyage de commerce est un problème NP-complet.

Ainsi nous proposons d'utiliser un algorithme glouton pour définir un nouvel ordonnancement des actions. Cependant, la phase d'optimisation doit respecter un certain nombre de contraintes entre les actions : les contraintes d'ordre et les liens causaux.

Les contraintes d'ordre sont envoyées par le planificateur de tâches au début du processus d'optimisation. Elles spécifient les relations de précédence entre actions sous la forme $a_i \prec a_j$, *i.e.,* l'action a_i précède l'action a_j. Les liens causaux précisent les relations de type producteur / consommateur. Par exemple, la relation $a_i \xrightarrow{p} a_j$ indique que le prédicat p produit dans les effets de l'action a_i

intervient dans les préconditions de l'action a_j. Ces liens causaux sont vérifiés par le planificateur de tâches.

L'algorithme 3.3 illustre le processus d'optimisation du plan solution.

Algorithme 3.3 PlanOptimization($\mathcal{A}$, $\prec$)

Input: $\mathcal{A} = \{a_0, \ldots, a_n\}$ est l'ensemble des actions du plan
Input: $\prec$ est l'ensemble des contraintes d'ordre
1: $Result \leftarrow \emptyset$;
2: $cur \leftarrow R_{init}$;
3: **while** ($\mathcal{A} \neq \emptyset$) **do**
4: $i \leftarrow$ select_best_action_index($\mathcal{A}$, cur, $Result$, $\prec$) ;
5: $path \leftarrow$ compute_path(cur, q_{start_i}) ;
6: **if** ($path \neq$ null) **then**
7: send_advice(optimization, a_i) ;
8: **if** ($advice.answer = ACK$) **then**
9: add(a, $Result$) ;
10: remove(a_i, $\mathcal{A}$) ;
11: $cur \leftarrow q_{end_i}$
12: **end if**
13: **end if**
14: **end while**

Le déroulement de l'algorithme d'optimisation est le suivant : À partir de l'ensemble des actions nécessitant un déplacement dans le plan précédemment calculé, l'algorithme sélectionne la meilleure action par rapport au critère d'optimisation (ligne 4). Puis, il calcule un nouveau chemin entre la configuration finale de l'action précédente (ou de la position initiale du robot pour la première action) et la configuration initiale de la nouvelle action (ligne 5). S'il existe un chemin permettant de rejoindre la configuration alors l'algorithme soumet l'action au planificateur de tâches pour vérification des liens causaux (ligne 7). Si l'action est validée, le processus est itéré et l'action suivante est recherchée en utilisant comme configuration actuelle du robot la configuration finale de l'action validée.

La sélection de la meilleure action par rapport au critère d'optimisation est illustrée par l'algorithme 3.4.

Pour chaque action de l'ensemble $\mathcal{A}$, l'algorithme calcule le coût heuristique d'un nouveau chemin (ligne 5) en additionnant le résultat de l'heuristique pour atteindre la configuration de début d'action au coût de déplacement durant la réalisation de l'action. Si le coût de déplacement pour cette nouvelle action est inférieur aux coût de déplacement des autres actions, alors l'ensemble des contraintes d'ordre dans lesquelles l'action apparait en partie de droite est sélectionné (ligne 7). Puis l'algorithme vérifie que l'action apparaissant à gauche,

Algorithme 3.4 select_best_action_index($\mathcal{A}$, *cur*, *Result*, $\prec$)

```
1:  i ← 0 ;
2:  best_v ← heuristic(cur, q_start_i) + value(q_start_i, q_end_i) ;
3:  for each (a ∈ A) do
4:     j ← index(a) ;
5:     val ← heuristic(cur, q_start_j) + value(q_start_j, q_end_j)
6:     if (val < best_v) then
7:        listConstraints ← getOrderConstraints(a, ≺) ;
8:        isSatisfied ← true ;
9:        for each (c ∈ listConstraints) do
10:          if (left_part(c) ∉ Result) then
11:             isSatisfied ← false ;
12:          end if
13:       end for
14:       if (isSatisfied = true) then
15:          i ← j ;
16:       end if
17:    end if
18: end for
19: return i ;
```

i.e., l'action précédente, appartient déjà au nouveau plan (ligne 10). Si l'insertion de l'action dans le nouveau plan respecte les contraintes d'ordre, alors l'action devient la nouvelle meilleure action. L'algorithme se termine lorsque toutes les actions ont été testées. Le coût de déplacement de l'action choisie est alors localement optimale.

Du point de vue du module de raisonnement géométrique, les actions intervenant dans ce processus d'optimisation sont uniquement les actions faisant intervenir un déplacement. Au niveau du planificateur de tâches, à la réception des actions envoyées par le module de raisonnement géométrique, celui-ci compose le plan optimisé à partir des actions reçues et des actions ne nécessitant pas de déplacements en respectant les contraintes d'ordre et les liens causaux.

Ces deux algorithmes sont complétés par un mécanisme empêchant qu'une action déjà sélectionnée et testée soit de nouveau sélectionnée.

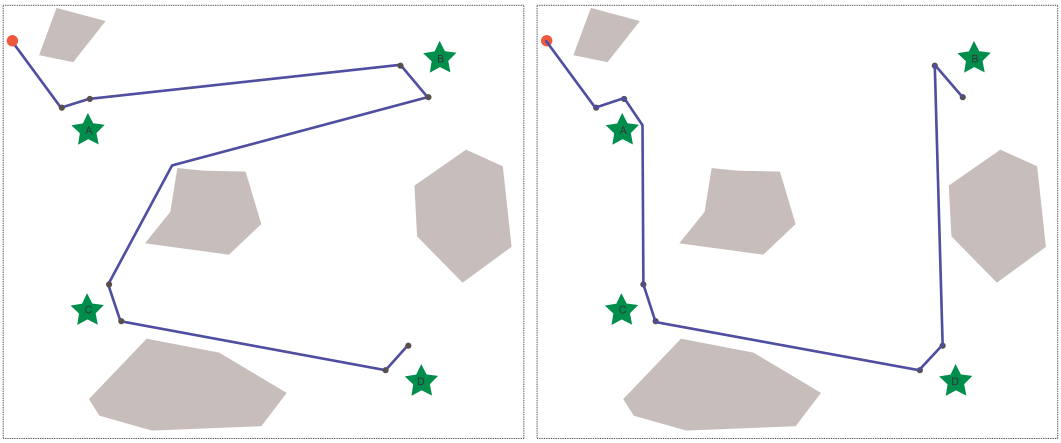
Vérification du plan optimisé

Comme le planificateur de tâches reçoit les actions dans l'ordre du plan optimisé, il peut vérifier simplement les liens causaux. Lorsqu'il reçoit la première action, il l'applique sur l'état initial. À chaque nouvelle action reçue, il applique celle-ci sur l'état courant. Lorsqu'un défaut apparait alors l'action est rejetée.

Cette vérification se décompose en trois traitements successifs. Lors de la réception d'une action a_g envoyée par le module de raisonnement géométrique, le planificateur de tâches :

1. vérifie qu'il peut appliquer toutes les actions purement symboliques a_s liées par une contrainte d'ordre de la forme $(a_s \prec a_g)$ et modifie l'état courant ;
2. vérifie qu'il peut appliquer l'action a_g et modifie l'état courant ;
3. vérifie qu'il peut appliquer toutes les actions purement symboliques a_s liées par une contrainte d'ordre de la forme $(a_g \prec a_s)$ et produit modifie l'état courant.

La figure 3.5 illustre les séquences de déplacements du robot avant optimisation du plan et après. Les étoiles représentent les objectifs ; le point rouge, la configuration initiale du robot ; et les points noirs, les configurations de début et de fin d'action.



(a) avant optimisation (b) après optimisation

FIGURE 3.5 – Déplacements du robot durant la mission avant et après la phase d'optimisation

3.7.3 Réparation du plan

Plusieurs types de défauts peuvent intervenir lors de la création du plan. Le défaut peut être purement symbolique, *i.e.*, lors de la sélection d'une action, les préconditions ne sont pas respectées et, dans ce cas, le planificateur de tâches cherche à appliquer une autre action. Le défaut peut être purement géométrique, par exemple, il n'existe pas de chemin pour réaliser l'action. Dans ce cas l'action courante est rejetée et le planificateur de tâches essaie de planifier une nouvelle action. Le défaut peut concerner les ressources partagées (*cf.* paragraphe 3.3).

Par exemple, il n'y a pas assez de carburant pour continuer la mission. Dans ce cas, une phase de réparation du plan est entamée.

Ces défauts sur les ressources partagées peuvent être réparés de plusieurs manières :

- Symboliquement. Le planificateur de tâches gère le défaut comme un défaut purement symbolique. Il cherche à planifier une autre action puis entame une phase de retour en arrière jusqu'à obtenir un plan sans défaut.
- Géométriquement. Le problème est traité par le module de raisonnement géométrique en remettant en cause l'ordre d'exécution des actions précédemment planifiées.

L'inconvénient de l'approche symbolique est que toutes les actions peuvent être remises en cause même celle ne faisant pas intervenir de ressources partagées. De plus, comme la recherche d'un nouveau plan satisfaisant les ressources n'est pas guidée, celle-ci peut être longue et remonter jusqu'à la première action du plan.

La réparation du défaut par le module de raisonnement géométrique fait intervenir le processus d'optimisation du plan précédemment présenté. En effet, comme la consommation des ressources, au niveau du module de raisonnement géométrique, correspond à une évolution strictement décroissante, diminuer cette consommation peut permettre d'éviter le défaut. Ainsi, lorsque le module de raisonnement géométrique reçoit une demande de réparation, il démarre le processus d'optimisation sur le plan partiel courant.

Lorque la phase d'optimisation est terminée, si le défaut est réparé alors le processus de planification continue en utilisant comme début de plan solution le plan optimisé.

Si la réparation du plan par optimisation du parcours du robot n'est pas possible, la réparation est alors effectuée de manière symbolique, c'est-à-dire de manière classique. Ainsi, l'objectif de cette phase de réparation par le module de raisonnement géométrique est d'utiliser les connaissances sur l'environnement pour guider et accélérer la résolution des défauts sur les ressources du robot.

3.8 Interactions au sein de l'architecture

Au paragraphe 3.2, nous avons défini un formalisme permettant d'exprimer les préconditions géométriques en termes de concepts, propriétés de concept, commandes et contraintes. Puis au paragraphe 3.7 nous avons présenté comment exprimer les demandes de calcul d'heuristiques. Ces différents éléments de langage, ou primitives de communication, sont encapsulés dans des messages

transmis entre les planificateurs. Ces messages sont exprimés sous la forme de **requêtes**, de **réponses** à ces requêtes et de **conseils**.

3.8.1 Aperçu des interactions

La figure 3.6 illustre l'ensemble des messages échangés par les différents modules. Les rectangles représentent les différentes phases du processus de production d'un plan solution et les flèches représentent les messages permettant de passer d'un traitement à un autre.

Les modules producteurs des messages sont représentés par le tableau 3.5 et la signification de ces messages est donnée par le tableau 3.6.

`I.`	interface
`TP.`	module de raisonnement symbolique
`PP.`	module de raisonnement géométrique

TABLEAU 3.5 – Expéditeurs des messages

`system_req`	requête systême
`planning_req`	requête de planification
`cancel_req`	requête d'annulation
`advice_req`	requête de conseil
`heur_advice`	conseil heuristique
`optim_advice`	conseil d'optimisation
`repair_advice`	conseil de réparation
`x_success`	accusé de succès
`x_failure`	constat d'échec
`x_content`	retour de données

TABLEAU 3.6 – Principaux codes et significations des messages échangés, avec x ∈ {req, advice, planning}

Dans la suite, nous détaillons la signification de ces différents messages.

Remarque. Afin de généraliser et de clarifier les interactions au sein de l'architecture hybride de planification, nous découplons les envois des requêtes de planification des préconditions d'attitude et des préconditions de comportement ainsi que les demandes d'unification des effets géométriques. Cependant, ces deux types de préconditions géométriques ainsi que les effets géométriques peuvent être transmis à l'aide d'une seule requête de planification comme indiqué dans l'algorithme 3.1.

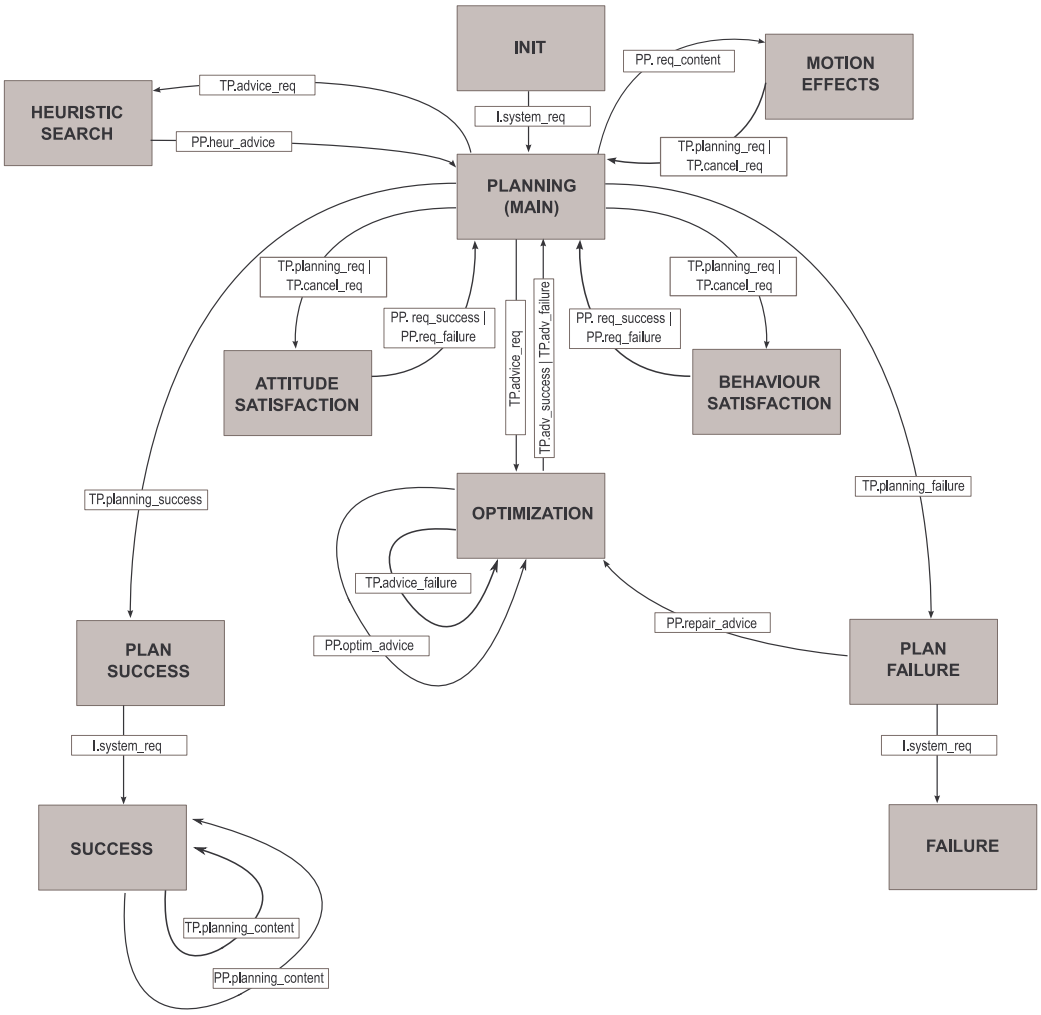


FIGURE 3.6 – Automate de contextualisation du dialogue

3.8.2 Les requêtes

Nous pouvons distinguer quatre types de requêtes permettant aux différents modules de l'architecture d'interagir : les **requêtes de planification** et les **requêtes d'annulation** pour les échanges entre le planificateur de tâches et le module de raisonnement géométrique ; les **requêtes système** permettant à l'interface de gérer l'ensemble du processus de planification ; et les **demandes de conseil** heuristique.

Les requêtes de planification

Les requêtes de planification permettent au planificateur de tâches de transmettre au module de raisonnement géométrique, d'une part, les préconditions géométriques qui doivent être satisfaites et d'autre part, les effets géométriques.

Définition. 3.3 (Requêtes de planification)
Une requête de planification R est définie comme un tuple
$< Type(R), Agent(R), Id_{action}(R), \mathcal{C}(\mathcal{R}) >$

- $Type(R)$ est le type de requête. Par exemple, **attitude** pour définir une requête de vérification des préconditions d'attitude, **behavior** pour une requête concernant les préconditions de comportement, ou **effects** pour une requête de demande d'unification des effets géométriques ;
- $Agent(R)$ permet d'identifier l'agent concerné par la requête dans un cadre multi-agent ;
- $Id_{action}(R)$ est un identifiant d'action permettant de maintenir une cohérence entre la résolution du planificateur symbolique et celle du planificateur spécialisé ;
- $\mathcal{C}(\mathcal{R})$ est l'ensemble des contraintes géométriques qui doivent être satisfaites durant les phases de déplacement des robots, ou l'ensemble des prédicats à unifier dans le cas des effets géométriques.

Exemple de requête de planification :
`planning_req(attitude, robot1, 1, {distance(robot1, o)>=10),...})`

Cette requête signifie que l'agent considéré est `robot1` et qu'il doit se positionner à plus de 10 unités de distance de l'objet `o`. Les champs `attitude` et `1` indiquent que cette requête correspond aux préconditions d'attitude de la première action envisagée par le planificateur de tâches.

L'accusé de réception retourné par le module de raisonnement géométrique contient le résultat de la requête. Ce résultat est de type *success* ou *failure* lorsqu'il correspond à une demande de satisfaction des préconditions d'attitude ou de comportement. Il est de type *content* et contient un ensemble de substitutions lorsqu'il correspond à une demande d'unification des effets géométriques.

Les requêtes d'annulation

Les requêtes d'annulation sont envoyées par le planificateur de tâches au module de raisonnement géométrique lorsqu'une action symbolique est annulée. Elles permettent de demander l'annulation des déplacements correspondant à l'action concernée. Le vecteur d'état du robot est alors remplacé par l'ancien vecteur d'état.

Les requêtes système

Les messages système sont utilisés par l'interface pour donner des instructions aux différents planificateurs et par les planificateurs pour transmettre leurs résultats finaux. Ces instructions permettent d'initialiser, de guider et de finaliser la construction du plan.

Lors de l'initialisation, l'interface transmet le domaine et le problème de planification au planificateur de tâches et la carte de l'environnement ainsi que les modèles cinématiques des robots qui interviendront durant la mission au module de raisonnement géométrique. À la fin du processus de planification, l'interface demande les résultats à chacun des modules et produit un plan global comportant des actions symboliques et des actions géométriques transmises par les planificateurs.

Les demandes de conseils

Ces demandes sont des requêtes envoyées par le planificateur symbolique au planificateur spécialisé afin de lui demander un **conseil**. Elles sont utilisées lorsque le planificateur symbolique est face à un choix relatif à l'environnement comme, par exemple, *faire quelque chose au point A ou faire quelque chose au point B*.

3.8.3 Les conseils

Les **conseils** sont des messages envoyés par le module de raisonnement géométrique au planificateur de tâches. Ils permettent l'apport de connaissances de la part du module de raisonnement géométrique afin d'aider et d'optimiser le processus de planification. Ce sont des messages optionnels envoyés par le module spécialisé suite à une demande du planificateur de tâches.

Nous pouvons distinguer deux types de conseils : les **conseils heuristiques** qui sont fournis à la demande du planificateur de tâches et ont pour objectif de guider le choix d'une action et les **conseils d'optimisation** qui expriment certaines propositions du module de raisonnement géométrique en vue d'optimiser le plan solution.

Définition. 3.4 (Conseil)
Un conseil A est défini comme un tuple
$< Type(A), Content(A), Agent(A), set\{Id_{action}(A)\} >$

- $Type(A)$ est le type de conseil : heuristique ou optimisation ;
- $Content(A)$ est le contenu du conseil ;
- $Agent(A)$ permet d'identifier l'agent concerné par le conseil dans un cadre multi-agent ;
- $set\{Id_{action}(A)\}$ est un ensemble d'identifiants d'actions.

Les conseils heuristiques et les conseils d'optimisation peuvent être vus comme des requêtes optionnelles : s'ils ne sont pas pris en compte, un plan solution peut être construit mais celui-ci pourra être de moindre qualité.

Conseil heuristique

Un conseil heuristique est renvoyé par le sous-module de calcul heuristique du raisonneur géométrique au planificateur de tâches uniquement suite à la réception d'une demande de conseil. Ce message contient le résultat du calcul heuristique. $Content(A)$ est alors une valeur numérique.

Exemple de conseil heuristique :
`heur_advice(100, robot1,{})`

Ici, la réponse à la demande heuristique de calcul de distance entre le robot et un objectif est la valeur 100.

Conseil d'optimisation

Les conseils d'optimisation sont envoyés par le module de raisonnement géométrique à la fin du processus de planification. En effet, à la fin du processus de planification, le module de raisonnement géométrique a une entière connaissance des déplacements qui devront être effectués durant la mission. Ainsi, il peut proposer des alternatives d'ordre de réalisation des différentes tâches afin d'optimiser la distance totale parcourue durant la mission. $Content(A)$ est alors un code d'opération à effectuer sur les tâches.

Exemple de conseil d'optimisation :
`optim_advice(reverse, robot1,{1,2})`

Dans cet exemple, le planificateur de déplacement demande au planificateur symbolique s'il peut inverser la réalisation des tâches 1 et 2.

À la réception du conseil, le planificateur de tâches vérifie s'il est applicable. Pour être appliqué, un conseil ne doit pas remettre en cause les contraintes d'ordre et les contraintes d'instanciation entre actions.

3.8.4 Exemple d'interactions

Comme support à l'étude des interactions entre les différents modules de l'architecture de planification, nous prenons un exemple simple de mission pour un robot mobile et deux objectifs.

Exemple

L'environnement de la mission est représenté par la figure 3.7. Dans cette mission, un rover doit prendre des photos de l'objectif A (action 1) et de l'objectif B (action 2). Il doit également prélever des échantillons de roches sur l'objectif A (action 3).

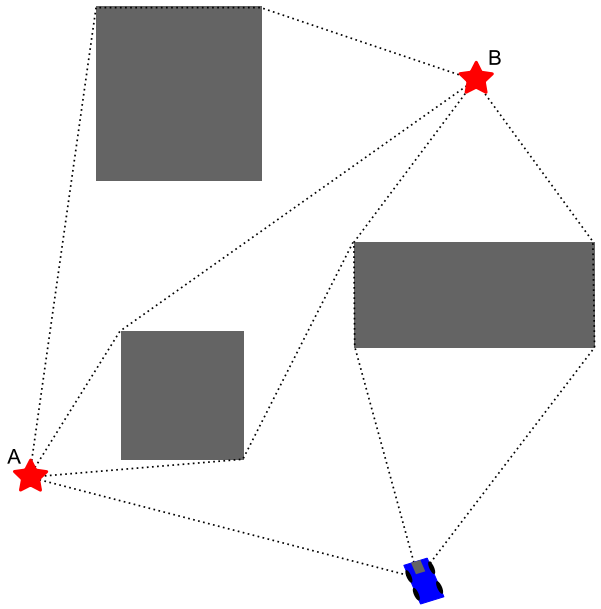


FIGURE 3.7 – Exemple de mission avec deux objectifs

Messages échangés

Le diagramme de séquence (figure 3.8) représente un exemple de séquence des messages échangés entre le planificateur de tâches et le module de raisonnement géométrique durant la construction d'un plan solution.

Après la phase d'initialisation (ligne 1), le planificateur de tâches essaie de planifier la première action (filmer l'objectif A). Il envoie une requête (ligne 2) au planificateur de déplacements afin de vérifier les préconditions d'attitude de cette action. La requête contient les contraintes géométriques spécifiées dans l'opérateur de planification `film_objective`. Le planificateur de déplacement répond positivement (ligne 3), *i.e.*, les mouvements nécessaires au positionnement

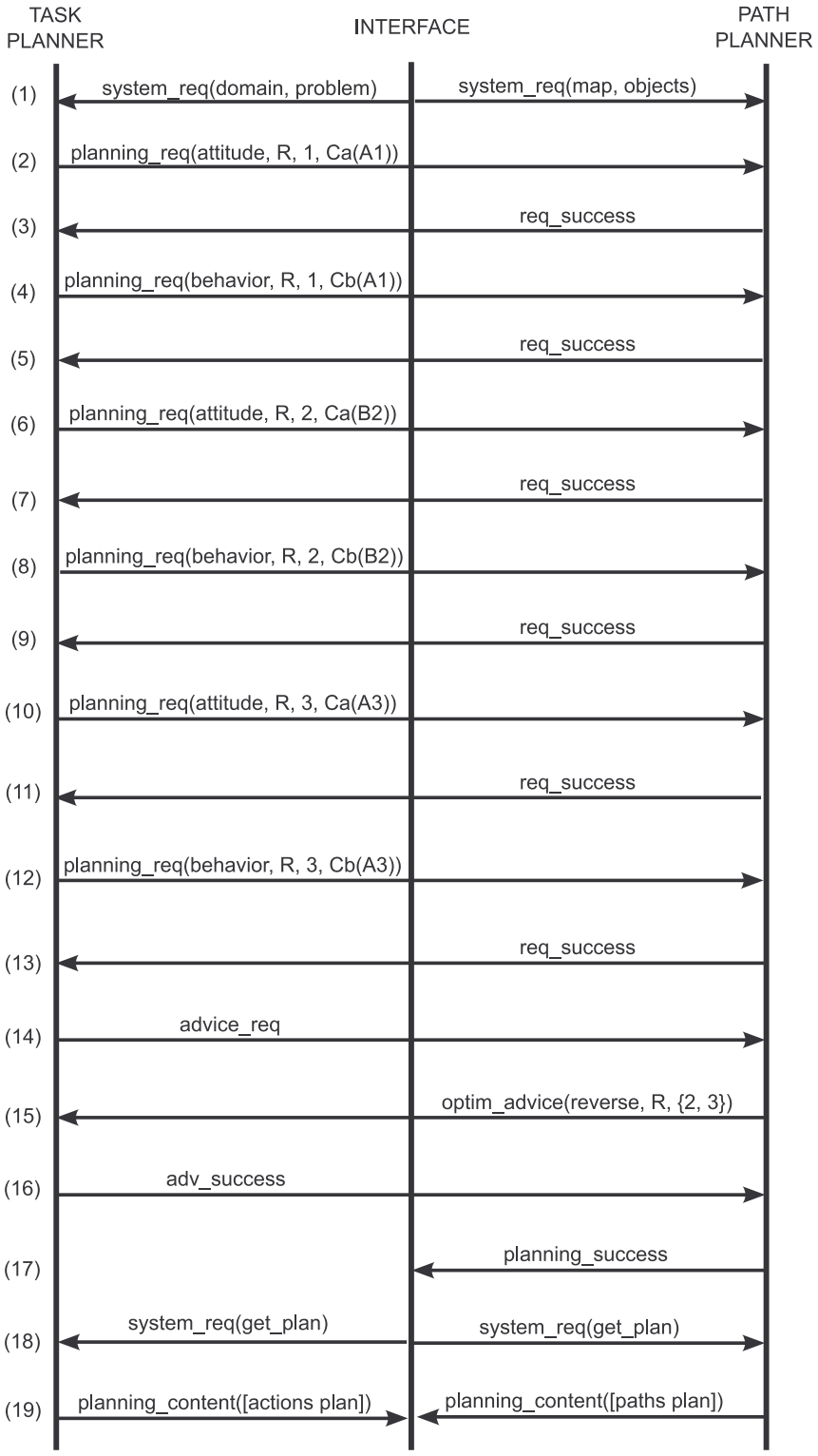


FIGURE 3.8 – Diagramme de séquence simplifié des messages échangés durant la construction du plan.

du robot sont possibles. Le planificateur de tâches envoie une seconde requête qui a pour but de vérifier les préconditions de comportement (ligne 4).

De la même façon, les deux planificateurs échangent des messages pour planifier l'action `film_objective` au point B (lignes 6 à 9) et l'action `sample_rock` au point A (lignes 10 à 13).

À ce niveau de la construction du plan, le planificateur de tâches n'a plus d'action à planifier, *i.e.,* tous les objectifs ont été atteints : le planificateur a réussi à décomposer les tâches de haut niveau qui lui ont été spécifiées en un ensemble de tâches primitives réalisables par le robot.

Comme le planificateur spécialisé a calculé les chemins du robot pour l'ensemble de la mission, il a une vision globale des déplacements de l'agent. Il peut maintenant chercher à optimiser la solution. Ainsi il propose, sur demande du planificateur de tâches (ligne 14) un conseil d'optimisation (ligne 15). La proposition est d'inverser l'ordre de réalisation des actions 2 et 3. Comme la séquence des tâches de haut niveau a été spécifiée en tant que séquence non ordonnée (*i.e.,* les actions peuvent être planifiées dans n'importe quel ordre) et que cette inversion ne remet pas en cause les différentes contraintes entre actions, le planificateur de tâches répond positivement à ce conseil (ligne 16). Modifier l'ordre d'exécution des actions n'impacte pas le reste du plan.

Le plan final peut alors être construit et envoyé au gestionnaire de mission (lignes 18, 19). Cette construction est effectuée en entrelaçant le plan des actions fourni par le planificateur de tâches avec le plan des déplacements du planificateur spécialisé. La correspondance entre les actions symboliques et les déplacements se fait grâce aux identifiants des actions présents dans les messages échangés.

3.9 Conclusion

Nous avons proposé une architecture de planification hybride dans laquelle les raisonnements symbolique et géométrique sont fortement couplés par l'expression de préconditions et d'effets géométriques dans la définition des opérateurs de planification et par l'échange de messages entre le module de raisonnement symbolique et le module de raisonnement géométrique.

Les préconditions géométriques définies au niveau des opérateurs de planification permettent l'expression de contraintes sur l'attitude que doit adopter le robot pour réaliser une action, ainsi que sur son comportement durant la réalisation de l'action. Ces contraintes géométriques sont transmises au module de raisonnement géométrique par l'utilisation de requêtes de planification. Puis elles sont traitées par un sous-module de satisfaction de contraintes du raisonneur géométrique par des techniques d'optimisation non linéaire.

Les effets géométriques permettent d'une part, de transmettre des références sur des configurations calculées du robot et, d'autre part, de partager des variables sur les ressources des agents entre le raisonnement symbolique et le raisonnement géométrique.

Par ailleurs, le planificateur de tâches peut faire appel au module de raisonnement géométrique pour guider la construction du plan en demandant des calculs heuristiques. Les deux planificateurs peuvent également entamer une phase d'optimisation du plan ainsi qu'une phase de réparation en cas de mauvaise gestion des ressources partagées. Ces échanges prennent la forme de demandes de conseil et de conseils.

La proposition d'architecture présentée ici reste à un niveau abstrait. En effet, une certaine flexibilité existe encore au niveau du choix des deux planificateurs. Pour pouvoir démontrer la faisabilité de cette architecture, il est nécessaire de choisir un planificateur de tâches et un planificateur de déplacements et de les adapter au cadre proposé. Ceci est l'objet du chapitre suivant.

Chapitre 4

Mise en œuvre de l'architecture

Dans ce chapitre, nous présentons un exemple de mise en œuvre d'une architecture de planification hybride qui s'appuie sur les fondements théoriques présentés au chapitre précédent.

Dans un premier temps, nous adaptons le planificateur hiérarchique utilisé lors des expérimentations préliminaires (chapitre 2) afin de permettre l'expression et la prise en compte des préconditions et effets géométriques. Puis, nous présentons le module de raisonnement géométrique développé pour répondre aux requêtes et demandes de conseils du planificateur de tâches. Le planificateur de déplacement au cœur du module de raisonnement géométrique est le planificateur Cell-RRT [Guitton, 2009; Guitton *et al.*, 2009]. Pour rechercher un chemin solution entre une attitude initiale et une attitude finale du robot, Cell-RRT s'appuie sur un algorithme probabiliste incrémental : l'algorithme RRT, optimisé par une phase de réduction de l'espace de recherche. Finalement, nous présentons un aperçu de l'architecture développée et des connexions entre modules de planification.

4.1 Adaptation du planificateur hiérarchique

Dans cet exemple de mise en œuvre d'une architecture de planification hybride, nous avons utilisé le planificateur hiérarchique présenté au chapitre 2. Dans un premier temps, nous présentons son adaptation permettant la prise en compte des contraintes et effets géométriques. Puis, nous détaillons la manière dont les demandes de calculs heuristiques sont formulées et envoyées au module de raisonnement géométrique.

4.1.1 Préconditions et effets géométriques

Les préconditions d'attitude et de comportement sont prises en compte lors de la sélection d'un opérateur à appliquer pour réaliser la tâche courante lorsque celle-ci est une tâche primitive. L'algorithme 4.1 présente les modifications apportées à l'algorithme 2.2 pour la gestion de ces préconditions.

Lors de la sélection d'un opérateur à appliquer, après avoir testé les préconditions symboliques (ligne 5), l'algorithme envoie une requête de planification au module de raisonnement géométrique (ligne 8). Cette requête contient les préconditions d'attitude définies au niveau de l'opérateur et unifiées avec les substitutions résultantes de l'unification des préconditions symboliques. Si les préconditions d'attitude sont satisfaites, *i.e.,* il existe un chemin permettant d'initier l'action courante, alors le planificateur de tâches transmet une seconde requête contenant les préconditions de comportement (ligne 10). Si celles-ci sont satisfaites, alors l'opérateur peut être inséré dans le plan.

Les effets géométriques sont récupérés par l'envoi d'un troisième message fournissant les prédicats associés (ligne 12). Après analyse, le module de raisonnement géométrique remplace les variables libres de ces prédicats par les valeurs correspondantes issues des calculs géométriques. À la réception de la réponse, le planificateur de tâches ajoute les couples (variable, valeur) à l'ensemble de substitutions σ (ligne 13) puis applique les effets symboliques sur l'état courant (ligne 14).

Dans le cas où l'un des deux ensembles de contraintes géométriques ne peut pas être satisfait l'opérateur est rejeté. Dans ce cas, le planificateur de tâches envoie des requêtes d'annulation (ligne 21 et 22) des demandes de calcul de chemins avant d'effectuer un retour arrière afin de tester une autre tâche.

4.1.2 Utilisation d'heuristiques

Les demandes de conseils heuristiques sont spécifiées par l'utilisation du mot-clé `heuristic` au niveau des préconditions des méthodes.

La prise en compte des résultats du calcul heuristique est faite au niveau de l'algorithme FindSubstitutions (algorithme 4.2).

Durant l'unification des prédicats contenus dans les préconditions de la méthode, si le mot clé `heuristic` est rencontré, alors une demande de conseil est envoyée au module de raisonnement géométrique. La valeur résultante est ensuite intégrée à l'ensemble σ des substitutions courantes.

Algorithme 4.1 Modification de Develop(n, $\mathcal{O}$, $\mathcal{M}$)

```
1: ...
2: for each (opérateur o ∈ O) do
3:    θ ← Unify(name(t_0), name(o), ∅) ;
4:    if (θ ≠ null) then
5:       Σ ← FindSubstitutions(symbPreconds(o), s, θ) ;
6:       if (Σ ≠ null) then
7:          for each (substitution σ ∈ Σ) do
8:             att_req ← Request(attitude, attitudeConds(o, σ)) ;
9:             if (result(att_req) = true) then
10:               behav_req ← Request(behavior, behavConds(o, σ)) ;
11:               if (result(behav_req) = true) then
12:                  resultEffects ← Request(effects, geomEffects(o, σ)) ;
13:                  σ ← addSubstitutions(σ, resultEffects) ;
14:                  s' ← appliquer symbEffects(o, σ) à s ;
15:                  n' ← (s', < t_1, ..., t_n >) ;
16:                  Develop(n', O, M) ;
17:                  if (n' ≠ null) then
18:                     ajouter n' en tant que fils à n ;
19:                     return n ;
20:                  else
21:                     Req_Cancel(att_req) ;
22:                     Req_Cancel(behav_req) ;
23:                  end if
24:               else
25:                  Req_Cancel(att_req) ;
26:               end if
27:            end if
28:         end for
29:      end if
30:   end if
31: end for
32: ...
```

Algorithme 4.2 Modification de FindSubstitutions(*preconds*, s, σ)

```
1: result ← ensemble vide de substitutions ;
2: if (preconds = ∅) then
3:    return σ ;
4: end if
5: p ← le premier prédicat de preconds ;
6: R ← le reste des prédicats de preconds ;
7: if (p begin with heuristic) then
8:    heurResult ← RequestAdvice(heuristic, p) ;
9:    σ ← Unify(p, heurResult, σ)
10:   Σ ← FindSubstitutions(R, s, σ) ;
11:   for each (substitution γ ∈ Σ) do
12:      ajouter γ à result ;
13:   end for
14: else
15:   for each (predicat p' ∈ s) do
16:      θ ← Unify(p, p', σ) ;
17:      if (θ = null) then
18:         return null ;
19:      else
20:         Σ ← FindSubstitutions(R, s, σ + θ) ;
21:         for each (substitution γ ∈ Σ) do
22:            ajouter γ à result ;
23:         end for
24:      end if
25:   end for
26: end if
27: return result ;
```

4.2 Le module de raisonnement géométrique

Le planificateur de déplacements CELL-RRT est le sous-module principal du module de raisonnement géométrique. Il permet de répondre aux requêtes de planification envoyées par le planificateur de tâches pour les préconditions d'attitude. Il est aidé par le sous-module de satisfaction de contraintes qui est en charge de transformer les préconditions d'attitude en une configuration de destination. Pour la gestion des préconditions de comportement, nous avons mis en œuvre un second planificateur de déplacements. Les demandes de conseils heuristiques émises par le planificateur de tâches sont gérées par le sous-module de calcul heuristique.

4.2.1 Justifications et présentation de Cell-RRT

L'algorithme que nous proposons pour Cell-RRT se compose de deux traitements principaux. Tout d'abord l'environnement est découpé en cellules. À partir de cette grille, un ensemble de zones de passage est recherché. Cet ensemble de zones permet de limiter la recherche d'un corridor connectant l'origine à la destination à certaines parties de l'environnement. Le résultat de ce traitement est un ensemble de cellules pour lesquelles sont définis deux points de passage : un point d'entrée et un point de sortie. Puis, un algorithme de type RRT est appliqué sur chacune de ces cellules fournissant ainsi un ensemble d'arbres qui, connectés entre eux, définissent un chemin solution permettant de rejoindre la configuration finale à partir de la configuration initiale.

Cet algorithme permet de calculer un chemin entre la configuration initiale et la configuration finale tout en respectant les contraintes cinématiques spécifiées par le modèle du robot utilisé et par les préconditions d'attitude.

Un des reproches que l'on peut faire aux planificateurs s'appuyant sur l'algorithme RRT est leur manque d'efficacité en terme de temps de calcul lorsque la recherche d'un chemin solution est effectuée dans des environnements complexes tels que les environnements labyrinthiques. En effet, dans le cas de la version originale de la méthode RRT-Extend, l'algorithme étend un réseau de configurations possibles uniformément dans l'espace de configurations, ou en direction de l'objectif si le tirage aléatoire est biaisé, mais ne tient pas compte de la topologie de l'environnement.

Pour pallier ce manque d'efficacité, nous proposons une méthode permettant de restreindre l'espace des configurations à un **corridor** dont les deux extrémités contiennent les configurations initiales et finales. Ce corridor est constitué d'un ensemble de cellules qui sont sélectionnées à l'aide de l'algorithme de recherche du plus court chemin A*. La grille contenant l'ensemble des cellules est une représentation de la topologie de l'environnement au moyen du calcul de la **traversabilité** de chaque cellule.

Dans un premier temps, nous présentons la modélisation choisie pour représenter l'environnement et le robot mobile. Puis nous détaillons les deux principaux traitements au cœur de l'algorithme mis en place : la méthode de restriction de l'espace de configuration à un corridor, puis l'algorithme RRT permettant d'obtenir un chemin solution.

4.2.2 Modélisation de l'environnement et du robot

Dans ce paragraphe, nous présentons les choix effectués pour la modélisation de l'environnement dans lequel doit évoluer le robot mobile ainsi que la modélisation du robot utilisé.

Modélisation de l'environnement

Dans un grand nombre de travaux, les obstacles de l'environnement sont modélisés comme un ensemble de polygones. Cette modélisation présente l'avantage de restreindre la mémoire utilisée pour représenter l'environnement. Cependant, elle entraine une certaine dégradation de cette représentation par rapport à la réalité. Nous avons fait le choix de représenter l'environnement sous la forme d'une image, c'est-à-dire, sous la forme d'un tableau dans lequel chaque couple de coordonnées est un point de l'environnement. Ainsi à l'échelle 1, chaque point de l'environnement a une correspondance dans le tableau. En deux dimensions, un point appartenant à un obstacle prend la valeur 1, et 0 correspond à l'absence d'obstacle, *i.e.,* à une position libre. En trois dimensions, cette valeur représente l'altitude du terrain au point (x, y) avec la valeur 0 correspondant au niveau de la mer.

Modélisation du robot

Le robot mobile que nous considérons pour nos expérimentations est un véhicule terrestre de type voiture. Si nous négligeons la dynamique, ce type de robot peut être décrit par trois variables (x, y, θ). Nous considérons que le robot se déplace à vitesse constante $(v = cte > 0)$ et que son angle de giration ϕ est borné :

$$-\phi_{max} \leq \phi \leq \phi_{max}$$

Le modèle cinématique du robot est décrit de la manière suivante :

$$\begin{cases} \dot{x} &= v.cos(\theta) \\ \dot{y} &= v.sin(\theta) \\ \dot{\theta} &= \frac{v}{L}tan(\phi) \end{cases} \tag{4.1}$$

x et y sont les coordonnées cartésiennes de la position du robot dans l'environnement, θ le cap, ϕ l'angle de giration, v la vitesse et L la longueur entre l'essieu et le centre d'inertie du véhicule.

Calculer un déplacement du robot correspond à intégrer ce modèle sur un intervalle de temps Δt. Afin de limiter l'espace de recherche, les changements de l'angle de giration sont modélisés par une commande à trois états :

$$\phi \;=\; \{-\phi_{max}, 0, \phi_{max}\}$$

Ainsi le robot peut soit se déplacer en ligne droite, soit effectuer des virages avec un rayon de giration r_{min} fixe.

$$r_{min} \;=\; \frac{L}{tan(\phi_{max})}$$

Modélisation d'une configuration

À partir du modèle cinématique du robot, une configuration est définie par le triplet :

$$q = < x, y, \theta >$$

dans lequel x et y sont les coordonnées cartésienne et θ est le cap du robot.

Modélisation d'une transition entre deux configurations

Une transition entre deux configurations du robot est représentée par la paire $< \phi, \Delta t >$:

$$< x, y, \theta > \xrightarrow{<\phi, \Delta t>} < x', y', \theta' >$$

où ϕ est l'angle de giration associé au mouvement et Δt est la durée du mouvement.

4.2.3 La boucle principale du planificateur

L'algorithme centralisant les différentes fonctions du planificateur est présenté ci-dessous (algorithme 4.3) :

Dans un premier temps, le planificateur découpe l'environnement en un ensemble de cellules (ligne 2), calcule la traversabilité de chacune d'elles (ligne 4) et définit l'ensemble des points de passage permettant de connecter les cellules entre elles (ligne 5). Puis il cherche un corridor solution entre la configuration initiale et la configuration finale (ligne 9). À partir de ce corridor, les sous-arbres sont calculés pour chaque cellule en utilisant comme configuration initiale et finale les points de passages entre les cellules adjacentes du corridor (ligne 15). Si une cellule du corridor ne contient pas de solution alors une pénalité lui est attribuée (ligne 17) afin d'empêcher l'algorithme A* de la rechoisir lors de la recherche d'un nouveau corridor. La boucle est itérée jusqu'à obtenir un chemin solution ou un échec. Il y a échec lorsqu'il n'est plus possible de trouver de corridor et donc de chemin solution. En cas de succès, le chemin solution est construit en assemblant les plus courts chemins extraits des arbres de chaque cellule appartenant au corridor (ligne 25).

4.2.4 Découpage de l'environnement et calcul du corridor

Dans la première partie de l'algorithme mis en œuvre dans le planificateur, l'environnement est discrétisé sous la forme d'un ensemble de cellules. Cet ensemble permet de limiter la recherche d'un chemin solution à une partie de l'environnement que l'on nomme **corridor**. Dans un premier temps, un poids est associé à chaque cellule de la grille. Ce poids permet de guider la recherche du corridor en favorisant les zones contenant peu d'obstacles, *i.e.,* les zones les

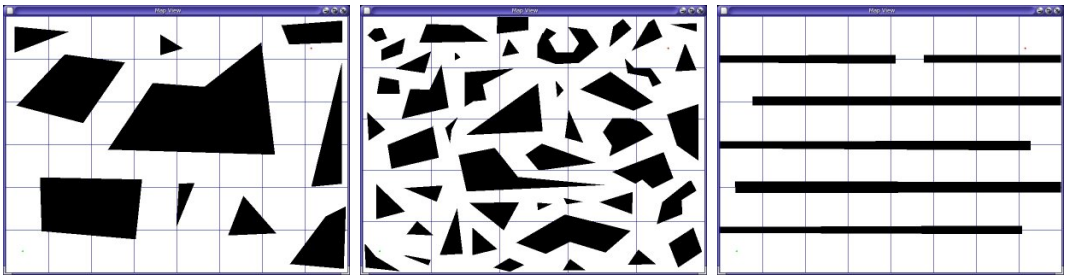
Algorithme 4.3 CellRRTMainLoop($environmentFile$, q_{start}, q_{goal})

```
1:  environment ← loadEnvironment(environmentFile)
2:  tabCells ← decomposeIntoCells(environment) ;
3:  for each (cell of tabCells) do
4:    computeCellTraversability(cell)
5:    findCellWaypoints(cell) ;
6:  end for
7:  solutionFound ← false ;
8:  while (not solutionFound) do
9:    path ← searchAStar(tabCells, q_start, q_goal) ;
10:   if (path ≠ null) then
11:     solutionFound ← true ;
12:     for each (cell of path) do
13:       q_in ← getStartConfiguration(cell) ;
14:       q_out ← getGoalConfiguration(cell)
15:       rrt ← computeRRTExtExt(cell, q_in, q_out) ;
16:       if (rrt = null) then
17:         addPenaltyToCell(cell) ;
18:         solutionFound ← false ;
19:       else
20:         addPartialSolution(environment, rrt) ;
21:       end if
22:     end for
23:   end if
24: end while
25: return extractPathSolution(environment) ;
```

plus traversables. Dans un deuxième temps, des points d'entrée et de sortie sont calculés pour chaque cellule. Enfin, le corridor est calculé à l'aide de l'algorithme A*.

Découpage et calcul de la traversabilité

La figure 4.1 représente le découpage de trois environnements qui serviront d'illustrations dans la suite de ce chapitre. Les environnements 1 et 3 sont découpés selon un quadrillage de 8 par 6 cellules. L'environnement 2 est découpé selon un quadrillage de 5 par 5 cellules. Chaque cellule est stockée dans un tableau et sera considérée par la suite comme un environnement à part entière, c'est-à-dire qu'il lui est associé une configuration initiale et une configuration finale. Si cette cellule appartient au corridor solution, alors elle doit contenir un chemin entre ces deux configurations.



(a) environnement 1 (b) environnement 2 (c) environnement 3

FIGURE 4.1 – Exemple de découpages en cellules de trois environnements

Pour chaque cellule c_i de l'environnement, un poids correspondant à la traversabilité de la cellule $t(c_i)$ est calculé. Ce poids est le ratio d'occupation des obstacles par rapport aux zones libres. Pour un environnement en deux dimensions, il suffit de calculer le rapport du nombre de point valant 1 sur le nombre total de points de la cellule. En environnement à trois dimensions, la cellule est caractérisée par sa trace au sol. On somme alors pour chaque point :

- 0 si $z(x, y) <= z_{min}$;
- ...
- 1 si $z(x, y) > z_{max}$.

z_{min} est l'altitude la plus basse de l'environnement et z_{max} est liée aux capacités du robot. Puis on divise par le nombre de points de la cellule.

Un poids de 0 correspond à une cellule sans obstacle et donc parfaitement traversable. Par contre, suivant la direction de traversée choisie, la configuration des obstacles peut faire que, même avec un faible ratio d'occupation, la traversée ne soit pas possible Par exemple, la majorité des cellules de l'environnement 3 sont dans cette situation. L'algorithme considère donc un seuil de traversabilité t_{MAX} à partir duquel on interdit à la cellule d'appartenir au corridor solution :

$$t(c_i) > t_{MAX} \Rightarrow t(c_i) = 1$$

Définition des points de passage

Pour qu'une cellule de l'environnement appartienne au corridor solution, elle doit pouvoir contenir un chemin solution. Or, à ce stade, il n'est pas possible de calculer le chemin car le but de cette partie de l'algorithme est de restreindre efficacement et rapidement l'espace de recherche. Pour que le chemin solution puisse traverser une cellule, celle-ci doit contenir un point de passage en commun avec les cellules adjacentes sur chacune de ses frontières traversées. Ces points de passage doivent être le plus éloigné possible des obstacles que ce soit longitudinalement ou latéralement. Il faut donc, pour les choisir, tenir compte de l'espace à la fois le long de la frontière et perpendiculairement à la frontière.

En deux dimensions, les points de passage sont calculés de la manière suivante : pour chacune des frontières de la cellule considérée, les segments libres dont la longueur est supérieure à un seuil dépendant des caractéristiques du robot sont identifiés. Ces segments sont ensuite divisés récursivement en sous-segments jusqu'à atteindre une taille minimale égale au seuil. Pour chacun des segments seg_i, on calcule un critère C_{seg_i} correspondant à la surface d'un carré dont la longueur des côtés est définie par le minimum entre la longueur du segment et la longueur du segment libre de la médiatrice correspondante, c'est-à-dire ne contenant pas d'obstacles au niveau de la médiatrice dans les deux cellules adjacentes :

$$C_{seg_i} = min(sL(seg_i), mL(seg_i))^2$$

Où sL et mL sont respectivement la longueur du segment et de sa médiatrice.

Le point de passage associé à la frontière considérée est le centre du segment Seg ayant une valeur C_{seg} maximale :

$$Seg = max_i(C_{seg_i})$$

Cette méthode, représentée sur la figure 4.2, permet de tenir compte de la longueur du passage entre deux cellules en plus de sa largeur et donc d'éviter les passages étroits. En effet, plus la surface correspondante est grande, plus le planificateur pourra trouver des configurations permettant de connecter deux cellules adjacentes en respectant les contraintes cinématiques du robot. Le point de passage correspondant à la plus grande surface est représenté par une croix rouge.

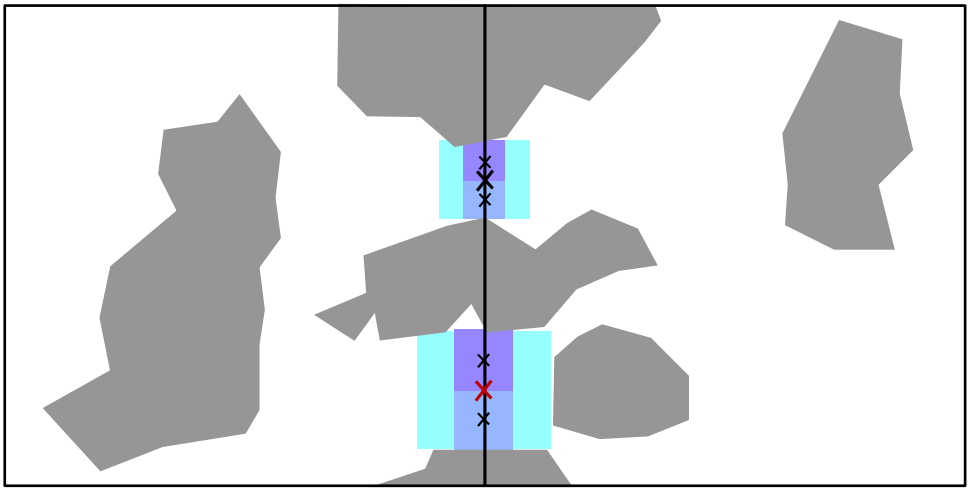


FIGURE 4.2 – Méthode de calcul des points de passage à partir des surfaces

La figure 4.3 illustre les points de passage possibles calculés pour les cellules de l'environnement 1.

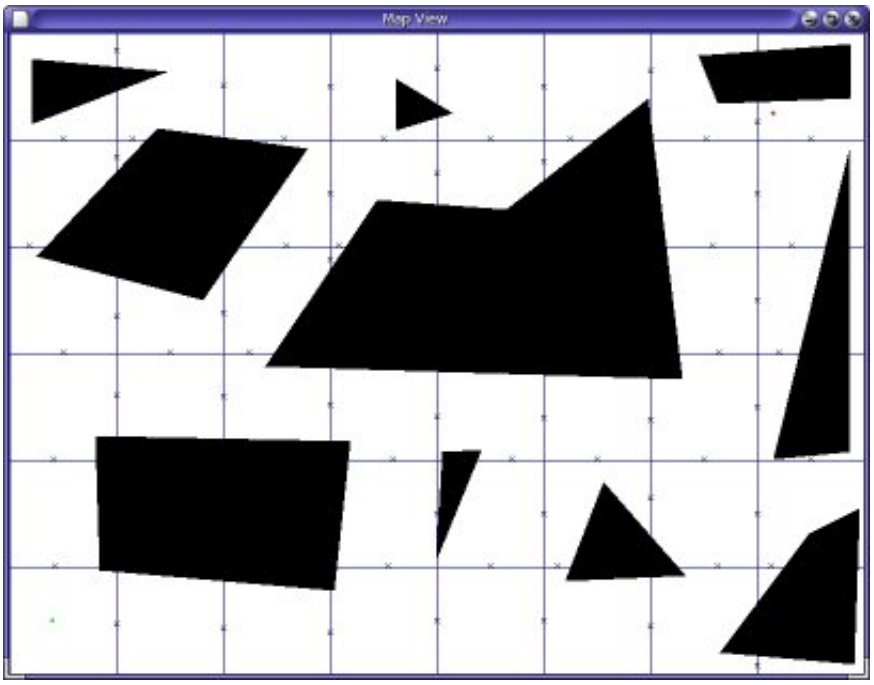


FIGURE 4.3 – Exemples de points de passage pour l'environnement 1

Calcul du corridor solution

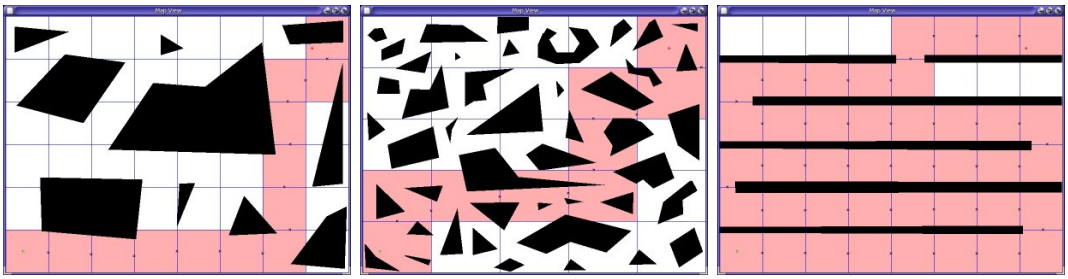
Une fois l'ensemble des points de passage calculé, un corridor solution entre la cellule contenant la configuration initiale du robot et celle contenant la configuration finale est calculé avec l'algorithme A*. Une heuristique possible pour le choix de la prochaine cellule c_i du chemin est la distance euclidienne entre le centre de la cellule et le centre de la cellule destination c_{goal} et comme fonction de coût de transition la distance $d(.,.)$ entre les deux cellules, pondérée par le coefficient de traversabilité $t(.)$ de la cellule :

$$\begin{cases} g(c_i) = g(c_{i-1}) + (d(c_{i-1}, c_i) \times (1 + \gamma \times t(c_i))) \\ h(c_i) = d(c_i, c_{goal}) \end{cases}$$

Où γ est un paramètre positif permettant d'ajuster l'influence de la distance par rapport à la traversabilité dans le calcul du corridor.

Si la cellule c_i ne contient pas de point d'entrée sur sa frontière avec la cellule précédente c_{i-1} alors elle est abandonnée et la cellule suivante de moindre coût est sélectionnée. Le corridor solution est obtenu lorsque l'algorithme atteint la cellule destination, *i.e.,* la cellule contenant la configuration finale et que toutes les cellules pendantes ont été traitées.

Les corridors formés par les cellules en rose (figure 4.4) sont les solutions de l'algorithme A* pour les trois environnements tests.



(a) environnement 1 (b) environnement 2 (c) environnement 3

FIGURE 4.4 – Exemple de corridors solutions pour les trois environnement

4.2.5 Détails de l'algorithme RRT

Une fois que l'espace de recherche a été restreint à un corridor, un chemin respectant les contraintes cinématiques du robot est calculé. L'algorithme utilisé est l'algorithme RRT dans sa version RRT-ExtExt (*cf.* paragraphe 1.3.3). La principale difficulté est de trouver une méthode efficace pour connecter les deux arbres. Nous avons utilisé la méthode Cercle-Segment-Cercle (CSC), inspirée des travaux de [Dubins, 1957], qui permet de connecter tout couple de configurations espacées d'au moins deux fois le rayon de giration minimum.

Comme nous l'avons vu dans le paragraphe 1.3.3, l'algorithme RRT est divisé en trois fonctions principales : le tirage d'une configuration aléatoire q_{target}, le choix de la configuration $q_{nearest}$ la plus proche appartenant à l'arbre et l'extension de $q_{nearest}$ vers q_{target} qui permet d'obtenir une nouvelle configuration q_{new}. Nous décrivons dans la suite le fonctionnement de ces trois fonctions ainsi que la fonction de génération d'une nouvelle configuration.

Tirage de la configuration aléatoire

Afin de diriger le développement de l'arbre en direction de la configuration finale, la distribution aléatoire des configurations peut être non uniforme. Selon une probabilité *probabilityGoal*, l'algorithme va tirer des configurations qui se trouvent à moins de *radius* unités de la configuration finale. Cette probabilité est appelée biais de tirage. Sinon, dans le cas de probabilité $(1 - probabilityGoal)$, le tirage est uniforme. La méthode est décrite par l'algorithme 4.4.

Choix du nœud le plus proche

L'objectif de l'algorithme 4.5 est de choisir un nœud dont la configuration associée sera ensuite étendue vers la configuration tirée aléatoirement. Pour choisir ce nœud, nous utilisons comme heuristique la distance euclidienne entre deux configurations. Ainsi le nœud de l'arbre le plus proche de q_{target} est sélec-

Algorithme 4.4 chooseTarget()

```
1: p ← random([0,1]) ;
2: if (p < probabilityGoal) then
3:    q_target ← randomNodeAroundGoal(q_goal, radius) ;
4: else
5:    q_target ← randomNode() ;
6: end if
7: return q_target ;
```

tionné. Cependant, il doit être à une distance supérieure à $minDistance$ de la position cartésienne de la configuration q_{target} afin de permettre une extension respectant les contraintes cinématiques du robot. Cette distance minimale est fixée à deux fois le rayon de giration minimal r_{min} du robot.

Algorithme 4.5 nearestNeighbor(q_{target})

```
1: bestDistance ← +∞ ;
2: for each (q ∈ T) do
3:    distance ← euclidianDistance(q,q_target)
4:    if ((distance < bestDistance ) and ((distance > minDistance)) then
5:       q_nearest ← q ;
6:       bestDistance ← distance ;
7:    end if
8: end for
9: return q_nearest ;
```

Ajout d'une nouvelle configuration

Cet ajout est effectué en étendant la configuration $q_{nearest}$ vers la configuration q_{target}. l'algorithme 4.6 présente cette extension. La première étape consiste à calculer ϕ, c'est-à-dire à calculer si le robot doit aller en ligne droite pour rejoindre la configuration cible, ou bien s'il doit tourner à droite ou à gauche. Pour choisir la commande à appliquer on vérifie que la cible est dans un cone de visibilité restreint $minCone$ du robot. Si c'est le cas alors le robot doit avancer en ligne droite, sinon il doit se rapprocher de cette direction. Puis la nouvelle configuration est générée. Elle correspond à la configuration dans laquelle le robot se trouvera au bout d'un intervalle de temps Δt. Cet intervalle est le pas d'incrémentation de l'algorithme. Enfin, la fonction vérifie que le déplacement de la configuration $q_{nearest}$ à cette nouvelle configuration est possible.

Algorithme 4.6 extend($q_{nearest}$, q_{target}, Δt)

1: $\alpha_1 \leftarrow$ `computeDirection`($q_{nearest}$, q_{target}) ;
2: $\alpha \leftarrow$ (-$\theta(q_{nearest})$ + α_1)% $2 \times \pi$;
3: **if** ($|\alpha| > minCone$) **then**
4: $\quad \phi \leftarrow \phi_{max} \times$ signum(α) ;
5: **else**
6: $\quad \phi \leftarrow 0$;
7: **end if**
8: $q \leftarrow$ `generateConfiguration`($q_{nearest}$, ϕ, Δt) ;
9: **if** (notInObstacle(q) and isConnectable($q_{nearest}$, q)) **then**
10: $\quad q_{new} \leftarrow q$;
11: **else**
12: $\quad q_{new} \leftarrow$ null ;
13: **end if**
14: **return** q_{new} ;

Génération d'une configuration

Cette génération (algorithme 4.7) correspond à l'intégration du modèle citématique du robot décrit à l'équation 4.1 sur l'intervalle de temps Δt.

Les équations correspondantes aux modèles cinématiques (lignes 9 à 11) ont été simplifiées pour le cas où l'angle de giration ϕ est égal à 0 (lignes 5 à 7), c'est-à-dire dans le cas où le robot se déplace en ligne droite.

Algorithme 4.7 generateConfiguration($q_{nearest}$, ϕ, Δt)

1: $x_0 \leftarrow q_{nearest}.x$;
2: $y_0 = q_{nearest}.y$;
3: $\theta_0 = q_{nearest}.\theta$;
4: **if** ($\phi = 0$) **then**
5: $\quad x_1 \leftarrow x_0 + \Delta t \times v \times \cos(\theta_0)$;
6: $\quad y_1 \leftarrow y_0 + \Delta t \times v \times \sin(\theta_0)$;
7: $\quad \theta_1 \leftarrow \theta_0$;
8: **else**
9: $\quad x_1 \leftarrow x_0 +$ (sin($\Delta t \times v \times \tan(\phi) + \theta_0$) - sin($\theta_0$)) / tan($\phi$) ;
10: $\quad y_1 \leftarrow y_0 +$ (-cos($\Delta t \times v \times \tan(\phi) + \theta_0$) + cos($\theta_0$)) / tan($\phi$) ;
11: $\quad \theta_1 \leftarrow \theta_0 + \Delta t \times v \times \tan(\phi)$;
12: **end if**
13: $q \leftarrow$ new configuration(x_1, y_1, θ_1, ϕ, Δt) ;
14: **return** q ;

Connection des arbres : la méthode CSC

L'algorithme RRT-ExtExt développé génère simultanément deux arbres et se termine lorsque les configurations associées à deux nœuds feuilles de ces arbres peuvent être connectées ensemble. Afin de connecter ces deux arbres, nous utilisons la méthode CSC.

Le robot modélisé correspond à un robot de type *Dubins Car*. En effet, sa vitesse est positive et fixe, et le rayon de giration est fixé au rayon minimum. La trajectoire du robot peut alors être modélisée par des courbes de Dubins. Il a été montré dans [Dubins, 1957] que le chemin le plus court entre deux configurations peut toujours être exprimé par une combinaison d'au maximum trois mouvements. Ces mouvements peuvent être "aller tout droit" (S pour *segment*), "tourner à droite" (R pour *right*) ou "tourner à gauche" (L pour *left*). La séquence des mouvements est appelée **mot**. Il n'existe que dix mots de longueur trois et seulement six d'entre eux peuvent modéliser un chemin optimal entre deux configurations : {LRL, RLR, LSL, LSR, RSL, RSR}.

La méthode CSC correspond à l'utilisation uniquement des mots {LSL, LSR, RSL, RSR} pour connecter deux configurations, *i.e.,* le robot doit effectuer un premier virage, un parcours en ligne droite, puis un deuxième virage pour rejoindre la configuration de destination.

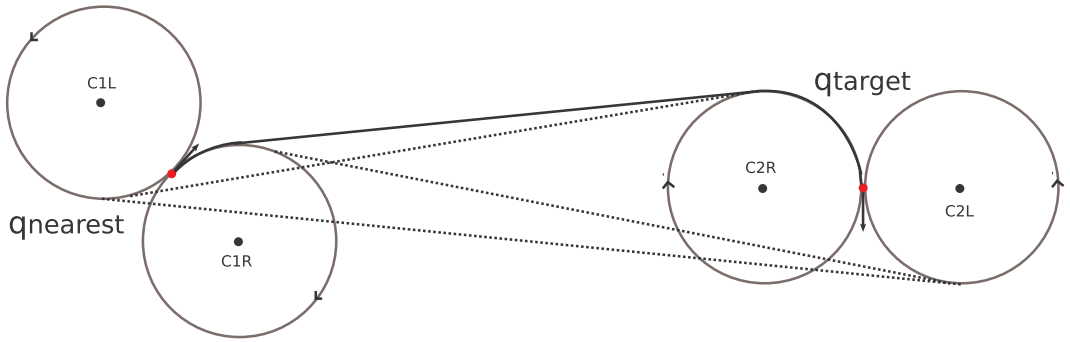


FIGURE 4.5 – Connexion entre deux cercles : 4 tangentes possibles

Le déroulement est le suivant (algorithme 4.8) : les deux cercles de rayon égal au rayon minimum de giration sont calculés pour chacune des deux configurations à connecter. Puis les bitangentes permettant de relier deux à deux les cercles sont calculées. Il existe quatre tangentes possibles (figure 4.5) car les cercles sont orientés suivant les caps des configurations. Ensuite la longueur des quatre trajectoires {LSL, LSR, RSL, RSR} est calculée. Enfin les configurations intermédiaires permettant de relier les configurations $q_{nearest}$ et q_{target} sont générées à partir de la séquence de mouvements de longueur minimale.

La figure 4.6 illustre la connexion des deux arbres lors de la recherche d'un chemin solution pour l'environnement 1 sans utiliser de découpage.

Algorithme 4.8 computeCSC($q_{nearest}$, q_{target})

1: $C_{1R} \leftarrow$ `computeCircleCenter`($q_{nearest}$, 'right') ;
2: $C_{1L} \leftarrow$ `computeCircleCenter`($q_{nearest}$, 'left') ;
3: $C_{2R} \leftarrow$ `computeCircleCenter`(q_{target}, 'right') ;
4: $C_{2L} \leftarrow$ `computeCircleCenter`(q_{target}, 'left') ;
5: $T_{RR} \leftarrow$ `computeTangent`($q_{nearest}$, q_{target}, C_{1R}, C_{2R}) ;
6: $T_{LL} \leftarrow$ `computeTangent`($q_{nearest}$, q_{target}, C_{1L},C_{2L}) ;
7: $T_{RL} \leftarrow$ `computeTangent`($q_{nearest}$, q_{target}, C_{1R},C_{2L}) ;
8: $T_{LR} \leftarrow$ `computeTangent`($q_{nearest}$, q_{target}, C_{1L},C_{2R}) ;
9: $T \leftarrow$ `bestPathTangent`(T_{RR}, T_{LL}, T_{RL}, T_{LR})
10: $\{q_1,q_2\} \leftarrow$ `computeConfigurations`($q_{nearest}$, q_{target}, T) ;
11: **return** $\{q_1,q_2\}$;

FIGURE 4.6 – Exemple d'arbres connectés pour l'environnement 1

La figure 4.7 représente un chemin solution pour l'environnement 2 avec un découpage en 5×5 cellules.

Réutilisation des arbres et chemins précédemment calculés

Lorsque le planificateur ne trouve pas de solution avec l'algorithme RRT mis en œuvre, alors un nouveau corridor doit être calculé. Ce nouveau corridor ne contient pas la cellule précédemment incriminée, cependant il peut être constitué de cellules appartenant à l'ancien corridor. Dans ce cas, les calculs précédents peuvent être réutilisés.

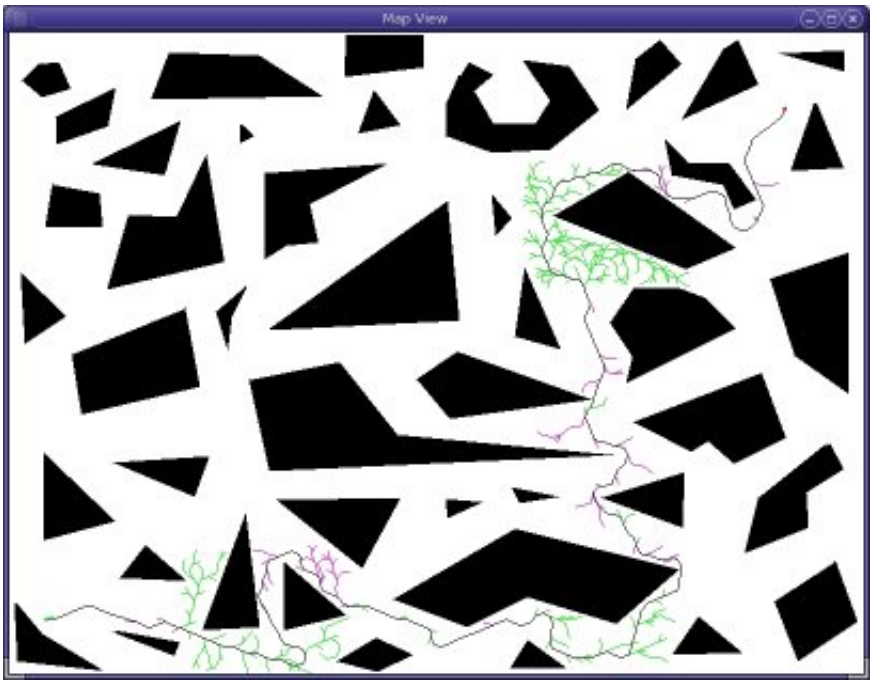


FIGURE 4.7 – Exemple de chemin solution pour l'environnement 2

Réutilisation de segments de chemins. Le segment de chemin calculé pour une cellule i peut être réutilisé si et seulement si les cellules $i-1$ et $i+1$ du nouveau corridor appartiennent à l'ancien corridor. En effet, les points de passage entre les cellules sont les mêmes que pour l'ancien corridor, le segment de chemin est donc valide. Concernant la cellule initiale et la cellule finale, il suffit que la cellule $i+1$, respectivement $i-1$, soit la même.

Réutilisation d'arbres. Lorsque la cellule précédente ou suivante n'est pas la même, le segment de chemin précédemment calculé ne peut pas être réutilisé. Cependant, si l'un des points de passage est le même alors l'arbre précédemment développé ayant pour racine ce point de passage peut être réutilisé. Lors de la recherche d'un chemin dans cette cellule, l'algorithme continue de développer cet arbre simultanément au développement du second arbre ayant pour racine le nouveau point de passage.

4.2.6 Sous-module de satisfaction des préconditions d'attitude

Le planificateur de déplacements CELL-RRT doit trouver un chemin permettant au robot de se déplacer dans l'environnement entre deux configurations afin de réaliser une action suite à la réception d'une requête de planification envoyée par le planificateur de tâches. Cette requête contient un ensemble de

contraintes géométriques ne spécifiant pas directement une configuration de destination mais un ensemble dans lequel doit se trouver cette configuration. Afin de permettre au planificateur de déplacements de calculer un chemin, ce sous-module de satisfaction de contraintes doit préalablement calculer une configuration satisfaisant les contraintes géométriques.

Mise en œuvre dans le module de raisonnement géométrique

La résolution formelle décrite au paragraphe 3.5 du chapitre précédent permet, à partir des données géométriques du vecteur d'état courant du robot ou d'un vecteur choisi aléatoirement, de calculer un état destination pour le planificateur de déplacements. Le principe de cette mise en œuvre est illustré par l'algorithme 4.9.

Algorithme 4.9 Satisfaction des préconditions d'attitude

```
1:  r_0 ← S ;
2:  nbTries ← 0 ;
3:  noPathFound ← true ;
4:  while (noPathFound and nbTries < maxTries) do
5:    k ← 0 ;
6:    while (F(r_k) > 0 and k < maxK) do
7:      compute ∇F(r_k) ;
8:      compute r_{k+1} ;
9:      k ← k + 1 ;
10:   end while
11:   if (notInObstacle(r_k)) then
12:     noPathFound ← CellRRTMainLoop(environmentFile, S, r_k) ;
13:     if (noPathFound) then
14:       r_0 ← Srand() ;
15:     end if
16:   else
17:     r_0 ← Srand() ;
18:   end if
19:   nbTries ← nbTries + 1 ;
20: end while
```

Cet algorithme cherche à calculer un nouvel état qui :
- satisfait les contraintes géométriques (ligne 6) ;
- n'est pas dans un obstacle (ligne 11) ;
- est atteignable par le planificateur de déplacements (ligne 12).

L'algorithme démarre en initialisant la descente de gradient avec l'état courant S du robot (ligne 1). Si cette initialisation aboutit à un échec, alors des initialisations aléatoires successives sont testées (lignes 14 et 17).

Durant la descente de gradient, l'algorithme calcule le gradient en utilisant les équations 3.6, 3.7 et 3.8 (ligne 7) et calcule un nouveau vecteur d'état r_k en utilisant l'équation 3.5 (ligne 8).

L'optimisation du pas λ_k est effectuée en fixant initialement la valeur à 1 et en la divisant par 2 tant que $F(r_{k+1})$ est supérieure à $F(r_k)$.

Pour garantir la terminaison de l'algorithme, le nombre d'itérations des boucles est borné.

Si $F(r_k) = 0$, notInObstacle(r_k) = false et $noPathFound$ = false, l'algorithme retourne une réponse positive au planificateur de tâches. Dans tous les autres cas, il lui indique que l'action n'est pas envisageable.

4.2.7 Gestion des préconditions de comportement

Contrairement aux préconditions d'attitude, les préconditions de comportement ne s'appliquent pas qu'à la configuration finale du déplacement mais à toute la trajectoire. De plus, la spécification de ce déplacement ne fait pas référence à une configuration de destination mais est un critère d'arrêt.

Pour la définition d'un chemin respectant les préconditions de comportement, nous avons développé un algorithme de planification de déplacements qui s'appuie sur l'algorithme RRT-Extend et qui développe un unique arbre ayant pour racine la configuration initiale. L'algorithme 4.10 illustre cette mise en œuvre.

À la différence de l'algorithme d'extension de référence (algorithme 1.1), la sortie de la boucle principale ne s'effectue pas lorsque la configuration finale peut être connectée à l'arbre en développement, mais lorsque le critère d'arrêt est atteint, *i.e.,* lorsque la configuration courante satisfait le critère de terminaison (ligne 4). Ce critère est fonction de la durée du déplacement ou peut être ramené à la durée du déplacement.

Afin d'optimiser le nombre de nœuds testés, lorsqu'un nœud cible est tiré, celui-ci est projeté dans une zone de l'environnement qui satisfait les contraintes des préconditions de comportement (ligne 5). Cette projection est effectuée en utilisant le module de satisfaction des contraintes d'attitude. En effet, du fait de la transformation des contraintes constantes en contraintes d'égalité, la forme des contraintes de comportement est la même que celle des contraintes d'attitude. Puis un nouveau nœud q_{new} est créé (ligne 7). Le traitement associé à cette création dépend du type des contraintes : égalité ou inégalité.

Pour les contraintes d'inégalité. L'algorithme fait classiquement appel à la méthode extend (algorithme 4.6) (ligne 7). Dans l'hypothèse où ce nœud ne

Algorithme 4.10 RRTExtendForBehavior(q_{start}, $\mathcal{C}$, c_{stop})

```
1:  T.init(q_start) ;
2:  i ← 0 ;
3:  q_new ← q_start ;
4:  while (i < K and canBeGoal(q_new, c_stop) = false) do
5:     q_target ← project(chooseTarget(), C) ;
6:     q_nearest ← T.nearestNeighbor(q_target) ;
7:     q_new ← extend(q_nearest, q_target) ;
8:     if (C contains constant constraints) then
9:        q_new ← computeConstant(q_new, q_nearest, C) ;
10:    end if
11:    if (q_new ≠ null and respectConstraints(q_new)) then
12:       T.add(q_new) ;
13:    end if
14:    i = i + 1 ;
15: end while
16: if (canBeGoal(q_new, c_stop)) then
17:    T.setGoal(q_new) ;
18: end if
19: return T ;
```

respecterait pas les contraintes de comportement, *i.e.,* le nœud sortirait de la zone définie par les contraintes, alors il est rejeté (ligne 11).

Pour les contraintes d'égalité. Les contraintes d'égalité correspondent à la traduction des contraintes constantes. Pour ce type de contraintes, l'objectif est de calculer une configuration intermédiaire pour une incrémentation de l'algorithme (ligne 9). Pour ce faire, nous faisons également appel à la méthode extend (ligne 7) afin, d'une part, de définir une configuration dont les paramètres peuvent être utilisés comme éléments connus pour la définition de la configuration intermédiaire et, d'autre part, de restreindre la durée du mouvement à une durée proche de celle du pas d'incrément. Nous traitons deux types de fonctions d'égalité :

- celles pour lesquelles le cap de la configuration intermédiaire est préalablement fixé (*e.g.,* fonction `rel_angle`) ;
- celles pour lesquelles la position de la configuration intermédiaire est préalablement fixée (*e.g.,* fonction `distance`).

Ainsi lors du calcul d'un mouvement correspondant à un pas de temps de l'algorithme, l'objectif est de définir les valeurs des paramètres qui respectent la contrainte, *i.e.,* l'angle de giration ϕ et l'intervalle de temps Δt, puis d'en déduire les paramètres de configuration manquants. Puis ce nouveau nœud est vérifié (ligne 11).

L'algorithme se termine lorsque aucune solution n'a été trouvée au bout de K itérations ou lorsque le critère d'arrêt a été atteint à ϵ près. Pour chaque nœud ajouté à l'arbre, nous calculons la durée du déplacement depuis le nœud origine, *i.e.,* le nœud racine de l'arbre (algorithme 4.11).

Algorithme 4.11 canBeGoal(q, c_{stop})

```
1: duration ← 0 ;
2: while (getRoot(q) != null) do
3:    duration ← computeDuration(getRoot(q), q) ;
4:    q ← getRoot(q) ;
5: end while
6: if (duration + ε ≥ c_stop) then
7:    return true ;
8: else
9:    return false ;
10: end if
```

4.2.8 Sous-module de calcul heuristique

Le sous-module de calcul heuristique est chargé de répondre aux demandes de conseil du planificateur de tâches. Il a été défini pour répondre à des demandes bien précises, par exemple, trouver le point de passage le plus proche.

Principe de calcul de l'heuristique

Une demande de conseil heuristique est formulée par l'envoi d'une requête contenant un prédicat dont le symbole est le nom de l'heuristique demandée. Les constantes présentes dans le prédicat permettent d'identifier l'agent et les objets nécessaires au calcul de la valeur de l'heuristique. Le prédicat contient également une variable non unifiée.

Cette variable est remplacée par le résultat du calcul heuristique puis le prédicat est renvoyé au planificateur de tâches. Celui-ci peut alors ajouter ce couple (variable, valeur) à l'ensemble des substitutions courantes afin d'utiliser le résultat du calcul heuristique.

Lors de son initialisation, le module de raisonnement géométrique crée un tableau contenant l'ensemble des étiquettes identifiant les objets de l'environnement. Ainsi les demandes de calculs sont effectuées rapidement. Par exemple, pour l'heuristique permettant de définir l'objet le plus proche d'une position, le sous-module calcule l'ensemble des distances entre les coordonnées des objets et le point de référence à l'aide d'un simple calcul de distance euclidienne.

Différentes heuristiques

Les différentes heuristiques mises en œuvre et accessibles au planificateur de tâches sont les suivantes :

- point de passage (objet) le plus proche ;
- distance entre un objet et un agent ou un autre objet ;

L'heuristique permettant de rechercher l'objet le plus proche se formule à l'aide du prédicat (`nearest_waypoint ?x ?value`) dans lequel la variable `?x` identifie un agent ou un objet de l'environnement. La variable résultat `value` est remplacée, après calcul, par l'étiquette symbolique identifiant l'objet le plus proche, en termes de distance euclidienne, de la position de l'agent ou de l'objet `?x`.

L'heuristique déterminant la distance entre deux objets ou entre un agent et un objet est formulée avec le prédicat (`distance_between ?x ?y ?value`) où `?x` et `?y` sont les étiquettes symboliques identifiant l'agent et les objets. Le résultat est la distance euclidienne entre ces deux éléments.

4.2.9 Calcul des ressources

Le robot dispose d'une ressource d'énergie identifiée par la propriété de concept `energy_level`.

Le niveau d'énergie est initialisé et mis à jour par le planificateur de tâches en utilisant d'une commande d'affectation directe :

```
(setProperty(?r.energy_level, ?q))
```

Lors du calcul d'un déplacement du robot, le module de raisonnement géométrique met à jour automatiquement la valeur de cette ressource. La quantité d'énergie nécessaire à un déplacement est calculée en multipliant la distance parcourue par un coefficient de consommation.

La quantité consommée lors d'un déplacement est accessible au planificateur de tâches grâce à la propriété globale `conso_energy`.

4.3 Communication entre modules

Le module de raisonnement symbolique et le module de raisonnement géométrique sont des agents logiciels s'exécutant dans des unités d'exécution différentes. Leurs exécutions sont synchronisées par l'envoi de messages correspondant aux messages d'interaction entre planificateurs. Par exemple, lorsque le

planificateur de tâches envoie une requête de planification, il met en pause son exécution en attente de la réponse du module de raisonnement géométrique.

Le programme principal fait office d'interface entre les deux modules de raisonnement et est en charge de l'initialisation du processus de planification et de la gestion des résultats à la fin de ce processus. Il assure également la cohérence des messages échangés entre les deux modules.

L'architecture de planification hybride mise en œuvre a été développée en Java (version 1.5).

4.4 Conclusion

Dans ce chapitre, nous avons proposé des extensions du planificateur de tâches HTN présenté au chapitre 2 permettant de prendre en compte l'expression de préconditions et effets géométriques au niveau de la définition des opérateurs de planification et l'utilisation de conseils heuristiques définis au niveau des préconditions des méthodes.

Puis, nous avons présenté l'algorithmique du planificateur Cell-RRT qui permet de trouver un chemin dans un environnement statique composé d'obstacles en respectant les contraintes cinématiques du robot utilisé. Le planificateur s'appuie sur la méthode RRT-ExtExt qui est une variante de l'algorithme probabiliste incrémental RRT et qui permet de connecter deux configurations de l'environnement en développant simultanément deux arbres. L'un prend pour racine la configuration initiale et l'autre la configuration finale. Afin d'optimiser le temps de recherche d'une solution, le planificateur peut découper l'environnement en un ensemble de cellules, définir des points de passage entre ces cellules et calculer, à l'aide de l'algorithme A* un corridor qui contiendra le chemin solution.

Le planificateur Cell-RRT est aidé d'un module de gestion des calculs heuristiques qui a pour objectif de répondre aux demandes de conseil du planificateur de tâches, ainsi que d'un module de satisfaction de contraintes géométriques pour la satisfaction des préconditions d'attitude. Les préconditions de comportement et donc les déplacements du robot durant la réalisation des actions sont gérées par un second planificateur de déplacements qui s'inspire de l'algorithme RRT-Extend et qui prend en entrée un critère d'arrêt au lieu d'une configuration de destination.

Le logiciel résultant des travaux menés dans ce chapitre a donc la capacité de traiter des problèmes impliquant planification de tâches et planification de déplacements. L'intérêt de l'approche reste à être validé sur des scenarios de planification pertinents. C'est l'objet du chapitre suivant.

Chapitre 5

Expérimentations et illustrations de l'architecture

Dans ce chapitre, nous proposons, dans un premier temps, un ensemble d'expérimentations permettant d'évaluer les performances du planificateur de déplacements Cell-RRT ainsi que de l'évolution de ces performances en fonctions de l'ajustement des différents paramètres de configuration.

Puis, dans un deuxième temps, nous illustrons le fonctionnement et les capacités de l'architecture de planification hybride proposée sur des exemples de satisfactions des contraintes d'attitude, de comportements particuliers et de scénarios de mission pour un robot mobile.

5.1 Expérimentations de Cell-RRT

Nous avons effectué des expérimentations sur un ensemble d'environnements non structurés. Ces expérimentations ont pour objectif de démontrer, d'une part, l'intérêt du découpage de l'environnement afin de restreindre l'espace de recherche à un corridor et, d'autre part, les évolutions de performance obtenues en fonction des différents paramètres de configuration du planificateur ainsi que des optimisations utilisées.

5.1.1 Environnement de test et expériences

Afin de mesurer les performances du planificateur de déplacements Cell-RRT et de comparer les différentes configurations des paramètres de configuration possibles, nous avons mené des tests de recherche d'un chemin solution sur un ensemble de 20 environnements non structurés, c'est-à-dire générés aléatoirement. Les résultats présentés dans la suite de cette première partie sont des

moyennes des valeurs obtenues lors de la recherche d'un chemin solution pour 1000 exécutions sur chacun des environnements et pour chaque découpage.

Les environnements utilisés sont des cartes en deux dimensions de 800*600 points. Chaque point correspond à un carré de côté égal à une unité de distance. Les obstacles sont modélisés par des polygones. La figure 5.1 représente trois environnements parmi les 20 qui ont servis aux expérimentations en environnement non structurés. Pour tous les environnements, la configuration initiale est située à l'extrémité sud-ouest de la carte et la configuration finale à l'extrémité nord-est.

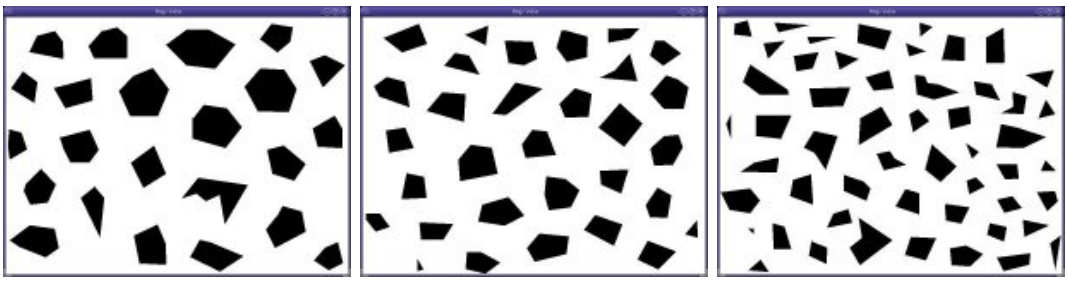


FIGURE 5.1 – Exemple de trois environnements non structurés

Les paramètres de configurations du planificateur que nous avons fait varier et dont nous étudions les effets dans la suite de ce chapitre sont :

- la granularité du découpage de l'environnement ;
- le nombre maximum de configurations tirées par l'algorithme RRT ;
- la méthode de tirage aléatoire des configurations ;
- l'heuristique utilisée par l'algorithme A* pour la recherche d'un corridor ;
- la réutilisation des arbres RRT et des segments de chemin précédemment calculés ;
- la valeur du seuil de traversabilité d'une cellule.

Paramètres de configuration du robot.

Le robot utilisé pour les expérimentations est celui défini par le modèle cinématique de l'équation 4.1 (chapitre 4). Le robot est initialisé avec une vitesse v de 10 unités par seconde et un angle de giration maximal ϕ de $\pi/20$ radians. La longueur L de l'essieu du robot est fixée à une unité.

Expériences

Dans un premier temps, nous justifions l'intérêt de restreindre l'environnement à un corridor avant d'appliquer l'algorithme RRT. Puis nous étudions l'évolution des résultats obtenus en fonction du nombre maximum de configurations qui peuvent être tirées aléatoirement. Ensuite, nous présentons les résultats

obtenus avec différentes méthodes de tirage aléatoire des configurations. Nous proposons également une étude de l'impact du choix du critère à optimiser dans l'algorithme A* lors de la recherche du corridor. Enfin, nous montrons que la réutilisation des arbres et des segments de chemin déjà calculés peut améliorer les résultats précédemment obtenus.

5.1.2 Intérêt du découpage de l'environnement

Ces premières expérimentations ont pour objectif de démontrer l'intérêt du découpage de l'environnement en un ensemble de cellules puis la réduction de l'espace de recherche à un corridor solution. Dans ce paragraphe, nous étudions l'impact de notre approche sur le temps de calcul d'un chemin solution ainsi que sur la qualité de cette solution en termes de distance parcourue.

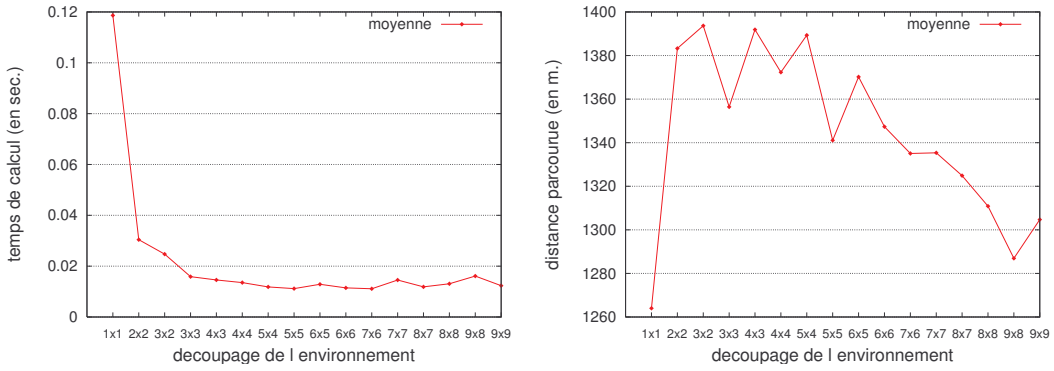


FIGURE 5.2 – Temps de calcul et distance en fonction du découpage

Sur la figure 5.2, on remarque que le fait de découper l'environnement en 2x2 cellules permet de réduire le temps de calcul de 75% et le découpage en 5x5 cellules permet de le réduire de 91% par rapport au temps de calcul d'un chemin solution sans phase de découpage.

Ce temps de calcul est minimal pour un découpage de l'environnement en 5x5 cellules. Pour une taille de cellule inférieure, le temps de calcul subit de légères variations tendant à indiquer que certains découpages pour certains environnements ne permettent pas d'obtenir directement un chemin solution et nécessitent plusieurs cycles de planifications. Pour une taille de cellule supérieure à celle donnant le temps minimal, on peut faire l'hypothèse que celui-ci décroit avec la taille des cellules par le fait que la surface explorée décroit également avec la taille des cellules.

Cependant, le découpage de l'environnement induit une dégradation de la distance parcourue par rapport à l'absence de découpage. Cette dégradation diminue avec la taille des cellules du fait que les points de passage du corridor

entre la configuration initiale et la configuration finale induisent un chemin plus direct. Néanmoins, cette dégradation persiste pour une taille de cellule minimale. L'analyse du graphe représentant la distance parcourue nous indique qu'elle varie entre 1,5% (9x8 cellules) et un peu plus de 10% (3x2 cellules).

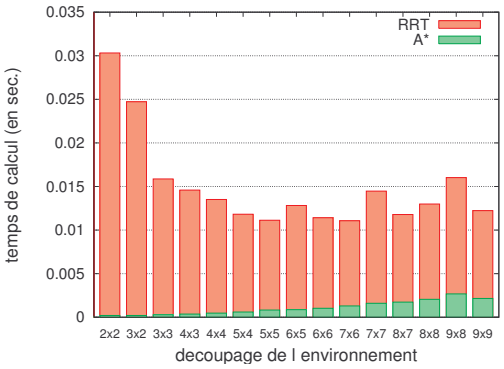


FIGURE 5.3 – Cumul temps A*/RRT en fonction du découpage

Par ailleurs, le temps de calcul utilisé pour la recherche d'un corridor à l'aide de l'algorithme A* peut être considéré comme négligeable par rapport au temps nécessaire pour la recherche d'un chemin solution par l'algorithme RRT (figure 5.3). Cependant, ce ratio augmente avec le découpage. L'augmentation du temps utilisé pour la recherche du corridor est due à l'augmentation du nombre de cellules prises en compte lors de cette recherche, mais est également due à un nombre de replanifications nécessaires croissant (figure 5.4).

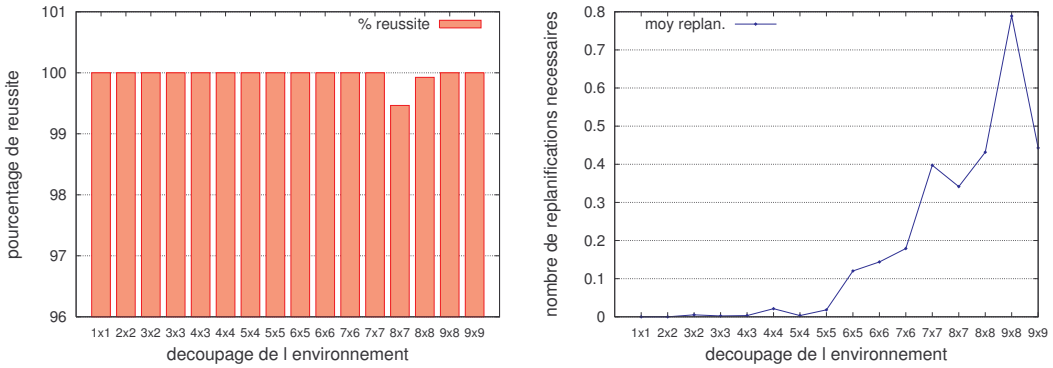


FIGURE 5.4 – Influence du découpage sur le nombre de solutions et replanifications

En effet, plus la surface des cellules diminue, plus le nombre de replanifications nécessaires augmente. L'absence de solution pour un environnement peut être expliquée par deux raisons :

1. Soit l'algorithme A* ne peut pas trouver de corridor satisfaisant les contraintes, c'est-à-dire que le ratio de traversabilité d'une cellule potentielle est supérieur au seuil fixé, ou alors que la cellule ne peut pas être traversée dans un certain sens, ce qui arrive lorsque des obstacles empêchent la définition du point de passage entre deux cellules ;
2. Soit l'algorithme RRT ne permet pas de définir de segment de chemin solution dans au moins une cellule du corridor. Il y a plusieurs raisons possibles à un tel échec : le nombre de configurations tirées est insuffisant, ou bien, la configuration des obstacles de la cellule ne permet pas de calculer un chemin respectant les contraintes cinématiques du robot.

Il y a échec de la planification lorsqu'après plusieurs cycles de replanification on se trouve dans le cas 1 ; plus aucun corridor ne peut être défini.

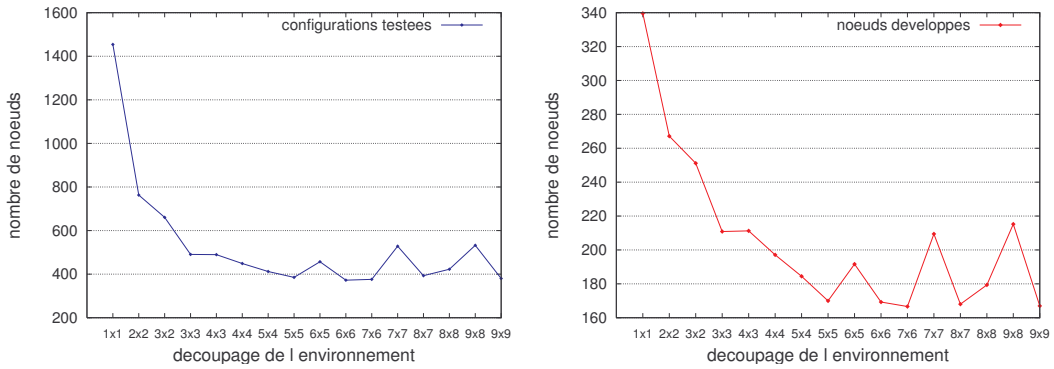


FIGURE 5.5 – Influence du découpage sur le nombre de configurations testées et nœuds développés

D'autre part, on remarque sur la figure 5.5 que le nombre de configurations testées et le nombre de nœuds développés semblent corrélés avec le temps de calcul. Le nombre de configurations testées correspond au nombre de configurations que l'on cherche à étendre en direction des configurations tirées aléatoirement. Le nombre de nœuds développés renseigne la taille cumulée de l'ensemble des arbres RRT développés lors de la recherche d'une solution. Le nombre de configurations testées pour le découpage en 5x5 cellules diminue de 73% par rapport au nombre de configurations testées lors de la recherche d'un chemin sans utiliser de découpage tandis que le nombre de nœuds développés diminue de 50%.

Ces résultats indiquent, d'une part, que la réduction de l'environnement à un corridor permet de limiter le nombres de configurations testées, c'est-à-dire qu'elle permet de limiter la création de nouvelles configurations et donc le nombre d'applications du modèle cinématique, et d'autre part, qu'elle permet de limiter le nombre de nœuds ajoutés aux arbres. Le rapport entre le nombre de nœuds développés et le nombre de configurations testées passe de 34/140

pour le cas sans découpage à approximativement 18/40 pour les découpages les plus fins. Cette décroissance indique que, avec un découpage fin, ces nouvelles configurations sont plus souvent valides et donc intégrées aux arbres RRT. Cette constatation peut être expliquée par le fait que lorsqu'une configuration est tirée localement, *i.e.,* dans une petite cellule, elle dirige plus probablement le déplacement du robot sur un temps Δt hors des obstacles locaux que lorsqu'elle est tirée globalement dans l'environnement.

Le développement d'un nouveau nœud nécessite le calcul d'une nouvelle configuration à partir du modèle cinématique du robot. Ce calcul est coûteux en temps. D'où la corrélation entre l'évolution des configurations testées selon la diminution de la taille des cellules et celle du temps de calcul global d'un chemin solution.

5.1.3 Influence du nombre maximum de configuration tirées

Une des variables d'ajustement de l'algorithme RRT est le nombre maximum de configurations que l'algorithme peut tirer aléatoirement avant de renvoyer un échec. Plus cette valeur est élevée, plus la probabilité que le planificateur de déplacement renvoie une solution augmente. Il n'existe pas de travaux sur la valeur optimale de ce paramètre car il est très dépendant de l'environnement dans lequel le robot évolue, du modèle cinématique utilisé, ainsi que du pas d'incrément de l'algorithme. Cependant, pour le planificateur que nous avons développé, cette valeur a un fort impact sur les résultats obtenus.

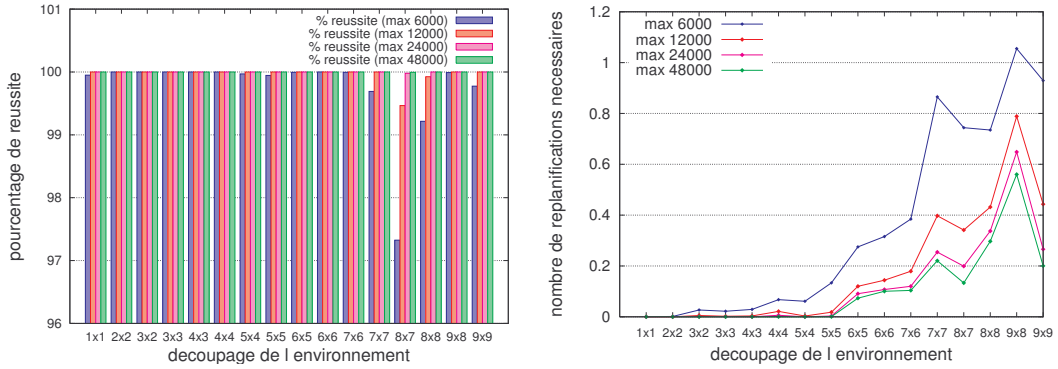


FIGURE 5.6 – Influence du nombre de configurations tirées sur le taux de réussite et le nombre de replanifications

Le nombre maximum de configurations qui peuvent être tirées durant tout le processus de calcul d'un chemin solution impacte directement le taux de réussite (figure 5.6). Plus cette valeur est élévée, plus la probabilité de trouver 100% des solutions est forte. Pour les environnements qui ont servis de support aux

expériences, une valeur comprise entre 24000 et 48000 configurations tirées permet d'obtenir 100% de réussite quelque soit le découpage utilisé. Ainsi, choisir une valeur élévée permet de garantir le meilleur taux de succès. Nous remarquons également que l'augmentation du nombre de configurations tirées permet de diminuer le nombre de replanifications.

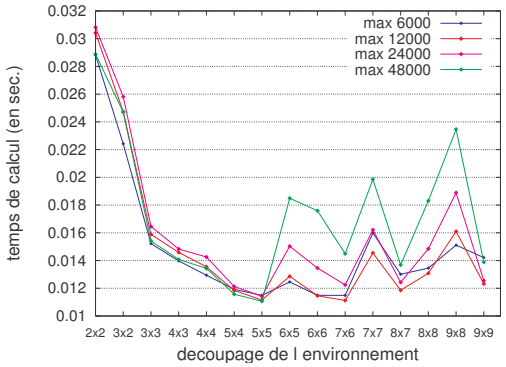


FIGURE 5.7 – Influence du nombre de configurations tirées sur le temps de calcul et la distance

Bien que permettant de diminuer le nombre de replanifications nécessaires, cette augmentation du nombre de configurations tirées a également une influence négative sur le temps de calcul (figure 5.7). À partir d'un découpage en 5x5 cellules, le temps de calcul augmente avec la valeur maximale de configurations tirées. Par exemple, pour un découpage en 9x8 cellules, le temps de calcul, pour une valeur de 48000 configurations, est de 53% supérieur au temps de calcul pour une valeur maximale de 6000 configurations tirées.

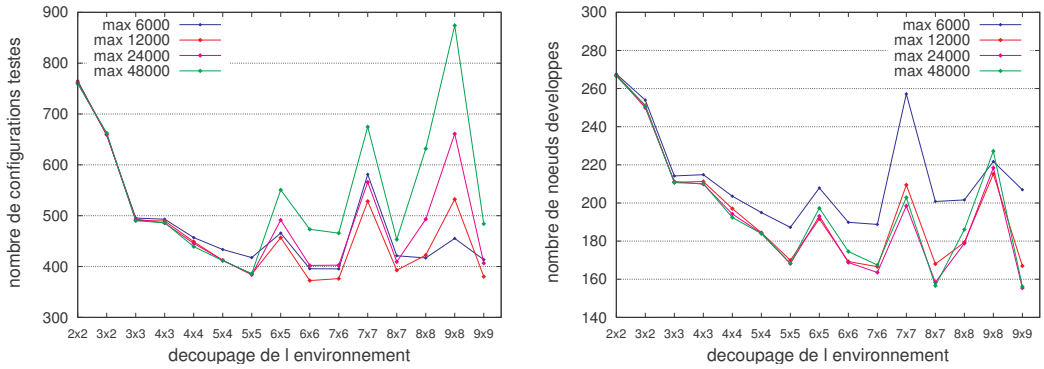


FIGURE 5.8 – Influence du nombre de configurations tirées sur le nombre de configurations testées et nœuds développés

La figure 5.8 représente le nombre de configurations testées ainsi que le nombre de nœuds développés en fonction du nombre maximum de configurations

tirées. On remarque que le nombre de configurations testées est plus faible avec un seuil élevé pour un découpage inférieur à 5x5 cellules. Ceci peut s'expliquer par une absence de replanification, *i.e.,* une solution est trouvée avant que le seuil maximal n'ait été atteint. Par contre, pour un découpage supérieur à 5x5 cellules, le nombre de configurations testées est dépendant du seuil fixé. Plus le seuil est élévé plus le nombre de configurations testées est élevé. Pour un découpage en 9x8 cellules, le nombre de configurations testées pour un seuil de 48000 configurations est 93% plus élevé que le nombre de configurations testées avec un seuil de 6000. Dans ce cas, on ne décide pas assez tôt d'une replanification.

Concernant le nombre de nœuds développés, on remarque que, quelque soit le découpage, un seuil élevé permet de développer moins de nœuds du fait des replanifications évitées.

5.1.4 Influence de la méthode de tirage

Dans les expérimentations précédentes, la méthode de tirage aléatoire d'une nouvelle configuration est uniforme. Nous avons vu précédemment que ce tirage pouvait être biaisé. Le but de l'utilisation d'un biais est de favoriser une extension des arbres RRT en direction des configurations finales. Dans ces nouvelles expérimentations, nous comparons le tirage uniforme d'une nouvelle configuration avec des méthodes de tirage pour lesquelles il y a 25% et 50% de probabilité qu'une nouvelle configuration soit tirée dans un périmètre de 10 unités autour de la configuration finale.

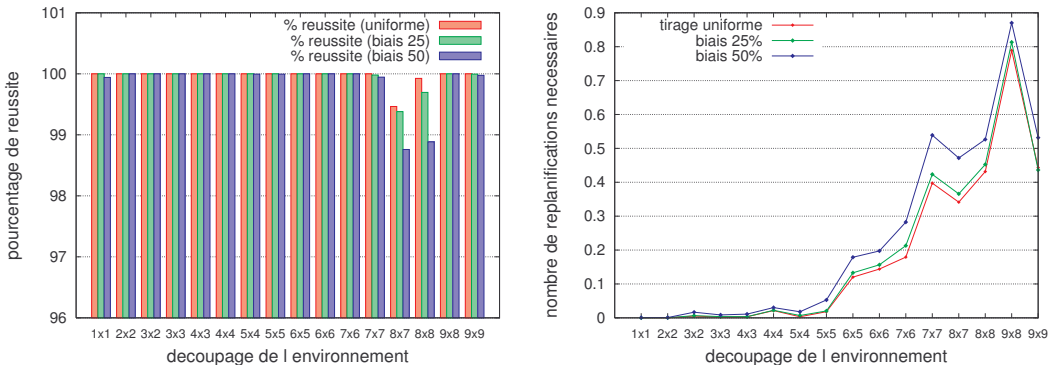


FIGURE 5.9 – Influence de la méthode de tirage sur le taux de réussite et le nombre de replanifications

Les graphiques de la figure 5.9 indiquent que plus le biais augmente plus le nombre de replanifications nécessaires augmente et la probabilité de trouver une solution diminue. Cette constatation est particulièrement valable pour les

granularités de découpage qui ne sont pas adaptées aux environnements utilisés lors des expérimentations.

On peut remarque sur la figure 5.10 que l'utilisation d'un tirage aléatoire biaisé pour le choix des configurations augmente le temps de calcul d'un chemin solution. Cette augmentation est de l'ordre de 10 à 15%.

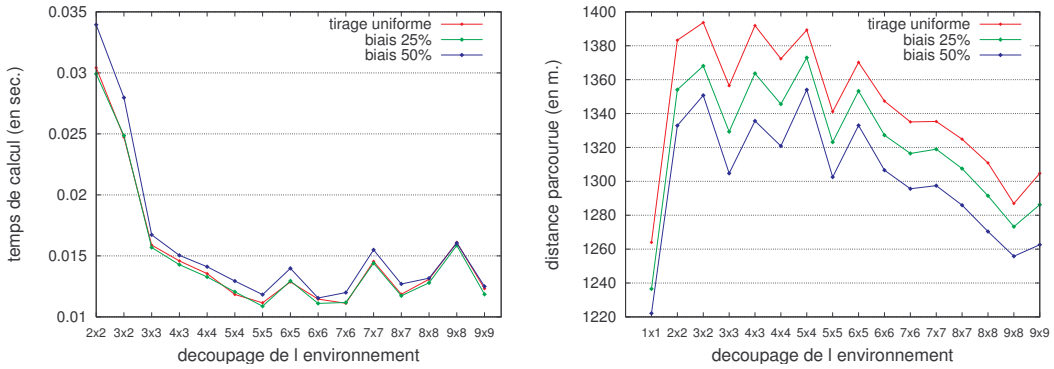


FIGURE 5.10 – Influence de la méthode de tirage sur le temps de calcul et la distance

Par contre, ce biais permet de limiter la perte de qualité obtenue lors de l'utilisation d'un découpage. Il est intéressant de remarquer que ce gain de performance en termes de distance parcourue est constant quelque soit la taille des cellules et est de l'ordre de 3% pour un biais de 50%. Pour un découpage en 9x8 cellules la longueur du chemin solution est même inférieure à celle de la solution obtenue sans découpage et sans biais.

5.1.5 Influence du critère optimisé par A*

Un autre paramètre ajustable du planificateur de déplacement est le critère optimisé dans la méthode A* lors la recherche d'un corridor pour le choix d'une cellule à explorer. En effet, deux types d'information peuvent être pris en compte : la distance entre la nouvelle cellule et la cellule précédente et la traversabilité de cette nouvelle cellule. Nous avons expérimenté trois critères différents :

1. g_{normal} : la distance euclidienne est pondérée par le ratio de traversabilité de la cellule. Ce critère est le critère utilisé lors des autres expérimentations présentées dans ce chapitre et sert donc de référence aux deux autres ;
2. $g_{obstacles}$: seule la traversabilité de la cellule est utilisée ;
3. $g_{distance}$: la distance euclidienne est prise en compte uniquement ;

La figure 5.11 représente le pourcentage de réussite et le nombre de replanifications nécessaires lors de l'utilisation de chacun des trois critères. On

remarque que le critère $g_{distance}$ est celui qui nécessite le plus de replanifications mais présente en même temps le meilleur taux de réussite. Le critère $g_{obstacles}$ est moins performant que $g_{distance}$ du point de vue du taux de réussite. Cependant il est meilleur que g_{normal}. Il permet également d'éviter plus de replanifications que $g_{distance}$ et g_{normal} lorsque le nombre de cellules augmente. Ceci s'explique par le fait que l'algorithme recherche un chemin solution passant en priorité dans les cellules ayant peu d'obstacles.

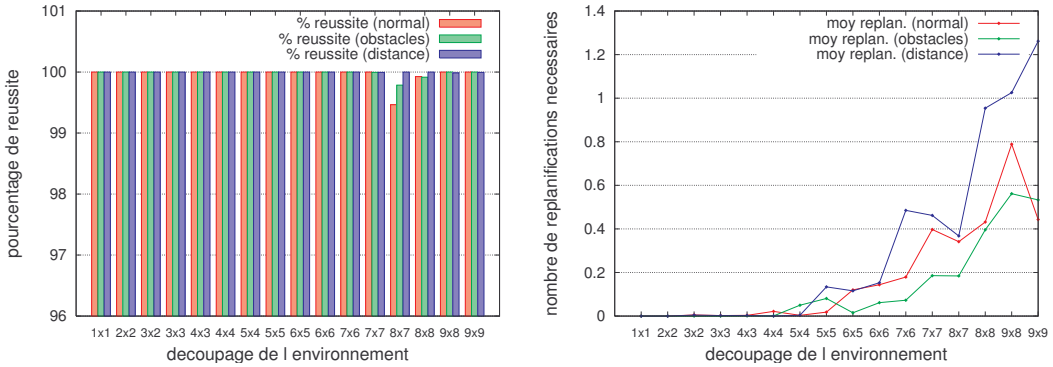


FIGURE 5.11 – Influence du choix du critère sur le taux de réussite et le nombre de replanifications

On remarque sur les graphiques de la figure 5.12 que l'influence du choix du critère sur le temps de calcul est difficilement quantifiable. L'utilisation du critère $g_{distance}$ semble augmenter le temps de calcul lorsque le découpage augmente du fait du nombre plus élevé de replanifications.

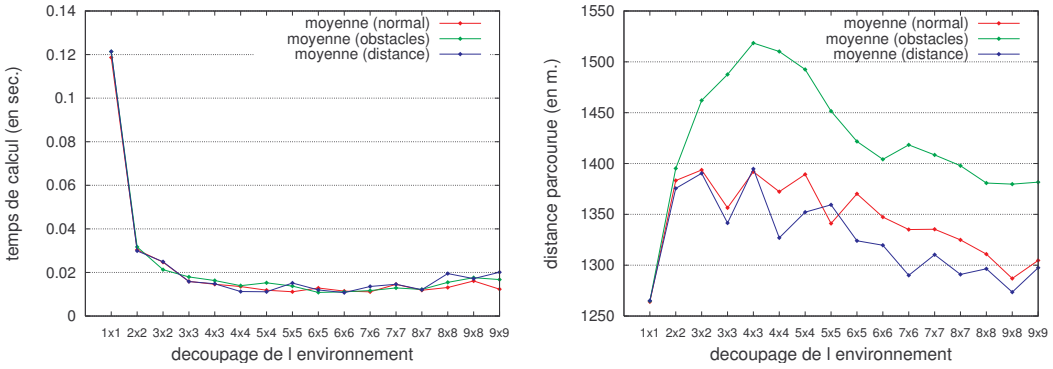


FIGURE 5.12 – Influence du choix du critère sur le temps de calcul et la distance

Par contre, le choix de ce critère influe fortement sur la qualité de la solution en termes de distance parcourue par le robot. Quelque soit le découpage de l'environnement, le critère $g_{obstacles}$ présente les pires performances car, afin d'éviter au maximum les obstacles, le corridor entre la configuration initiale et

la configuration finale est moins direct. Cette perte de performance est supérieure à 20% au pire cas, contrairement aux 10% de dégradation obtenus avec le critère g_{normal}. Le critère $g_{distance}$ est le meilleur car il favorise la recherche d'une solution la plus directe possible.

5.1.6 Réutilisation d'arbres et de segments de chemin solution

Réduire l'environnement à un corridor permet de réduire l'espace de recherche de l'algorithme RRT et donc de diminuer le temps de calcul. Cependant, il n'existe pas forcément de solutions permettant de connecter à l'aide d'un chemin la configuration initiale et la configuration finale. Des replanifications peuvent être nécessaires afin de définir un nouveau corridor dans lequel un chemin solution sera recherché. Ce nouveau corridor peut contenir des cellules qui faisaient partie de l'ancien corridor. Dans ce cas, si les configurations initiales et finales de la cellule sont les mêmes et s'il existe un chemin permettant de la traverser, alors celui-ci n'a pas besoin d'être recalculé. De plus, les arbres RRT précédemment calculés peuvent être réutilisés afin de limiter le processus de recherche d'une solution.

Nous avons comparé les résultats de la version basique avec les valeurs obtenues lorsque le planificateur peut réutiliser les arbres précédemment calculés, les segments de chemins déjà définis, ainsi que les arbres et segments simultanément.

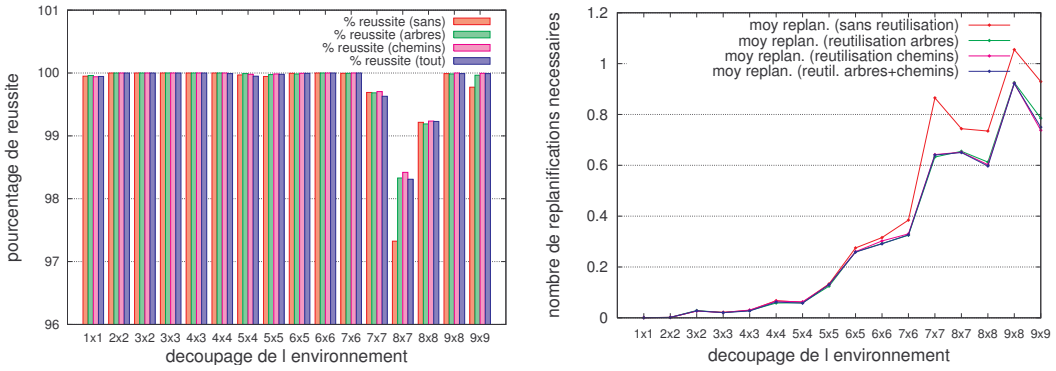


FIGURE 5.13 – Influence de la réutilisation sur le taux de réussite et le nombre de replanifications

Les expérimentations ont été menées avec un nombre maximum de 6000 configurations tirées par expériences.

On remarque sur la figure 5.13 qu'en permettant au planificateur de réutiliser les arbres et chemins précédemment calculés, le taux de réussite est globalement amélioré. C'est le cas pour les expériences menées avec un découpage

en 8x7 cellules et en 9x9 cellules. Cependant, cette réutilisation ne permet pas d'obtenir un meilleur taux de réussite pour le découpage en 8x8 cellules. Par ailleurs, cette réutilisation permet de diminuer le nombre de replanifications nécessaires lorsque la taille des cellules diminue. On déduit des deux graphiques que la réutilisation des arbres et chemins des cycles de replanification précédents permet d'obtenir plus facilement une solution pour les cellules dont la première recherche d'un chemin est complexe mais pas pour les cellules dont la traversée est impossible.

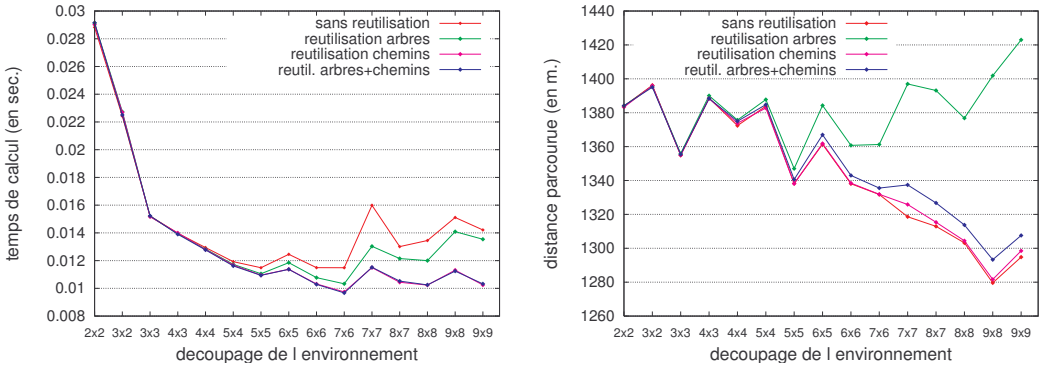


FIGURE 5.14 – Influence de la réutilisation sur le temps de calcul et la distance

Les graphiques de la figure 5.14 indiquent que cette réutilisation permet de diminuer le temps de calcul lorsque le découpage augmente. Cet effet est principalement du à la réutilisation des segments de chemin préalablement calculés. Cependant, on remarque que cette réutilisation offre des solutions moins optimales en termes de longueur de chemins. Ceci est particulièrement le cas lorsque les arbres sont réutilisés. En effet, plus l'arbre réutilisé est développé, plus la probabilité que le nœud le plus proche soit sur une branche "directe" est faible. Lorsque seuls les chemins sont réutilisés, les distances parcourues sont identiques à celles obtenues sans réutilisation.

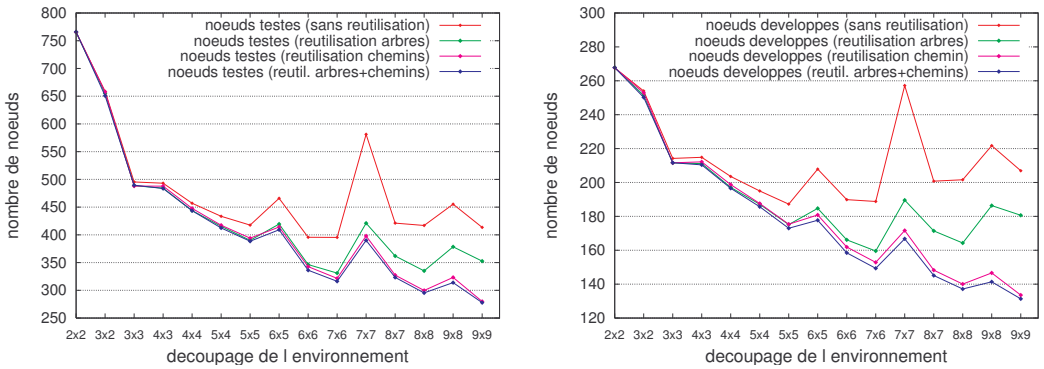


FIGURE 5.15 – Influence de la réutilisation sur le nombre de noeuds testés / développés

La réutilisation des arbres et des chemins calculés permet de réduire le nombre de configurations testées et le nombre de nœuds développés à partir d'un découpage en 3x3 cellules (figure 5.15). Cette diminution atteint 33% pour un découpage en 9x9 cellules. Ces observations expliquent la réduction du temps de calcul et conforte l'intérêt de réutiliser les segments de chemin et les arbres précédemment calculés.

5.1.7 Influence du seuil de traversabilité

Le dernier paramètre de configuration du planificateur de déplacement que nous étudions est le seuil de traversabilité d'une cellule. Ce seuil définit le ratio espaces libres / zones d'obstacle de la cellule. un ratio de 1 signifie que la cellule est entièrement libre, un ratio de 0 indique qu'elle est entièrement occupée par les obstacles. Lors de la réduction de l'environnement à un corridor, l'algorithme A* utilise ce seuil de traversabilité afin de décider des cellules à ajouter au corridor.

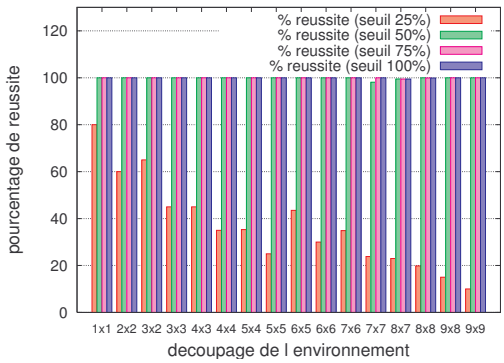


FIGURE 5.16 – Influence du seuil de traversabilité sur le taux de réussite

La figure 5.16 représente le taux de réussite du planificateur pour différents seuils de traversabilité. On s'aperçoit qu'il existe un seuil minimal pour lequel le taux de réussite est proche de 100%. Pour un seuil de traversabilité inférieur à cette valeur, le taux de réussite décroit avec la surface des cellules. Cependant cette valeur est fortement dépendante de la topologie de l'environnement concerné.

Le graphe 5.17 indique qu'à partir d'un seuil de traversabilité de 50% le nombre de replanifications nécessaires est sensiblement le même quelque soit le découpage. On remarque également que ce nombre de replanifications est très faible pour un seuil de traversabilité de 25%. Ceci est du au fait que, dans de nombreux cas, la première cellule, *i.e.,* la cellule contenant la configuration initiale, n'est pas traversable. Plus aucun corridor ne peuvant être trouvé, le planificateur renvoie un échec.

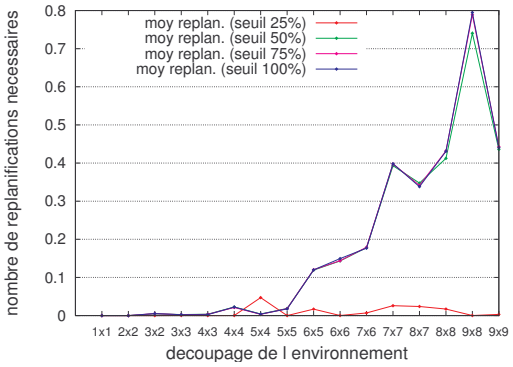


FIGURE 5.17 – Influence du seuil de traversabilité sur le nombre de replanifications

5.1.8 Bilan

Dans cette première partie, nous avons effectué une étude expérimentale du planificateur de déplacements CELL-RRT qui a été proposé au chapitre précédent. Cette étude met en avant l'intérêt de l'utilisation d'une phase de découpage de l'environnement afin de restreindre l'espace de recherche à un corridor et, d'autre part définit l'influence des différents paramètres de configuration de ce planificateur sur le temps de calcul et la qualité de la solution.

Les expérimentations ont été menées sur un ensemble d'environnements non structurés et ont montré que la restriction de l'environnement permettait une diminution du temps de calcul d'au moins 75% avec une dégradation de la qualité de la solution en termes de distance parcourue de moins de 15%. Nous avons vu par la suite que le temps de calcul et la qualité de la solution pouvaient être améliorés en règlant de manière appropriée les différents paramètres de configuration :

- le nombre maximum de configurations tirées permet de décider au plus tôt d'une replanification ;
- la méthode de tirage biaisée des nouvelles configurations permet d'optimiser la qualité de la solution en termes de distance mais a une influence négative sur le temps de calcul ;
- le choix du critère à optimiser par A* permet d'éviter au plus tôt certaines cellules encombrées ou au contraire de favoriser la recherche d'un chemin le plus court ;
- la réutilisation des arbres et segments de chemin précédemment calculés permet d'éviter des recherches inutiles et donc de diminuer le temps de calcul.

5.2 Illustration du fonctionnement et des capacités de l'architecture

Dans cette seconde partie, nous illustrons le fonctionnement de l'architecture de planification hybride mise en œuvre. Dans un premier temps, nous présentons un aperçu des interfaces utilisateurs permettant de spécifier les données nécessaires à la résolution d'un problème de planification pour un robot mobile. Puis nous détaillons, sur un exemple, les calculs effectués par le module de raisonnement géométrique lors de la satisfaction de préconditions géométriques. Enfin nous présentons et illustrons les résultats obtenus lors de la planification de comportement géométrique particuliers ainsi que lors de la résolution de missions pour un robot mobile durant lesquelles les actions à réaliser nécessitent des déplacements et comportements bien précis.

5.2.1 Aperçu des interfaces utilisateur

Interface principale

Le logiciel développé comprend un ensemble d'interfaces permettant de configurer et d'exécuter les problèmes de planification, ainsi que de visualiser les résultats du processus de planification aussi bien du point de vue du plan symbolique que du plan des déplacements. L'image 5.18 est une capture d'écran de la fenêtre de visualisation des résultats de l'architecture développée.

Ces interfaces visent à permettre une validation du processus de planification avant intégration dans l'architecture de commande du robot.

Configuration du projet

L'architecture de planification développée prend en entrée un projet de planification. Ce projet est composé de :

- une carte de l'environnement sous la forme d'une image dont les pixels de couleur noire correspondent aux zones de l'environnement non traversables, *i.e.,* les obstacles ;
- un domaine de planification décrivant la liste des actions possibles dans le formalisme présenté précédemment ;
- une description des capacités du robot utilisé (vitesse, angle de giration maximum, ...) ;
- une liste des objets de l'environnement. Chaque objet est décrit par une étiquette symbolique et des coordonnées ;
- un problème de planification décrivant l'état initial et la liste des tâches à planifier.

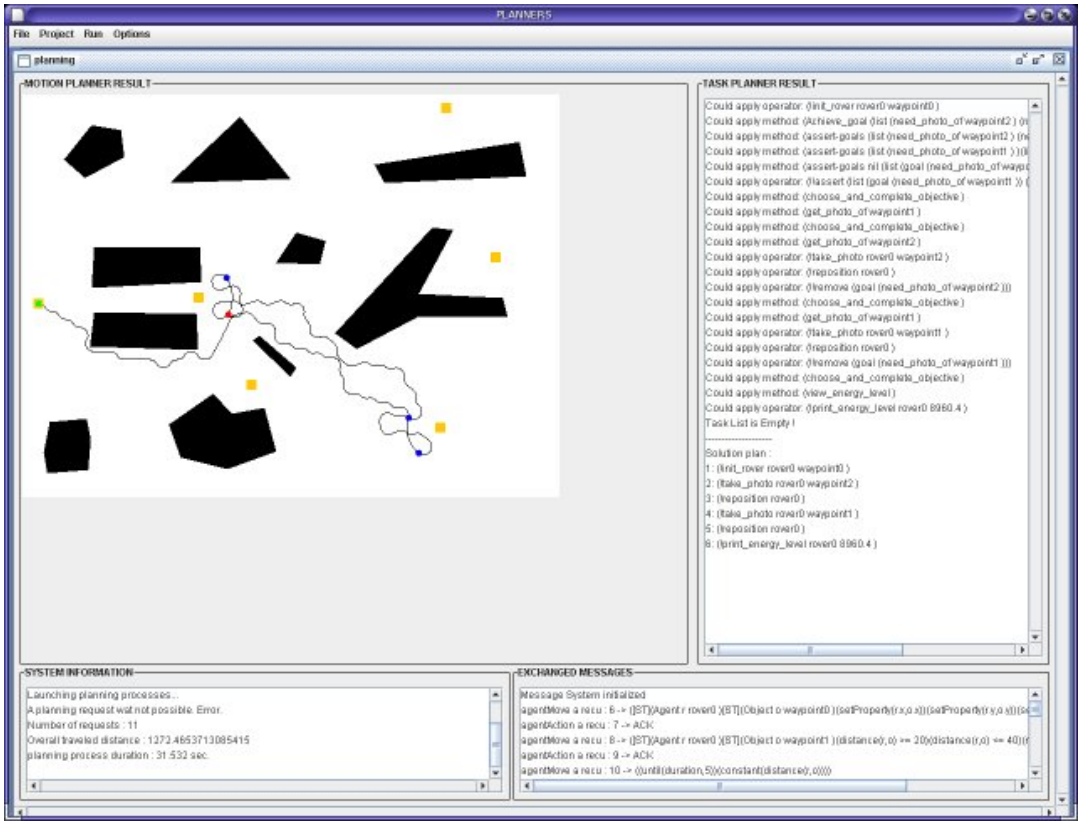


FIGURE 5.18 – Fenêtre principale d'affichage des résultats

Nous avons développé un ensemble d'interfaces de saisie et de visualisation permettant de créer et modifier les projets de planification. Ces différentes interfaces sont présentées en annexe C.

5.2.2 Fonctions de contraintes utilisées pour les illustrations

Une donnée nécessaire au module de raisonnement géométrique pour lui permettre de satisfaire les requêtes de planification du module de raisonnement géométrique est l'ensemble des formules mathématiques des fonctions et dérivées partielles intervenant dans les préconditions géométriques.

L'algorithme d'analyse du fichier de fonctions mathématiques que nous avons développé utilise des variables typées. Ainsi, nous pouvons définir trois types de fonction de contrainte : les fonctions prenant en argument un agent et des objets ou constantes, les fonctions ayant pour argument uniquement des objets et constantes et les fonctions utilitaires permettant d'effectuer des calculs sur des constantes numériques.

Les tableaux 5.1, 5.2 et 5.3 complètent les exemples du tableau 3.2 et illustrent l'ensemble des éléments de contraintes auxquels sont associées des fonctions mathématiques et qui interviennent dans les illustrations de ce chapitre.

`heading(agent)`	Cap du robot
`position(agent)`	Coordonnées de la position du robot
`distance(agent, objet)`	Distance euclidienne entre la position de l'agent et la position de l'objet
`distance_coord(agent, x, y)`	Distance euclidienne entre la position de l'agent et la position (x,y)
`rel_angle(agent, objet)`	Cosinus et sinus de l'angle relatif entre le cap de l'agent et le segment de droite entre la position de l'agent et celle de l'objet
`rel_angle2(agent, objet, objet)`	Cosinus et sinus de l'angle relatif entre le cap de l'agent et le segment de droite entre les positions des deux objets

TABLEAU 5.1 – Fonctions ayant pour argument un agent

5.2.3 Exemple de satisfaction des contraintes d'attitude

Dans le chapitre précédent, nous avons présenté l'algorithme mis en œuvre pour la satisfaction des préconditions d'attitude. Nous illustrons au travers d'un

`translate_x(objet, c)`	Abscisse de la position de l'objet plus c unités
`translate_y(objet, c)`	Ordonnée de la position de l'objet plus c unités
`dist_obj(objet, objet)`	Distance euclidienne entre les positions des deux objets
`rotation(ref,objet,a,s)`	Rotation de la position exprimée par la référence sur un cercle de centre objet selon un angle a dans le sens s

TABLEAU 5.2 – Fonctions ayant pour argument des objets

`mult(c1, c2)`	Multiplication entre c1 et c2
`cos_and_sin(angle)`	Cosinus et sinus de l'angle passé en paramètre et exprimé en radian

TABLEAU 5.3 – Fonctions de contraintes sur des constantes numériques

exemple, les calculs effectués par cet algorithme pour satisfaire un ensemble de préconditions d'attitude qui spécifie que le robot r doit se placer à une distance comprise entre 10 et 20 unités de distance de l'objectif o.

Soit les préconditions géométriques :

$$\begin{cases} distance(r, o) \geq 10 \\ distance(r, o) \leq 20 \end{cases}$$

Nous obtenons les contraintes suivantes :

$$\begin{cases} c1 = distance(r, o) - 10 \geq 0 \\ c2 = -distance(r, o) + 20 \geq 0 \end{cases}$$

Ainsi :

$$\begin{cases} f_1 = \sqrt{(x_o - x_r)^2 + (y_o - y_r)^2} - 10 \\ f_2 = -\sqrt{(x_o - x_r)^2 + (y_o - y_r)^2} + 20 \end{cases}$$

Soit G_i la fonction de pénalisation pour f_1 et f_2

$$G_i(X) = \frac{1}{2} X^2 \mathcal{H}(X)$$

La fonction de pénalisation globale est :

$$F = \sum_{i=1}^{i=2} G_i = \frac{1}{2}(f_1)^2\mathcal{H}(f_1) + \frac{1}{2}(f_2)^2\mathcal{H}(f_2)$$

D'où :

$$\nabla F = \begin{bmatrix} \frac{\partial G_1}{\partial f_1} * \frac{\partial f_1}{\partial x} \\ \frac{\partial G_1}{\partial f_1} * \frac{\partial f_1}{\partial y} \\ 0 \end{bmatrix} + \begin{bmatrix} \frac{\partial G_2}{\partial f_2} * \frac{\partial f_2}{\partial x} \\ \frac{\partial G_2}{\partial f_2} * \frac{\partial f_2}{\partial y} \\ 0 \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\partial f_1}{\partial x} \\ \frac{\partial f_1}{\partial y} \\ 0 \end{bmatrix} f_1\mathcal{H}(f_1) + \begin{bmatrix} \frac{\partial f_2}{\partial x} \\ \frac{\partial f_2}{\partial y} \\ 0 \end{bmatrix} f_2\mathcal{H}(f_2)$$

Par exemple, soit $r_0 = [20, 100, 0]$ le vecteur d'état initial du robot et $o = [50, 50]$ la position de l'objectif,

$$distance(r_0, o) = 58.31$$

D'après les calculs précédents :

$$r_1 = r_0 - \lambda_0 \nabla F(r_0) = \begin{bmatrix} 20 + 19.71 \\ 100 - 32.85 \\ 0 \end{bmatrix} \approx \begin{bmatrix} 40 \\ 67 \\ 0 \end{bmatrix}$$

Avec $\lambda_0 = 1$.

Maintenant :

$$distance(r_1, o) = 19.72$$

Les contraintes c_1 et c_2 sont satisfaites, la configuration r_1 peut donc être utilisée comme destination par CELL-RRT.

5.2.4 Illustration de comportements particuliers

Les actions que doit réaliser le robot durant sa mission peuvent nécessiter des déplacements. Ces déplacements ont pour objectif de positionner le robot en vue de réaliser l'action et sont spécifiés par les préconditions d'attitude, ou de permettre la réalisation de l'action et correspondent aux préconditions de comportement. Lorsque les mouvements du robot nécessaires à la réalisation d'une action sont fortement contraints, on parle de **comportement particulier**.

Si le comportement particulier peut être décrit au niveau du planificateur de tâches par une action primitive, on qualifie ce comportement de **simple**.

S'il nécessite plusieurs actions, alors le comportement est un comportement **complexe**.

L'objectif des illustrations suivantes est de montrer que le modèle et le langage de planification que nous avons proposés et mis en œuvre permettent d'exprimer des comportements géométriques nécessaires à la réalisation d'une action, qu'ils soient simples ou complexes.

Comportements simples

Dans ce paragraphe, nous illustrons deux comportements simples : le parcours en ligne droite et le parcours en arc de cercle.

Parcours en ligne droite. L'objectif de cette action est de forcer le robot à se déplacer en ligne droite entre deux objets. Pour réaliser ce comportement, le robot doit se placer aux coordonnées du premier objet avec un cap pointant sur le deuxième objet. Puis il doit se déplacer sur une distance égale à la distance entre les deux objets en maintenant son cap constant. Cette action est décrite dans le domaine de planification par l'opérateur `!follow_wall` et est illustrée par la figure 5.19.

```
;; --- suivre un mur (spécifié par 2 objets)
(Operator (!follow_wall ?r ?o1 ?o2)
 ((rover ?r)(location ?o1)(location ?o2))
 (
  (agent ?r)(object ?o1)(object ?o2)
  (distance(?r, ?o1) = 0)
  (rel_angle2(?r, ?o1, ?o2) = cos-and-sin(0))
 )
 (
  (until(distance, dist_obj(?o1, ?o2)))
  (constant(?r.heading))
 )
 ()
 ()
)
```

Tour d'un objectif. Pour faire le tour d'un objet, le robot doit se placer à une certaine distance de l'objet et perpendiculairement à celui-ci. Puis il doit se déplacer en maintenant constante la distance le séparant de l'objet durant un trajet dont la longueur est égale au périmètre d'un cercle ayant pour rayon cette distance entre l'agent et l'objet. L'opérateur `!view_360` décrit ce comportement pour un robot qui doit filmer l'objectif avec, pour position initiale, une position

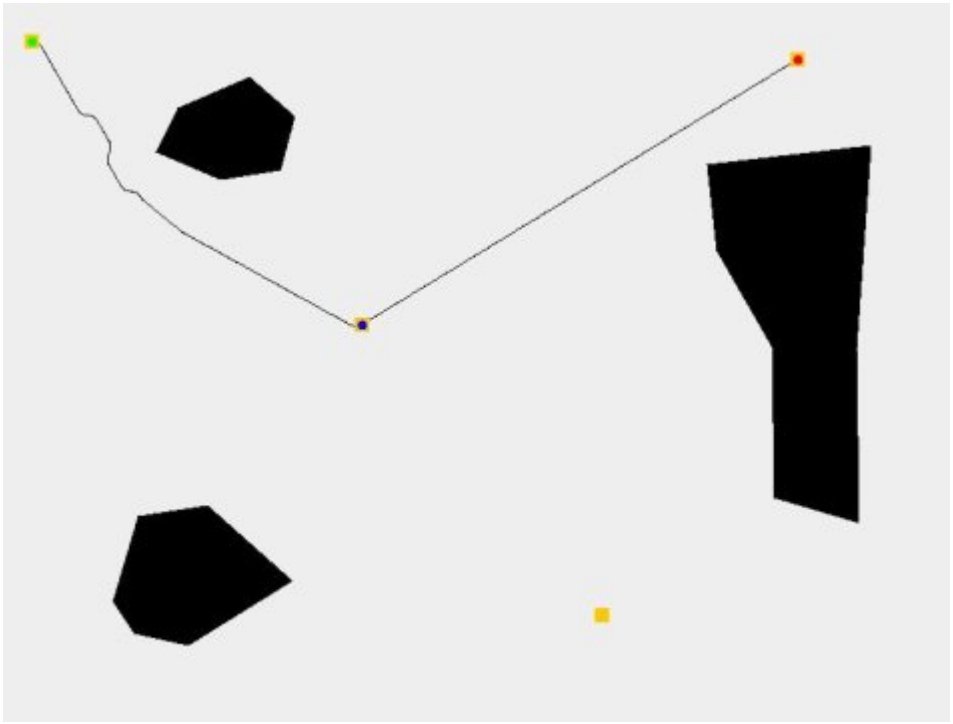


FIGURE 5.19 – Parcours en ligne droite

dont la distance à l'objectif est comprise entre 20 et 40 unités. La figure 5.20 illustre ce comportement.

```
;; --- filmer l'objectif en faisant le tour dans le sens direct
(Operator (!view_360 ?r ?o)
 ((rover ?r)(location ?o)(has_camera ?r HR_cam))
 (
  (agent ?r)(object ?o)
  (distance(?r, ?o) >= 20)
  (distance(?r, ?o) <= 40)
  (rel_angle(?r, ?o) = cos-and-sin(1.5708))
 )
 (
  (until(distance, mult(6.28318, distance(?r, ?o))))
  (constant(distance(?r, ?o)))
 )
 ()
 (
  (has_view_360_of ?o)
 )
)
```

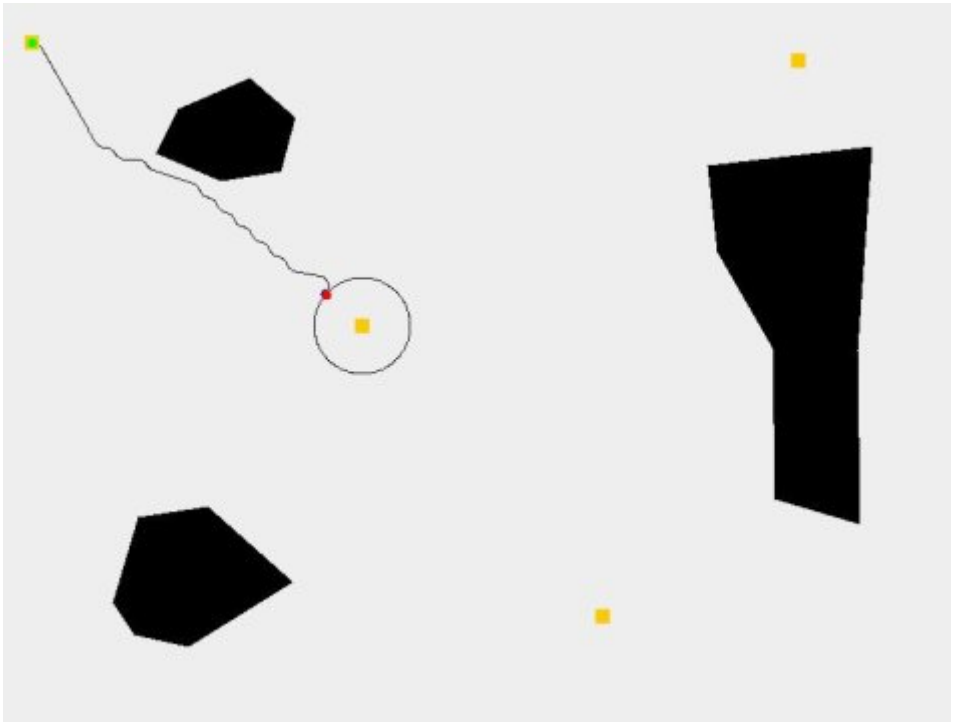


FIGURE 5.20 – Cercle autour d'un objectif

Comportements complexes

Ces comportements ne peuvent pas être décrits par un seul opérateur. En plus des contraintes géométriques les décrivant, ils nécessitent un enchaînement précis des actions. Cet enchaînement est décrit, d'une part, à l'aide de méthodes et, d'autre part, par l'utilisation de références géométriques au niveau des opérateurs. Nous présentons deux comportements complexes mis en œuvre : la prise de trois photos successives et le balayage d'une zone de l'environnement.

Prise de photos. Dans cet exemple, le robot doit prendre trois photos d'un objectif en utilisant une caméra située à l'avant du robot, *i.e.*, il doit être face à l'objectif. Chaque photo doit être prise à une position séparée de la précédente par un angle de 120 degrés ($2\pi/3$ radians) par rapport à l'objectif et à la même distance. Cette action permet, par exemple, de faire une reconstruction 3D d'un bâtiment à partir des photos prises. La figure 5.21 illustre ce comportement.

La méthode `Take_3V120_photos` exprime l'enchaînement nécessaire à la réalisation de l'action complexe.

```
;; --- prise de 3 photos vue de face avec angle de 120 deg.
(Method (Take_3V120_photos ?r ?o)
 ((rover ?r)(location ?o)(has_camera ?r FRONT_cam))
 ((!take_first_photo ?r ?o)
```

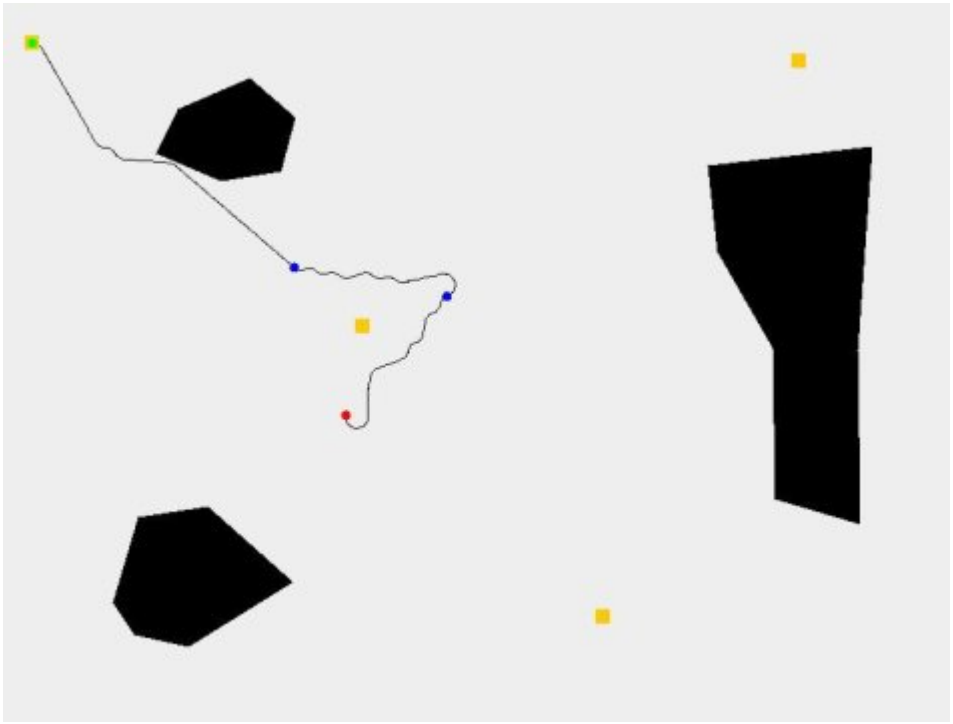


FIGURE 5.21 – Prise de trois photos séparées par un angle de 120 degrés

```
  (!take_next_photo ?r ?o)
  (!take_next_photo ?r ?o))
)
```

Pour prendre la première photo, le robot doit être à une distance comprise entre 70 et 75 unités de l'objectif. Son cap doit pointer sur l'objectif. La configuration du robot produite par cette première action sert de référence à l'action suivante. L'opérateur `!take_first_photo` décrit le comportement nécessaire à la réalisation de la première action.

```
;; --- prendre 1ere photo frontale
(Operator (!take_first_photo ?r ?o)
 ((rover ?r)(location ?o))
 (
    (agent ?r)(object ?o)
    (distance(?r, ?o) >= 70)
    (distance(?r, ?o) <= 75)
    (rel_angle(?r, ?o) = cos-and-sin(0))
 )
 ()
 ((@attitude ?refphoto))
 ((ref_photo ?refphoto))
)
```

Les configurations du robot nécessaires à la réalisation de la deuxième et troisième photos sont spécifiées au niveau des préconditions d'attitude de l'opérateur `!take_next_photo`. La position du robot est obtenue par rotation de la configuration associée à la référence produite par l'action précédente. La rotation se fait dans le sens indirect [1] selon un angle de 120 degrés.

```
;; --- prendre photo frontale suivante
(Operator (!take_next_photo ?r ?o)
 ((rover ?r)(location ?o)(ref_photo ?oldref))
 (
  (agent ?r)(object ?o)(reference ?oldref)
  (position(?r) = rotation(?oldref, ?o, 2.0944, 1))
  (rel_angle(?r, ?o) = cos-and-sin(0))
 )
 ()
 ((@attitude ?newref))
 ((not(ref_photo ?oldref))(ref_photo ?newref))
)
```

Balayage d'une zone. Le deuxième comportement complexe que nous illustrons est le balayage d'une zone en effectuant des lignes droites et des demi-tours. Ce comportement est spécifié uniquement par deux objets délimitant la zone à balayer ainsi que par un rayon de giration correspondant, d'une part, au rayon du demi-cercle permettant au robot de faire demi-tour et, d'autre part, à la distance séparant deux parcours en ligne droite.

La méthode `Sweep_zone` est la méthode principale de cette action de balayage. Elle fait appel au module de raisonnement géométrique pour déterminer les dimensions de la zone à balayer à l'aide de demande de conseils heuristiques. Puis elle appelle l'opérateur `!prepare_and_position` et lance le balayage décrit par la méthode `do_sweeping`.

```
;; --- balayage d'une zone
(Method (Sweep_zone ?r ?wp1 ?wp2 ?radius)
 ((rover ?r)(location ?wp1)(location ?wp2)
  (heuristic(v_distance ?wp1 ?wp2 ?vdist))
  (heuristic(h_distance ?wp1 ?wp2 ?hdist))
 )
 ((!prepare_and_position ?r ?wp1 ?hdist ?radius)
  (do_sweeping ?r ?vdist ?radius))
)
```

Une fois que le robot est correctement positionné, le balayage de la zone peut commencer. Ce balayage, décrit par la méthode `do_sweeping` est une suc-

1. le sens de rotation est spécifié par le dernier paramètre de la fonction rotation : 1 pour le sens indirect et -1 pour le sens direct.

cession de parcours en ligne droite et de demi-tours. La deuxième décomposition de la méthode indique que le balayage se termine par un parcours en ligne droite.

```
;; --- lance le balayage de la zone
(Method (do_sweeping ?r ?vd ?radius)
 ((sw_inc ?i)(sw_max ?m)(call < ?i ?m))
 ((!go_straight ?r ?vd)(!make_half_turn ?r ?radius)
  (do_sweeping ?r ?vd ?radius))
 ((sw_inc ?i)(sw_max ?m)(call >= ?i ?m))
 ((!go_straight ?r ?vd))
)
```

La première action primitive intervenant dans le balayage de la zone permet le positionnement du robot et l'insertion dans l'état courant de la référence géométrique sur cette configuration ainsi que de prédicats symboliques nécessaires au calcul du balayage. Cette action est décrite par l'opérateur `!prepare_and_position`.

```
;; --- positionne le robot et calcule le nombre d'AR
(Operator (!prepare_and_position ?r ?o ?hd ?radius)
 ((rover ?r)(location ?o))
 (
  (agent ?r)(object ?o)
  (distance(?r, ?o) = 0)
  (heading(?r) = 1.570796)
 )
 ()
 ((@attitude ?ref))
 ((current_pos ?ref)(sw_inc 0)
  (sw_max (call / ?hd (call * 2 ?radius)))
)
```

L'opérateur `!go_straight` décrit le parcours en ligne droite du robot. Le comportement produit est similaire au comportement simple décrit précédemment. Cependant, il n'est pas spécifié par deux objets mais par une distance. La position de référence est la position actuelle du robot.

```
;; --- aller tout droit sur ?dist mètres
(Operator (!go_straight ?r ?dist)
 ((rover ?r)(current_pos ?pos)(sw_inc ?i))
 ((agent ?r))
 (
  (until(distance, ?dist))
  (constant(?r.heading))
 )
 ((@behavior ?ref))
```

```
 ((not(current_pos ?pos))(current_pos ?ref)
  (not(sw_inc ?i))(sw_inc (call + ?i 1)))
)
```

L'opérateur `!make_half_turn` permet d'indiquer au robot qu'il doit effectuer un demi-tour, *i.e.,* il doit effectuer un arc de cercle dont la longueur est égale à un demi cercle de rayon précisé en paramètre de l'opérateur.

```
;; --- effectuer un demi-tour
(Operator (!make_half_turn ?r ?rad)
 ((rover ?r)(current_pos ?pos))
 (
  (agent ?r)(object ?pos)
 )
 (
  (until(distance, mult(3.14159, ?rad)))
  (constant(distance_coord(?r, translate_x(?pos, ?rad),
    translate_y(?pos, 0))))
 )
 ((@behavior ?ref))
 ((not(current_pos ?pos))(current_pos ?ref))
)
```

La figure 5.22 illustre ce comportement complexe suite à l'invocation de la méthode principale avec les paramètres suivants :

```
(Sweep_zone rover0 waypoint1 waypoint2 10)
```

Il est important de remarquer que, de la manière dont il est décrit, ce balayage n'est pas possible si la zone contient un obstacle. En effet, si un parcours en ligne droite ou un demi-tour n'est pas possible, alors tout le balayage est impossible. Un comportement de balayage tenant compte de la topologie de la zone à couvrir peut être facilement spécifié en proposant des actions alternatives aux actions impossibles.

Par ailleurs, sur la figure 5.22, on remarque que la position du robot à la fin du balayage est la position du deuxième objet. Ceci est fortuit et est du au fait que la division de la distance horizontale entre les deux objets par le diamètre du demi-cercle à pour résultat une valeur entière.

5.2.5 Mission 1 : analyse et collecte d'échantillons

Dans les paragraphes précédents, nous avons vu comment spécifier des comportements géométriques particuliers. Ces comportements permettent d'exprimer la manière de réaliser des actions du point de vue des déplacements du

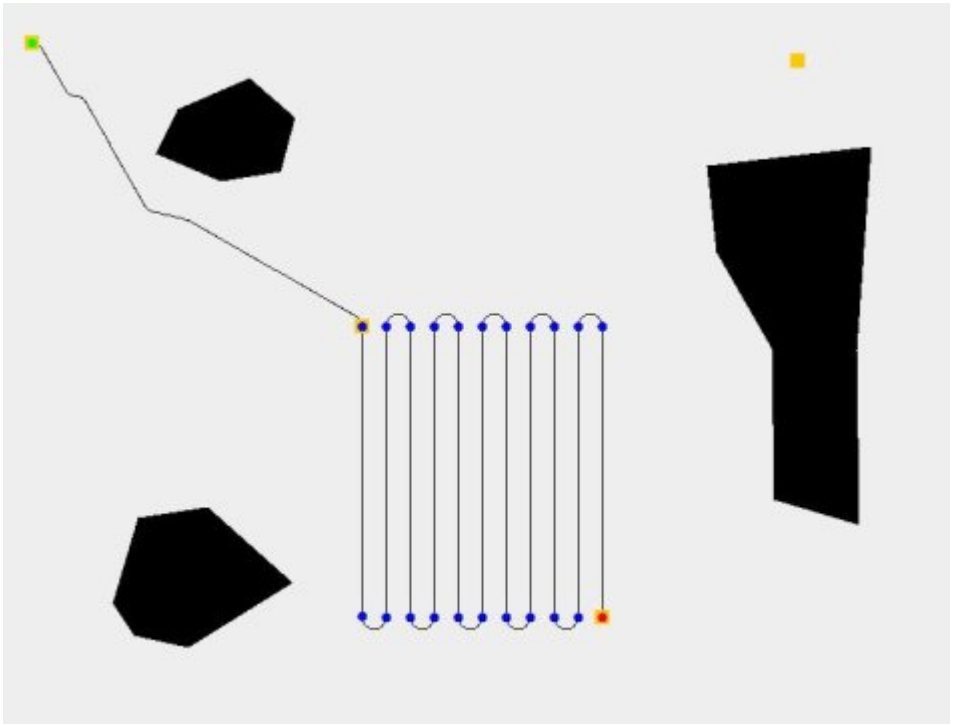


FIGURE 5.22 – Balayage d'une zone de l'environnement

robot. Dans ce paragraphe, nous présentons la formalisation d'une mission complète pour un robot mobile ainsi qu'une solution obtenue par l'architecture de planification hybride.

La mission est la suivante : un rover doit collecter un ensemble d'échantillons de roche dans l'environnement. Les échantillons de roche sont situés à des endroits précis de l'environnement, représentés par des objets. Le rover n'a pas connaissance de la quantité d'échantillons disponible pour chaque objet. Il doit, préalablement à une collecte, effectuer une analyse du sol afin de connaitre cette quantité. Le rover dispose d'un espace de stockage limité. Lorsque l'espace de stockage devient insuffisant, il doit rentrer au dépot central afin de décharger son stock. Si à la fin d'une collecte, il dispose de suffisamment de stockage libre, il peut effectuer un nouveau prélèvement avant de rentrer au dépot central. Lors d'une analyse de sol, le robot effectue des cercles concentriques de rayon de plus en plus grand autour de l'objet. L'objectif de la mission est que la quantité de roche prélevée et déchargée au dépot central soit d'au moins 30 kilos.

Définition du domaine de planification

La méthode `analyse_soil` décrit la séquence des actions nécessaires pour obtenir une analyse du sol sur un objectif passé en paramètre. Elle fait appel récursivement à l'opérateur `!analyse_soil` et lui transmet le numéro du

cercle en cours. La méthode s'arrête lorsque l'opérateur renvoie un échec, *i.e.,* lorsqu'un cercle n'est pas possible. Dans ce cas, elle appelle l'opérateur `!!compute_rock_qty` qui produit un prédicat stipulant la quantité de roche disponible pour l'objectif courant.

```
(Method (analyse_soil ?r ?wp ?n)
 ((max_analyse ?qn)(call < ?n ?qn))
 ((!analyse_soil ?r ?wp ?n)
  (analyse_soil ?r ?wp (call + ?n 1)))
 ((call > ?n 0))
 ((!!compute_rock_qty ?wp ?n))
)
```

L'opérateur `!analyse_soil` précise le comportement du robot durant une phase d'analyse. Celui-ci doit effectuer un cercle autour de l'objectif. Le rayon du cercle est calculé à partir du numéro de la phase d'analyse et du rayon du premier cercle donné par le prédicat `(analyse_radius ?rad)`. Dans la spécification de cette mission, cette valeur est fixée à 15. Ainsi, le deuxième cercle aura pour rayon 30, *etc.*

```
(Operator (!analyse_soil ?r ?wp ?n)
 ((analyse_radius ?rad)
  (assign ?radius (call * (call + ?n 1) ?rad)))
 ((agent ?r)(object ?wp)
  (distance(?r, ?wp) = ?radius)
  (rel_angle(?r, ?wp) = cos-and-sin(1.5708)))
 ((until(distance, mult(6.28318, ?radius)))
  (constant(distance(?r, ?wp))))
 ()()
)
```

La méthode `collect_rock` est chargée de calculer la quantité de roche que doit collecter le rover en fonction de la quantité disponible pour l'objectif passé en paramètre ainsi que de la quantité actuelle transportée. Si le rover dispose de suffisammenent d'espace disponible par rapport à la quantité à prélever, alors toute la roche est collectée. Sinon, la quantité à collecter correspond à la quantité maximale que peut transporter le rover connaissant son stock actuel.

```
(Method (collect_rock ?r ?wp)
 ((at_rock_sample ?wp ?qs)(call > ?qs 0)(store_level ?r ?qr)
  (max_store ?r ?qm)(assign ?qrfree (call - ?qm ?qr))
  (call > ?qrfree 0)(call >= ?qs ?qrfree))
 ((!collect_rock ?r ?wp ?qrfree))
 ((at_rock_sample ?wp ?qs)(call > ?qs 0)(store_level ?r ?qr)
  (max_store ?r ?qm)(assign ?qrfree (call - ?qm ?qr))
```

```
  (call > ?qrfree 0)(call < ?qs ?qrfree))
 ((!collect_rock ?r ?wp ?qs))
)
```

Pour effectuer une collecte, le robot doit se positionner à moins de 5 mètres de l'objectif et être face à lui. Si le déplacement est possible, alors l'opérateur met à jour la quantité restante après prélèvement au niveau de l'objectif ainsi que la quantité qu'il est en train de transporter.

```
(Operator (!collect_rock ?r ?wp ?q)
 ((at_rock_sample ?wp ?qs)(store_level ?r ?qr))
 ((agent ?r)(object ?wp)
  (distance(?r, ?wp) < 5)
  (rel_angle(?r, ?wp) = cos-and-sin(0)))
 ()()
 ((not(at_rock_sample ?wp ?qs))(at_rock_sample ?wp (call - ?qs ?q))
  (not(store_level ?r ?qr))(store_level ?r (call + ?qr ?q)))
)
```

Afin de pouvoir décharger son stock, le rover doit se rendre au dépot central en se positionnant sur l'objet de l'environnement représentant le dépot. Le résultat de l'opérateur `!empty_store` est une augmentation de la quantité disponible au dépot de exactement la quantité transportée et une remise à zéro de l'espace de stockage du rover.

```
(Operator (!empty_store ?r ?wp)
 ((store_level ?r ?qr)(central_rock_level ?q))
 ((agent ?r)(object ?wp)
  (distance(?r, ?wp) = 0))
 ()()
 ((not(central_rock_level ?q))(central_rock_level (call + ?q ?qr))
  (not(store_level ?r ?qr))(store_level ?r 0))
)
```

Le choix entre collecter un échantillon sur un objectif ou analyser la quantité disponible pour cet objectif est spécifiée par la méthode `collect_or_analyse_wp`. Si l'objectif a déjà été analysé et que la quantité disponible est supérieure à 0, alors le rover doit collecter tout ou partie de cette quantité. Sinon, il doit effectuer une analyse afin de découvrir cette quantité.

```
(Method (collect_or_analyse_wp ?r ?wp)
 ((at_rock_sample ?wp ?q)(call > ?q 0))
 ((collect_rock ?r ?wp))
 ((not(analysed ?wp)))
 ((analyse_soil ?r ?wp 0))
)
```

Le choix du prochain objectif à visiter est donné par la méthode `collect_or_analyse`. Cette méthode fait appel à l'expertise du module de raisonnement géométrique au travers d'une demande de conseil pour connaitre l'objectif le plus proche de la position actuelle du rover. Cette méthode, associée à la méthode de choix de l'action à entreprendre sur un objectif donné, fait que le rover va chercher, en priorité, à collecter toute la roche disponible sur les objectifs les plus proches du dépot central.

```
(Method (collect_or_analyse ?r)
 (:sort-by ?d < ((waypoint ?wp)(not(unaccessible ?wp))
   (heuristic(distance_from_waypoint ?r ?wp ?d))))
 ((verify_accessibility ?r ?wp))
)
```

La méthode principale du domaine de planification est la méthode `do_collect`. C'est une méthode récursive qui vérifie dans un premier temps si la quantité déjà collectée est suffisante pour satisfaire les objectifs de la mission. Puis, dans un deuxième temps, elle vérifie l'espace de stockage du rover. Si le robot est plein, alors il doit alors déposer son chargement au dépot central. Dans le cas contraire, la méthode principale appelle la méthode de choix de l'objet de l'environnement sur lequel le rover doit analyser et collecter.

```
(Method (do_collect ?r)
 ((has_enough_rock))
 ()
 ((is_full_store ?r)(central_waypoint ?wp))
 ((!empty_store ?r ?wp)(do_collect ?r))
 ()
 ((collect_or_analyse ?r)(do_collect ?r))
)
```

L'ensemble des opérateurs et méthodes du domaine de planification associé à cette mission est présenté en annexe B.2.

Problème de planification et résultats obtenus

Pour satisfaire la mission, nous spécifions l'état initial suivant :

```
(analyse_radius 15)(max_analyse 5)(rock_quantity 5)
(mission_rock_qty 30)(central_rock_level 0)
(rover rover0)(store_level rover0 0)(max_store rover0 20)
(central_waypoint central)
(waypoint locA)(waypoint locB)(waypoint locC)(waypoint locD)
(waypoint locE)(waypoint locF)(waypoint locG)
```

L'environnement contient 7 objets identifiant les zones contenant de la roche (locA à locG). Lors de l'analyse d'un objectif, le robot peut effectuer au maximum 5 phases d'analyse. Le rayon de la phase d'analyse initiale est fixé à 15 mètres. Chaque phase d'analyse permet de découvrir 5 kilos de roche. L'espace maximal de stockage du robot est de 20 kilos et est vide en début de mission. La quantité globale à récolter est de 30 kilos de roche.

Le but de l'architecture de planification est de trouver une solution pour le problème de planification formulé de la manière suivante :

```
(!init_rover rover0)
(do_collect rover0)
```

Un plan solution est obtenu si l'architecture de planification peut appliquer l'opérateur `!init_rover` et décomposer la méthode `do_collect` jusqu'à obtenir un ensemble d'actions primitives.

La figure 5.23 illustre les déplacements du robot durant la mission. Ces déplacements sont associés au plan symbolique suivant [2] :

```
(!init_rover rover0 )
(!analyse_soil rover0 locC 0 )
(!analyse_soil rover0 locC 1.0 )
(!collect_rock rover0 locC 10.0 )
(!analyse_soil rover0 locB 0 )
(!collect_rock rover0 locB 5.0 )
(!empty_store rover0 central )
(!analyse_soil rover0 locA 0 )
(!collect_rock rover0 locA 5.0 )
(!analyse_soil rover0 locE 0 )
(!analyse_soil rover0 locE 1.0 )
(!analyse_soil rover0 locE 2.0 )
(!collect_rock rover0 locE 15.0 )
(!empty_store rover0 central )
```

Ce plan a été obtenu en 8 secondes [3] et a nécessité l'envoi de 30 requêtes de planification de déplacement. La distance parcourue par le robot pour accomplir la mission est de 4294 mètres.

Remarques

Le scénario de mission proposé met en avant l'intérêt de l'utilisation de notre architecture de planification hybride concernant trois points :

2. Seule les actions correspondant à une action physique du robot sont représentées dans le plan.

3. Les problèmes de planification ont été résolus sur un PC équipé d'un processeur Intel Core 2 duo 6300 et de 2Go de mémoire RAM sous le système d'exploitation Microsoft XP.

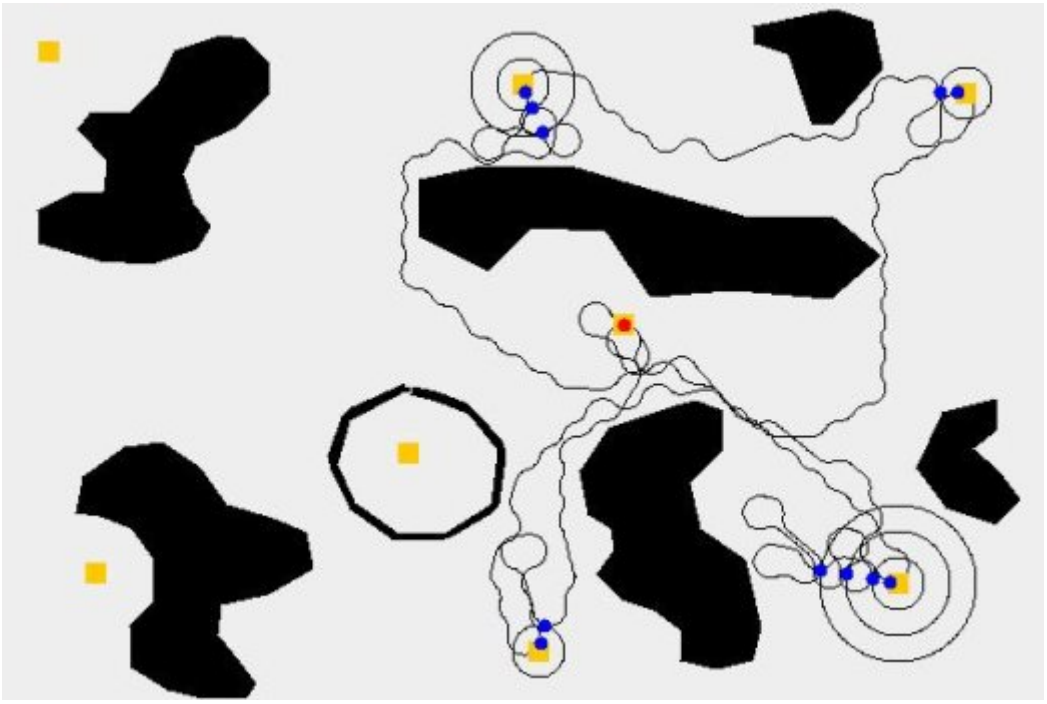


FIGURE 5.23 – Solution trouvée pour la mission 1

Tout d'abord, comme les quantités de roche disponibles pour chaque objectif ne sont pas connues à l'avance mais nécessitent des analyses du sol, l'utilisation d'une architecture classique de planification impliquerait une analyse préalable de l'environnement afin de déterminer ces quantités. La connaissance des différentes quantités est nécessaire car, d'une part, l'objectif de la mission est chiffré et, d'autre part, la quantité que peut stocker le robot est limitée et a donc un impact sur le choix des actions.

La zone représentée par l'objet locF n'est pas accessible par le robot. Ainsi, tout plan impliquant des actions du robot sur cette zone serait invalidé durant le calcul des mouvements. Une analyse préalable de l'environnement serait alors nécessaire afin de définir tous les déplacements possibles du robot entre les différentes zones. Au vue de la carte de l'environnement représentée sur la figure 5.23, il semble évident que l'objet locF n'est pas accessible. Cependant, dans certains cas, seule une analyse tenant compte du modèle cinématique du robot permettrait de connaitre ces zones inaccessibles.

Enfin, les actions que peut effectuer le rover nécessitent des comportements particuliers. Ces comportements sont contraints, d'une part, par les préconditions géométriques et, d'autre part, par la topologie de l'environnement. Il est donc nécessaire de calculer les déplacements correspondant à une action avant de savoir si celle-ci peut être effectivement réalisée et donc planifiée.

5.2.6 Mission 2 : collecte sous contrainte d'énergie

La deuxième mission que nous proposons se situe dans le même contexte et dans le même environnement que la mission précédente. Le rover doit collecter un ensemble d'échantillons de roche. Cependant les quantités disponibles pour chaque objectif sont connues à l'avance. Le but de la mission est d'effectuer au moins un prélèvement sur chaque objectif en respectant des contraintes sur l'énergie du rover. En effet, le rover est initialisé avec une quantité donnée d'énergie. Chaque déplacement et chaque prélèvement de roche nécessite une certaine quantité d'énergie. Le rover n'a pas la possibilité de recharger sa batterie. Dans l'hypothèse où le robot disposerait encore suffisamment d'énergie après avoir visité chacun des objectifs, il peut alors chercher à maximiser la quantité de roche collectée en effectuant de nouveaux prélèvements.

Définition du domaine de planification

Lors de l'initialisation, en plus de spécifier la position initiale et le cap du rover, l'opérateur `!init_rover` initialise le niveau d'énergie du robot à la valeur spécifiée par le prédicat `(total_energy ?q)` de l'état initial.

```
(Operator (!init_rover ?r)
 ((rover ?r)(central_waypoint ?o)(total_energy ?q)
  (not(initialized ?r)))
 (
  (agent ?r)(object ?o)
  (setProperty(?r.x, ?o.x))(setProperty(?r.y, ?o.y))
  (setProperty(?r.heading, 0))(setProperty(?r.energy_level, ?q))
   )
 ()()
 ((initialized ?r))
 )
```

La méthode décrivant la collecte d'échantillons sur un objectif est la même que celle définie pour la mission précédente. L'opérateur `!collect_rock` décrit, en plus, la manière dont est consommée l'énergie lors d'une action de collecte. Cette consommation est la somme de l'énergie nécessaire au prélèvement, somme qui dépend de la quantité collectée, et de l'énergie nécessaire aux déplacements du rover. L'énergie consommée par le rover durant les déplacements est récupérée par le planificateur de tâches au niveau des effets géométriques.

```
(Operator (!collect_rock ?r ?wp ?q)
 ((at_rock_sample ?wp ?qs)(store_level ?r ?qr)(needed_energy ?tot)
  (energy_sample_unit ?qe)(assign ?energy_sample (call * ?q ?qe)))
 ((agent ?r)(object ?wp)
  (distance(?r, ?wp) < 5)
```

```
  (rel_angle(?r, ?wp) = cos-and-sin(0)))
 ()
 ((conso_energy r ?energy_travel))
 ((not(at_rock_sample ?wp ?qs))(at_rock_sample ?wp (call - ?qs ?q))
  (not(store_level ?r ?qr))(store_level ?r (call + ?qr ?q))
  (not(needed_energy ?tot))
  (needed_energy (call + ?energy_sample ?energy_travel))
  (has_sample_of ?wp))
)
```

L'opérateur `!empty_store` défini précédemment est également modifié dans ce sens. Après avoir planifié une action consommant de l'énergie, le planificateur de tâches appelle l'opérateur `!update_energy_level` qui met à jour la propriété de concept relative au niveau d'énergie du robot à l'aide d'une commande d'affectation directe. Cette mise à jour est nécessaire car le planificateur de déplacements que nous avons développé tient compte du niveau d'énergie et de la consommation du robot lors du calcul d'un déplacement.

```
(Operator (!update_energy_level ?r)
 ((total_energy ?tot)(needed_energy ?q)
  (assign ?new_tot (call - ?tot ?q)))
 ((agent ?r)
  (setProperty(?r.energy_level, ?new_tot)))
 ()()
 ((not(total_energy ?tot))(total_energy ?new_tot))
)
```

La méthode chargée de décrire la manière d'atteindre l'objectif de la mission, *i.e.,* d'effectuer au moins un prélèvement au niveau de chaque objet de l'environnement, est la méthode `choose_and_collect`. Elle vérifie dans un premier temps le niveau du stock du rover et l'envoie décharger son stock au dépot central si besoin. Puis elle choisit un objectif sur lequel sera faite la prochaine collecte. Ce choix dépendant de la distance entre les objets de l'environnement et la position actuelle du rover. Cependant, cette méthode ne favorise pas l'exploration de l'environnement mais la maximisation de la quantité de roche collectée. Ainsi, si l'objectif le plus proche a déjà été visité mais dispose encore de roche, alors il sera choisi en priorité. Si l'énergie nécessaire pour accomplir l'ensemble de la mission n'est pas suffisant, ces actions de collecte sur des objectifs déjà visités sont annulées lors du retour arrière du planificateur de tâches. Finalement, la méthode se termine lorsque toutes les zones contenant des échantillons de roche ont été visitées au moins une fois, qu'il ne reste pas assez d'énergie pour effectuer une nouvelle action et que le robot se trouve au dépot central.

```
(Method (choose_and_collect ?r)
 ((is_full_store ?r)(central_waypoint ?wp))
```

```
 ((!empty_store ?r ?wp)(!update_energy_level ?r)(choose_and_collect ?r))
 (:sort-by ?d < ( (at_rock_sample ?wp ?qs) (call > ?qs 0)
  (heuristic(distance_from_waypoint ?r ?wp ?d))))
 ((collect_rock ?r ?wp)(!update_energy_level ?r)
  (!!remove (goal(need_sample_of ?wp)))(choose_and_collect ?r) )
 ((not(goal(need_sample_of ?loc)))(central_waypoint ?wp)
  (store_level ?r ?ql)(call > ?ql 0))
 ((!empty_store ?r ?wp)(!update_energy_level ?r))
 ((not(goal(need_sample_of ?loc)))(central_waypoint ?wp)
  (store_level ?r ?ql)(call = ?ql 0))
 ()
)
```

La méthode principale est la méthode `do_mission` qui a pour objectif d'initialiser le problème en créant un ensemble de prédicats de la forme `(goal(need_sample_of ?loc))` où `?loc` est un objet de l'environnement, puis d'appeller la méthode récursive `choose_and_collect`.

Le domaine complet associé à ce type de mission est présenté en annexe B.3.

Résultats obtenus

Pour ce scénario, l'état initial est le suivant :

```
(total_energy 10000)(energy_sample_unit 100)(needed_energy 0)
(rover rover0)(store_level rover0 0)(max_store rover0 10)
(central_waypoint central)(central_rock_level 0)
(waypoint locA)(waypoint locB)(waypoint locC)
(waypoint locD)(waypoint locE)(waypoint locG)
(at_rock_sample locA 17)(at_rock_sample locB 2)(at_rock_sample locC 15)
(at_rock_sample locD 12)(at_rock_sample locE 18)(at_rock_sample locG 6)
```

Le rover dispose d'une quantité d'énergie fixée initialement à 10000. La taille maximale de son chargement est fixée à 10 kilos. La collecte d'un kilo de roche nécessite 100 unités d'énergie. L'énergie nécessaire aux déplacements du rover n'est pas défini dans l'état initial mais au niveau du modèle du robot et est d' une unité d'énergie par mètre parcouru. Dans ce scénario, nous avons enlevé l'étiquette symbolique correspondant à l'objet locF car, celui-ci étant inaccessible, la mission ne serait pas réalisable.

Le problème de planification est spécifié de la manière suivante :

```
(!init_rover rover0)
(do_mission (list
 (need_sample_of locA) (need_sample_of locB) (need_sample_of locC)
 (need_sample_of locD) (need_sample_of locE) (need_sample_of locG)
))
```

Le plan solution obtenu et illustré par la figure 5.24 est le suivant :

```
(!init_rover rover0 )
(!collect_rock rover0 locC 10.0 )
(!update_energy_level rover0 )
(!empty_store rover0 central )
(!update_energy_level rover0 )
(!collect_rock rover0 locC 5.0 )
(!update_energy_level rover0 )
(!collect_rock rover0 locB 2 )
(!update_energy_level rover0 )
(!collect_rock rover0 locE 3.0 )
(!update_energy_level rover0 )
(!empty_store rover0 central )
(!update_energy_level rover0 )
(!collect_rock rover0 locA 10.0 )
(!update_energy_level rover0 )
(!empty_store rover0 central )
(!update_energy_level rover0 )
(!collect_rock rover0 locA 7.0 )
(!update_energy_level rover0 )
(!collect_rock rover0 locE 3.0 )
(!update_energy_level rover0 )
(!empty_store rover0 central )
(!update_energy_level rover0 )
(!collect_rock rover0 locG 6 )
(!update_energy_level rover0 )
(!collect_rock rover0 locD 4.0 )
(!update_energy_level rover0 )
(!empty_store rover0 central )
(!update_energy_level rover0 )
```

Ce plan solution a été obtenu en 48 secondes et a nécessité l'envoi de 76 requêtes de planification. La quantité de roche collectée par le rover est de 50 kilos et la distance parcourue de 4849 mètres. L'énergie restante à la fin de la mission est de 150.5 unités. On peut remarquer que ce plan, d'une part, respecte l'objectif de la mission, *i.e.,* toutes les zones ont été visitées et, d'autre part qu'il optimise les distances parcourues et donc l'énergie du robot. En effet, on remarque que locB puis locE sont visitées directement après locC car l'espace de stockage du rover n'est pas plein. Puis après avoir déchargé, le rover se rend à locA qui est la zone ayant encore de la roche disponible la plus proche du dépot central, avant de retourner prélever des échantillons sur la zone locE.

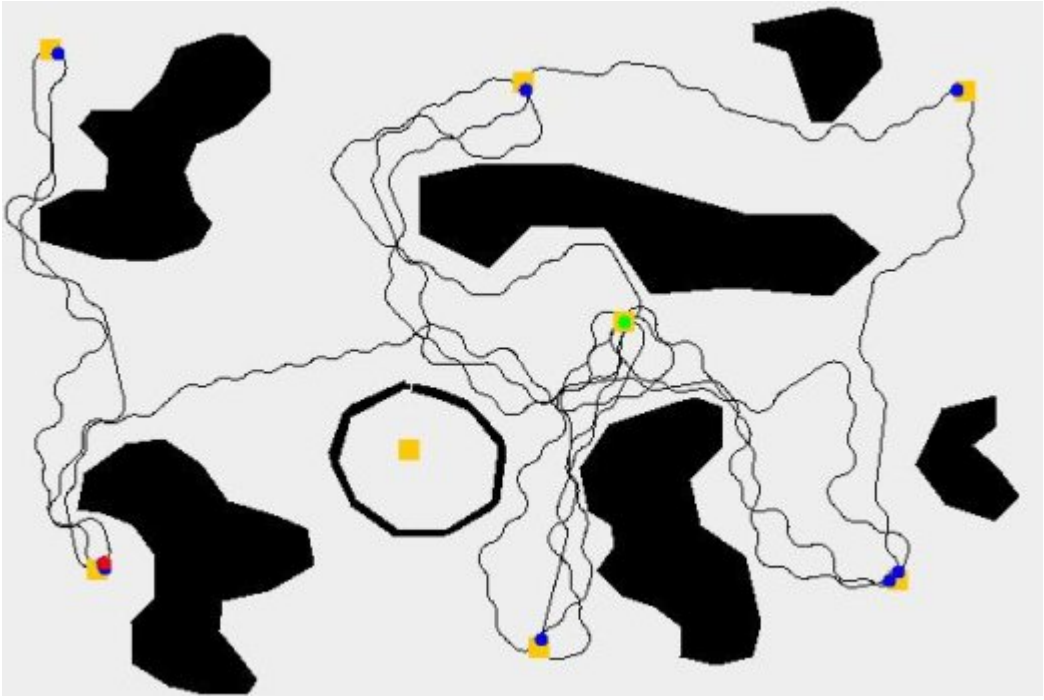


FIGURE 5.24 – Solution trouvée pour la mission 2

Remarques

En plus de devoir calculer les chemins possibles entre les différentes zones d'action, une architecture de planification classique doit estimer l'énergie nécessaire au robot pour la réalisation des déplacements et des actions de collecte. En effet, dans cette mission, l'énergie consommée pour réaliser une action est fortement dépendante des distances parcourues par le robot.

Ainsi, pour optimiser au mieux la mission, il n'est pas suffisant de savoir s'il existe un déplacement entre deux zones d'action mais il est nécessaire de connaitre la quantité exacte d'énergie associée à chaque déplacement.

5.3 Conclusion

Dans ce chapitre, nous avons, dans un premier temps, déterminé l'influence des paramètres de configuration du planificateur CELL-RRT sur la qualité de la solution en termes de distance parcourue et de temps de calcul. Nous avons montré l'intérêt de la réduction de l'espace de recherche à un corridor permettant une réduction du temps de calcul d'au moins 75% pour une faible perte de qualité. Puis, nous avons étudié l'impact de l'ajustement des différents paramètres de configuration du planificateur sur ses performances et nous avons montré qu'il est possible d'ajuster ces paramètres selon que l'on cherche une solution de moindre longueur ou en temps limité.

Puis, dans un deuxième temps, nous avons illustré le fonctionnement de l'architecture de planification hybride proposée en détaillant les calculs effectués lors de la satisfaction de contraintes géométriques, en présentant l'encodage de quelques comportements particuliers et, enfin, en illustrant les résultats obtenus sur des scénarios de mission pour un robot mobile de type rover d'exploration.

Ces illustrations mettent en valeur, d'une part, le pouvoir d'expressivité du langage d'expression des actions à l'aide de préconditions et effets relatifs aux déplacements du robot et, d'autre part, l'intérêt de l'architecture de planification hybride proposée pour la résolution de problèmes de planification dans lesquels les actions symboliques et les déplacements du robot sont inter-dépendants.

Conclusion générale

Dans ce mémoire, nous avons abordé le problème de planification de missions pour un robot mobile. Une mission est définie par un ensemble d'actions que le robot doit exécuter et par un ensemble de déplacements permettant de réaliser ces actions.

L'architecture logicielle d'un robot est souvent caractérisée par une hiérarchie de couches correspondant à différents niveaux d'abstraction. La planification de tâches, permettant de définir les actions à réaliser, et la planification de déplacements ne sont généralement pas traitées au même niveau. Cependant, ces deux types de planifications sont liés par le fait que certaines actions peuvent nécessiter des déplacements pour leur réalisations.

L'objectif de nos travaux a été de définir un modèle de planification hybride intégrant un module de planification de tâches et un module de planification de déplacements afin de résoudre, de manière unifiée, les problèmes de planification de mission pour un robot mobile.

Contributions

Dans le second chapitre, nous avons mis en évidence, au travers d'un ensemble d'expérimentations, l'intérêt du couplage entre planification de tâches et planification de déplacements. Une exécution entrelacée des deux planificateurs permet une meilleure gestion des échecs de planification dus à des déplacements impossibles et ainsi réduit le temps de calcul d'un plan solution. De plus, un guidage du planificateur de tâches par des informations numériques produites par le planificateur de déplacements peut être bénéfique.

Suite à ces constatations, nous avons proposé un modèle de planification hybride dans lequel les déplacements sont définis au niveau symbolique par l'expression de préconditions et d'effets géométriques. Les préconditions permettent de définir, d'une part, l'attitude que doit avoir le robot en vue de la réalisation d'une action et, d'autre part, le comportement géométrique qu'il doit adopter durant la réalisation de cette action. Ces préconditions sont exprimées à l'aide d'une sémantique bien indentifiée et font intervenir des contraintes géomé-

triques. Les contraintes géométriques sont traduites en fonctions mathématiques qui sont satisfaites à l'aide d'un algorithme d'optimisation non linéaire restreignant les configurations possibles du robot. Les effets géométriques servent à transmettre des informations résultant du calcul des déplacements au planificateur de tâches. Ils permettent de partager des variables sur les ressources du robot et transmettre des références sur les configurations calculées. Les échanges entre les deux modules se font par des messages standardisés sous la forme de requêtes de planification, de réponses à ces requêtes et de conseils. Les requêtes de planification contiennent les contraintes géométriques spécifiant les déplacements à calculer. Les conseils ont pour but d'apporter l'expertise du module de raisonnement géométrique afin de guider le planificateur de tâches dans ses choix, de proposer des optimisations ou de permettre une réparation plus efficace du plan.

Nous avons également proposé et mis en œuvre le planificateur Cell-RRT qui combine une technique de réduction de l'espace de recherche à un corridor avec un algorithme de planification de déplacements par construction incrémentale d'arbres aléatoires. Les expérimentations présentées dans la première partie du chapitre 5 ont montré l'intérêt de notre approche en termes de temps de calcul et de qualité de la solution ainsi que les possibilités de configurations du planificateur permettant d'améliorer ces performances.

La mise en œuvre du modèle de planification hybride par couplage d'un planificateur HTN avec le planificateur de déplacements Cell-RRT a permis d'illustrer la faisabilité de notre proposition. Nous avons montré comment exprimer des comportements géométriques complexes au niveau de la définition symbolique des actions. Ces comportements font intervenir, d'une part, la spécificité de la planification hiérarchique par l'utilisation de méthodes et, d'autre part, par l'expression de préconditions géométriques particulières. Puis nous avons illustré le fonctionnement de l'architecture sur des exemples de mission pour un robot d'exploration.

Perspectives

Le modèle de planification hybride proposé n'a pas été mis en œuvre totalement. En effet, nous n'avons pas développé et testé les modules d'optimisation et de réparation du plan. L'intégration de ces modules impliquent de profondes modifications du planificateur HTN développé. En effet, la recherche de plan est représenté en mémoire sous la forme d'un arbre. La modification de cet arbre suite à l'inversion de deux actions à exécuter tout en respectant les liens causaux et les contraintes d'ordre n'est pas trivial. Des recherches sur ce thème seraient un prolongement naturel de cette thèse.

Par ailleurs, suite aux travaux que nous avons menés, plusieurs extensions peuvent être proposées parmi lesquelles : la gestion de mission faisant intervenir plusieurs agents, la prise en compte d'incertitude sur l'environnement, la replanification suite à l'apparition de nouveaux buts après exécution d'un plan calculé ou suite à un échec durant cette exécution, ainsi que certaines extensions du planificateur CELL-RRT.

Extension au cadre multi-agent

Le modèle de planification hybride a été proposé en tenant compte du possible contexte multi robots qui peut être inhérent à certains types de missions. Les préconditions géométriques spécifient, à l'aide de la notion de concept, l'agent intervenant lors d'un déplacement. La structure des messages échangés entre les deux modules de raisonnement est prévue pour spécifier l'agent concerné par ce message. Cependant, plusieurs mises en œuvre de l'architecture sont possibles :

- approche centralisée : les agents sont gérés par un planificateur de tâches et un seul module de raisonnement géométrique central ;
- approche semi-centralisée : le planificateur de tâches est commun aux différents agents mais chaque agent a son propre module de raisonnement géométrique ;
- approche semi-décentralisée : chaque agent correspond à un robot et dispose d'une architecture de planification hybride ;
- approche décentralisée : chaque agent est soit un planificateur de tâches, soit un planificateur de déplacements.

Ces différentes approches sont amenées à traiter, en plus de la planification, des problèmes d'affectation, d'anti-colision, d'absence de blocage et de synchronisation des actions.

Pour les approches centralisées et semi-centralisées, l'affectation est du ressort du planificateur de tâches et doit se faire par unification des variables relatives aux robots. Pour l'approche semi-décentralisée, l'affectation est extérieure à l'architecture de planification hybride et un processus de dialogue entre les architectures doit être défini. Finalement, pour l'approche décentralisée, l'affectation se fait au travers d'une interprétation différente des messages entre planificateur de tâches et planificateur de déplacements. Le jeu de messages proposé ici doit alors être enrichi pour, par exemple, faire intervenir des notions d'engagement et de contrat.

En ce qui concerne l'anti-colision et l'absence de blocage, deux remarques peuvent être faites. La première est que la prise en compte des aspects temporels doit être plus complète au niveau du planificateur de déplacements pour

pouvoir obtenir ces fonctions. La seconde est que, mis à part pour l'approche centralisée, l'échange de messages entre planificateurs de déplacements concernant l'occupation de l'espace par les robots doit être défini.

La synchronisation des actions de différents robots demande également une prise en compte des aspects temporels plus explicite. Il est possible que ces contraintes supplémentaires ne concernent pas uniquement les déplacements mais doivent également être exprimées au niveau des tâches. Finalement, la mise en œuvre de ces synchronisations semble plus facile pour les approches centralisées et semi-centralisées que pour les approches distribuées et semi-distribuées.

Prise en compte des incertitudes

Tout au long de ces travaux, nous nous sommes placés dans un contexte déterministe : l'environnement est totalement connu et le résultat des actions et des déplacements est certain. Cependant, en condition réelle, ces hypothèses de travail ne sont pas forcément valides.

La conséquence de la présence d'une incertitude au niveau de l'environnement est que le résultat d'une requête de déplacement n'est plus une réponse booléenne indiquant si le déplacement est possible ou non, mais une valeur numérique indiquant sa probabilité de réussite. Le traitement de ce type de réponse par le planificateur de tâches est un problème ouvert. Par ailleurs, au niveau du planificateur de tâches, le problème de planification doit être reformulé de manière à ce que sa résolution par chaînage avant reste réaliste. On peut penser à une recherche de plan maximisant la probabilité des buts donnés.

Replanification

Si de nouveaux buts sont ajoutés ou si un échec survient en cours d'exécution du plan, l'architecture de planification hybride doit mettre en œuvre un mécanisme de replanification.

Du point de vue des déplacements, la replanification peut être facilitée par la possibilité de réutilisation des arbres et segments de chemins proposée par le planificateur Cell-RRT. Au niveau des actions, dans le cas dans échec, il faut identifier les branches de l'arbre de recherche remises en cause et proposer des alternatives.

Les deux principaux points d'étude dans le cas d'une replanification sont, d'une part, la cohérence entre ces alternatives et les actions et déplacements existants et, d'autre part, la vérification de la validité des ressources partagées.

Extensions de Cell-RRT

Cell-RRT a été conçu pour la gestion des déplacements d'un robot dans un environnement spécifié en deux dimensions. Une amélioration possible est l'extension du planificateur pour la planification de mouvements en trois dimensions. La prise en compte de l'altitude demande une réflexion sur la décomposition de l'environnement en cellules ainsi que sur la définition des points de passage entre ces cellules. Plusieurs pistes sont envisageables :

- un découpage régulier de l'environnement selon l'altitude ;
- l'environnement est décomposé en 2D et l'altitude est une variable à optimiser.

Une deuxième perspective d'extension de Cell-RRT concerne la réponse à la question suivante : est-il possible de définir automatiquement la granularité de la décomposition en fonction de la topologie de l'environnement ? En effet, dans nos travaux, la granularité de cette décomposition est fixée par l'utilisateur. Cependant, dans l'hypothèse où l'environnement n'est pas totalement connu avant le processus de planification, une méthode permettant de définir automatiquement le découpage, tout en assurant de bonnes performances, serait souhaitable.

Annexe A

Algorithmes de recherche de chemin

Les problèmes de chemin sont, en théorie des graphes, des problèmes qui visent à calculer une route entre les sommets d'un graphe qui minimise ou maximise une certaine fonction. Lorsque l'algorithme cherche à minimiser une fonction de coût, on parle d'algorithme de plus court chemin.

Le problème du recherche du plus court chemin dans un graphe peut être formalisé de la manière suivante :

Définition. A.1 (Recherche du plus court chemin)
Soit G un graphe pondéré $G : \langle V, E, l : E \rightarrow R \rangle$, trouver un chemin β du sommet v au sommet v' tel que $\sum l(e), e \in \beta$ soit minimale.

Dans cette annexe, nous présentons deux algorithmes de recherche de chemin qui sont utilisés dans le cadre de ces travaux : l'algorithme A* et l'algorithme de Dijkstra.

A.1 Algorithme A*

L'algorithme A* [Hart *et al.*, 1968] est un algorithme qui utilise une heuristique pour fournir rapidement un chemin proche de l'optimal. Il calcule une évaluation heuristique sur chaque nœud pour estimer le meilleur chemin, puis visiter ces nœuds dans l'ordre de l'évaluation heuristique jusqu'à obtenir un chemin solution entre le nœud initial et le nœud final.

Cette heuristique permet d'estimer la distance restante entre le noeud courant et le nœud de destination. Si l'heuristique utilisée est dite admissible et monotone, alors l'algorithme A* renvoie une solution optimale.

Définition. A.2 (Heuristique admissible)
Une heuristique est dite admissible si elle ne surestime jamais la distance au but : soit h une heuristique admissible et d une fonction de distance,

$$h(x) \leq d(x, y)$$

Définition. A.3 (Heuristique monotone)
Une heuristique est dite monotone si elle ne décroit jamais, *i.e.,* le coût estimé pour un nœud n est inférieur au coût estimé pour un nœud successeur de n. Soit h une heuristique admissible et monotone et d une fonction de distance,

$$h(x) \leq d(x, y) + h(y)$$

Algorithme

L'algorithme A.1 présente une implémentation possible pour A*.

À l'initialisation, le coût du nœud de départ n_{start} est calculé (lignes 1 à 3). Ce coût, représenté par la fonction f, est composé du coût g représentant la distance déjà parcourue (0 pour le premier nœud) et de l'estimation heuristique h de la distance au nœud but. L'algorithme utilise deux listes : la liste *openList* représentant les nœuds pendants et la liste *closedList* contenant les nœuds déjà visités.

Tant que la liste des nœuds pendants n'est pas vide (ligne 6) ou que le nœud destination n'a pas été atteint (ligne 10), l'algorithme choisit dans la liste des nœuds pendants le nœud ayant le coût le plus faible (ligne 7) et cherche tous les nœud adjacents à celui-ci dans le graphe. Pour chacun de ces nœuds, le coût est calculé et le nœud est ajouté à la liste des nœuds pendants (ligne 19). S'il appartenait déjà à cette liste mais que le nouveau coût est inférieur à la valeur précédente (ligne 23), alors le poids du nœud est mis à jour (ligne 25).

Si le nœud destination est atteint, l'algorithme renvoie le chemin solution (ligne 11) reconstitué par chainage arrière des nœuds de moindre coût appartenant à la liste des nœuds visités.

Variantes de A*

L'algorithme A* est un algorithme de recherche de chemin performant en termes de temps de calcul et d'espace mémoire utilisé. Cependant ses performances évoluent avec la taille du graphe. De plus, il n'est pas adapté aux environnements dynamiques, *i.e.,* pour lesquels les coûts de transition entre nœuds

Algorithme A.1 AStar(n_{start}, n_{goal})

```
 1: g(n_start) ← 0 ;
 2: h(n_start) ← heuristic_distance(n_start, n_goal) ;
 3: f(n_start) ← g(n_start) + h(n_start) ;
 4: openList ← {n_start} ;
 5: closedList ← ∅ ;
 6: while (openList ≠ ∅) do
 7:    n_current ← getBestNode(openList) ;
 8:    closedList ← closedList + n_current ;
 9:    openList ← closedList − n_current ;
10:    if (n_current = n_goal) then
11:       return retrievePath(n_start, n_goal) ;
12:    end if
13:    for each (n ∈ getAdjacentNodes(n_current)) do
14:       if (n ∉ closedList) then
15:          if (n ∉ openList) then
16:             g(n) ← g(n_current) + distance(n_current, n) ;
17:             h(n) ← heuristic_distance(n, n_goal) ;
18:             f(n) ← g(n) + h(h) ;
19:             openList ← openList + n ;
20:          else
21:             old_f ← f(openList(n)) ;
22:             new_g ← g(n_current) + distance(n_current, n) ;
23:             if (new_g < g(n)) then
24:                g(n) ← new_g ;
25:                f(n) ← g(n) + h(h) ;
26:             end if
27:          end if
28:       end if
29:    end for
30: end while
31: return null ;
```

peuvent évoluer au cours de la planification. Ainsi de nombreuses variantes ont été proposées.

Pour la prise en compte des graphes de grandes tailles, l'algorithme *Hierarchical Path-Finding* A* (HPA*) [Botea *et al.*, 2004] est une variante hiérarchique de A*. Cette technique utilise une abstraction de l'environnement sous forme de clusters reliés entre eux. Au niveau local, les chemins permettant de traverser chaque cluster sont calculés. Au niveau global, l'algorithme A* est appliqué afin de déterminer les clusters traversés par la solution finale.

Les variantes les plus célèbres pour la prise en compte des environnements partiellement connus ou dynamiques sont D* [Stentz, 1994] et *Focussed* D* [Stentz, 1995]. Lorsque l'environnement change, ces algorithmes recalculent une solution à partir de la position courante et non du nœud de départ. Une autre variante d'algorithme incrémental s'appuyant sur A* est *Lifelong Planning* A* (LPA*) [Koenig *et al.*, 2004].

A.2 Algorithme de Dijkstra

L'algorithme de Dijkstra [Dijkstra, 1959] est un algorithme de recherche du plus court chemin fonctionnant sur le principe des algorithmes de recherche en largeur d'abord : les nœuds du graphe sont évalués dans l'ordre de leur coût qui est fonction de leur distance au nœud origine.

L'algorithme de Dijkstra correspond à l'algorithme A* avec pour heuristique $\mathrm{h}(n) = 0$.

Algorithme

L'algorithme A.2 présente une implémentation possible de l'algorithme de Dijkstra.

L'algorithme initialise les nœuds du graphe avec un coût infini, signifiant ainsi que le nœud n'a pas encore été visité (ligne 2). Le nœud initial à pour coût 0. Puis, tant qu'il reste des nœuds à visiter, l'algorithme cherche le nœud de moindre coût (ligne 8). À partir de ce nœud, il recalcule le coût de chacun des nœuds adjacents (ligne 15). Ce coût correspond à la distance entre le nœud courant et le nœud initial plus la distance du nœud courant au nœud adjacent. Si le coût ainsi calculé est meilleur que le coût précédent alors le nœud est mis à jour (ligne 17 et 18). L'algorithme se termine lorsque le nœud de destination n_{goal} est atteint.

Le chemin est obtenu par chainage arrière à partir du nœud de destination.

Algorithme A.2 Dijkstra(n_{start}, n_{goal})

```
1: for each (Node n) do
2:    cost(n) ← ∞ ;
3:    root(n) ← null ;
4: end for
5: openList ← ensemble des nœuds ;
6: cost(n_start) ← 0 ;
7: while (openList ≠ ∅) do
8:    n_current ← getBestNode(openList) ;
9:    openList ← closedList − n_current ;
10:   if (n_current = n_goal) then
11:      return retrievePath(n_start, n_goal) ;
12:   end if
13:   if (cost(n_current) ≠ ∞) then
14:      for each (n ∈ getAdjacentNodes(n_current)) do
15:         newCost ← cost(n_current) + distance(n_current, n) ;
16:         if (newCost < cost(n)) then
17:            cost(n) ← newCost ;
18:            root(n) ← n_current ;
19:
20:         end if
21:      end for
22:   end if
23: end while
```

Encodage de l'algorithme en langage HTN

Durant les expérimentations présentées au chapitre 2, nous avons encodé l'algorithme de Dijkstra dans le formalisme HTN. Cet encodage est le suivant :

```
(Method (find_path ?wpo ?wpd)

    ((fini))
    ((mark_path ?wpo ?wpd))

    (:sort-by ?d < ((pendant ?wp ?wpp ?d)))
    ((develop ?r ?wp ?wpd)(find_path ?wpo ?wpd))

    ((not(init)))
    ((!!init ?wpo)(find_path ?wpo ?wpd))
)

(Method (develop ?wp ?wpd)

    ((same ?wp ?wpd))
    ((!!set_fini))

    ()
    ((!!develop ?wp))

)

(Operator (!!develop ?wp)
    ((pendant ?wp ?wpere ?f))
    (
      (forall (list ?ws)
        ((visible ?wp ?ws ?d)(not(developpe ?ws ?wp ?x)))
        ((pendant ?ws ?wp (call + ?d ?f) ))
      )
      (forall (list ?ws)
        ((pendant ?ws ?wpp1 ?d1)(pendant ?ws ?wpp2 ?d2)
        (call < ?d1 ?d2)) ((not(pendant ?ws ?wpp2 ?d2)))
      )
      (not(pendant ?wp ?wpere ?f))(developpe ?wp ?wpere ?f)
    )
)

(Operator (!!init ?wp)
    ()
    ((pendant ?wp nil 0)(init))
)
```

```
(Operator (!!set_fini)
    ()
    ((fini))
)

(Method (mark_path ?wpo ?wpd)

    ((fini)(pendant ?wpd ?wpere ?d))
    ((!!mark_to_dest ?wpd ?wpere)(mark_path ?wpo ?wpere))

    ((same ?wpd ?wpo))
    ()

    ((developpe ?wpd ?wpere ?d)(not(same ?wpd ?wpo)))
    ((!!mark ?wpd ?wpere ?d)(mark_path ?wpo ?wpere))

)

(Operator (!!mark_to_dest ?wpd ?wpere)
    ((pendant ?wpd ?wpere ?d))
    ((next ?wpere ?wpd ?d)(not(fini))(not(pendant ?wpd ?wpere ?d)))
)

(Operator (!!mark ?wpd ?wpere ?d)
    ()
    ((next ?wpere ?wpd ?d))
)

(Operator (!!clean_path)
    ()
    (
      (forall (list ?wp)
        ((pendant ?wp ?wpx ?d)) ((not(pendant ?wp ?wpx ?d))))
      (forall (list ?wp)
        ((developpe ?wp ?wpx ?d)) ((not(developpe ?wp ?wpx ?d))))
      (forall (list ?wp)
        ((next ?wp ?wpx ?d)) ((not(next ?wp ?wpx ?d))))
      (not(fini)) (not(init))
    )
)
```

Le prédicat `(pendant ?wp ?wpp ?d)` indique que `?wp` est dans *openList* avec comme nœud précédent `?wpp` et une distance à l'origine `?d`.

Le prédicat `(developpe ?wp ?wpp ?d)` indique les mêmes informations pour les nœuds déjà développés de *closedList*.

L'operateur `(!!develop)` calcule, pour les nœuds voisins qui ne sont pas encore développés, la distance à l'origine en passant par le nœud en cours de développement puis, pour les nœuds voisins déjà pendants, il ne conserve que la meilleure distance. Finalement, l'opérateur indique que le nœud en cours de développement n'est plus pendant mais développé.

Cette simulation de l'algorithme de Dijkstra est initialisée et exécutée par la méthode `find_path` avec comme paramètres le nœud origine et le nœud destination. L'opérateur `!!clean_path` n'intervient pas dans le processus de recherche mais permet de nettoyer l'état courant en vue d'une nouvelle recherche de chemin.

Annexe B

Fonctionnalités et formalisme du planificateur HTN

B.1 Fonctionnalités et syntaxe BNF du langage

Dans ce paragraphe, nous présentons différentes améliorations du planificateur par rapport au langage de base. Ces améliorations sont inspirées des fonctionnalités du planificateur SHOP2.

Les fonctions

Le planificateur permet d'utiliser des fonctions. Ces fonctions peuvent être des fonctions à résultat numérique (+, -, *, /, %, min, max), ou booléen (=, <, >, ,=, >=, member). Elles prennent en entrée des variables, des constantes ou d'autres fonctions. Un cas particulier est la fonction member qui prend comme deuxième paramètre une liste. L'appel d'une fonction s'effectue par l'utilisation du mot-clé `call`. Par exemple, étant donné une variable x unifiée et définie sur $\mathbb{R}^+$, le prédicat :

$$(\texttt{call} \leq (\texttt{call} \ + \ (\texttt{call} \ + \ 2 \ \ 3) \ ?x) \ 6)$$

renvoie vrai pour $x \ \in \ [0, 1]$.

Les axiomes

En plus des opérateurs et méthodes, le langage du planificateur permet d'exprimer des axiomes. Les axiomes sont des expressions logiques de la forme :

$$a \ = \ L_1 \ \vee \ L_2 \ ... \ \vee \ L_n$$

L'axiome a est évalué à vrai si au moins une des formule logique L_i est évaluée à vrai. Ils sont représentés par :

$$(:- \ a \ \ L_1 \ L_2 \ ... \ L_n)$$

Par exemple, on peut formuler l'axiome `(different ?x ?y)` signifiant que la valeur associée à la variable `?x` doit être différente de celle associée à la variable `?y`, de la manière suivante :

```
(:-(different ?x ?y)((call < ?x ?y))((call > ?x ?y)))
```

Fonctions particulières

Ce type de fonctions englobe les fonctions définies dans le langage : `assign` et `forall`.

La fonction `assign` permet d'assigner une valeur à une variable, *i.e.*, ajouter à la substitution courante le couple (variable, valeur).

```
(assign ?resultat (call + 1 ?x))
```

La quantification universelle (`forall`) est une expression de la forme :

$$(\texttt{forall}\ V\ Y\ Z)$$

dans laquelle Y et Z sont des expressions logiques et V est une liste de variables contenues dans Y. Lorsqu'elle se trouve au niveau des préconditions d'un opérateur ou d'une méthode, le sens de la fonction de quantification universelle est le suivant : pour chaque substitution u possible de la liste de variables V, si les prédicats issus de la substitution des variables des prédicats de Y par les valeurs associées contenues dans u (notée Y^u) sont satisfaits dans l'état courant, alors les prédicats de Z^u doivent l'être aussi. Lorsque la fonction `forall` fait partie des effets d'un opérateur ou d'une méthode, alors le sens est le suivant : Si Y^u est satisfait dans l'état courant du monde, alors les effets Z^u peuvent s'appliquer.

```
(forall (list ?x)((station ?x)(tarif SP95 ?x ?v))
  ((not(tarif SP95 ?x ?v))(tarif SP95 ?x (call + ?v 1))))
```

Dans cet exemple, le prix du super sans plomb 95 dans toutes les stations qui en vendent est augmenté de 1 euro.

Gestion des listes

Le mot clé `list` permet de définir des listes de prédicats ou plus généralement de termes. l'opérateur « | » est l'opérateur de concaténation de listes. Le symbole `nil` représente la liste vide.

Préconditions particulières

Les mots-clés `:first` et `:sort-by` permettent d'optimiser l'unification des préconditions. `:first` signifie que seule la première substitution trouvée doit être envisagée même si celle-ci mène à un échec En cas de retour arrière dans l'algorithme principal, les autres substitutions possibles ne seront pas prises en compte. Le mot-clé `sort-by` permet de trier les substitutions possibles sur les valeurs numériques d'une des variables. Par exemple, si l'on cherche à appliquer la méthode suivante :

```
(Method (paint ?caisse ?color)
  (:sort-by ?cost > ((caisse ?caisse) (color ?color)
                     (cost ?cost ?color)))
  ((!take-color ?color)(!do-paint ?caisse ?color))
  ()
)
```

sur l'état :

```
s = ( (caisse c1)(color red)(color blue)(color green)
      (cost 1 red)(cost 2 blue)(cost 3 green) )
```

Alors les substitutions possibles seront classées de la façon suivante :

$$\sigma_1 = \left\{ \begin{array}{lcl} ?caisse & \leftarrow & c1 \\ ?color & \leftarrow & \texttt{green} \\ ?cost & \leftarrow & 3 \end{array} \right\}$$

$$\sigma_2 = \left\{ \begin{array}{lcl} ?caisse & \leftarrow & c1 \\ ?color & \leftarrow & \texttt{blue} \\ ?cost & \leftarrow & 2 \end{array} \right\}$$

$$\sigma_3 = \left\{ \begin{array}{lcl} ?caisse & \leftarrow & c1 \\ ?color & \leftarrow & \texttt{red} \\ ?cost & \leftarrow & 1 \end{array} \right\}$$

Ainsi, le planificateur cherchera un plan en utilisant comme valeur pour la variable *?color* d'abord la valeur `green`, puis en cas d'échec la valeur `blue` et enfin la valeur `red`.

Protection des prédicats

La protection d'un prédicat s'effectue par l'utilisation du mot-clé `:protection`. Le but d'une protection est d'empêcher un opérateur de supprimer un prédicat de l'état courant, ce qui a pour conséquence de bloquer l'application de l'opérateur concerné. Une protection est un compteur associé à chaque prédicat de l'état courant. Si le compteur est à zéro, alors le prédicat n'est pas protégé. Exemple d'ajout et de suppression d'une protection :

```
(:protection (at John Paris))
(not (:protection (at John Paris)))
```

Ordre partiel

L'algorithme de planification dans sa version de base est un algorithme de planification totalement ordonné. L'amélioration apportée permet d'ordonner partiellement les sous-tâches. Le mot-clé `:unordered` associé à une liste des tâches indique que celles-ci peuvent être planifiées dans un ordre quelconque. Le planificateur peut alors entrelacer les différentes sous-tâches. Le mot-clé `:immediate` associé à une tâche indique que celle-ci doit être réalisée prioritairement. Si l'instruction `:immediate` n'est pas présente pour les tâches sans prédécesseur des différentes listes de tâches, alors les tâches sont exécutées dans l'ordre par défaut. Par exemple : soit `T` et `U` deux listes de tâches :

```
T = (t1, t2, t3, t4)
U = ((:immediate u1), (:immediate u2), u3, u4)
```

et `M` la liste de tâches principale :

```
M = (:unordered T U)
```

L'algorithme planifiera les tâches selon l'ordre suivant :

```
u1, u2, t1, t2, t3, t4, u3, u4
```

Syntaxe BNF du langage HTN

$< letter >$: := ["**a**"-"**z**", "**A**"-"**Z**", "**_**", "**-**", "**#**"] ;

$< digit >$: := ["**0**"-"**9**"] ;

$< dot >$: := "**.**" ;

$< number >$: := $< digit >+$ | $(< digit >+)< dot >(< digit >+)$
| -$< digit >+$ | -$(< digit >+)< dot >(< digit >+)$;

$< char >$: := $< letter >$ | $< digit >$;

$< symbol >$: := $< letter >(< char >*)$;

$< name >$: := $< symbol >$;

$< problem >$: := "**(problem** "$< name >$
$< initial\_state >< task\_list >$")" ;

$< domain >$: := "**(domain** "$< name >$
$(< axiom > | < operator > | < method >)*$")" ;

$< initial\_state >$: := "("$< literal >*$")" ;

$< task\_list >$: := "**nil**"
| "("[" **:ordered**"]$< task\_atom >*$")"
| "("[" **:unordered**"]$< task\_atom >*$")" ;

$< operator >$: := $< util\_operator > | < plan\_operator >$

$< util\_operator >$: := "**(Operator** "$< util\_op\_head >$
$< preconditions >< effects >$")" ;

$< plan\_operator >$: := "**(Operator** "$< plan\_op\_head >$
$< preconditions >< effects >< att\_pre >$
$< behav\_pre >< geom\_eff >$")" ;

$< method >$: := "**(Method** "$< method\_head >$
$(< decomposition >*)$")" ;

$< axiom >$: := "(**:-**"$< literal >< conjunction\_list >$")" ;

$< operator\_head >$: := $< plan\_op\_head > | < util\_op\_head >$;

$< plan\_op\_head >$: := "(**!**"$< symbol >$ $(< term >*)$ ")" ;

$< util\_op\_head >$: := "(**!!**"$< symbol >$ $(< term >*)$ ")" ;

$< method\_head >$: := "("$< symbol >$ $(< term >*)$ ")" ;

$< decomposition >$: := $< precondition >< task\_list >$;

$< task\_atom >$: := [" **:immediate**"]$< operator\_head >$
| [" **:immediate**"]$< method\_head >$;

$<preconditions>$ ::= [" **:sort-by** "$<variable>$("<"|">")]$<conjunction>$;
$<effects>$ ::= $<conjunction>$;
$<att\_pre>$ ::= *cf.* tableau 3.1 ;
$<behav\_pre>$ ::= *cf.* tableau 3.1 ;
$<geom\_eff>$ ::= *cf.* tableau 3.3 ;
$<conjunction\_list>$ ::= ($<conjunction>$*) ;
$<conjunction>$ ::= nil
| $<term>$
| "("[" **:first**"]$<literal>$*")" ;
$<literal>$ ::= $<logical\_atom>$
| "**(not** "$<logical\_atom>$")" ;
$<logical\_atom>$ ::= $<term>$
| "("$<symbol>$($<term>$*)")" ;
$<term>$ ::= $<variable>$
| $<constant>$
| $<list\_term>$
| $<call\_term>$
| $<function\_term>$
| $<special\_function>$;
$<variable>$ ::= " **?**"$<symbol>$;
$<constant>$ ::= $<symbol>$ | $<number>$;
$<list\_term>$ ::= "**(list** "$<term><sub\_list>$")" ;
$<sub\_list>$ ::= nil
| "|"$<term><sub\_list>$
| "|"$<term>$;
$<var\_list>$ ::= "**(list** "$<variable><sub\_var\_list>$")" ;
$<sub\_var\_list>$ ::= **nil**
| "|"$<variable><sub\_var\_list>$
| "|"$<variable>$;
$<call\_term>$ ::= "**(call** "$<function><term><term>$")" ;
$<function\_term>$ ::= "("$<symbol>$($<term>$*)")" ;
$<special\_function>$ ::= $<assign\_function>$
| $<forall\_function>$;
$<assign\_function>$ ::= "**(assign** "$<variable><term>$")" ;
$<forall\_function>$ ::= "**(forall** "$<list><conjunction><conjunction>$")"
$<function>$ ::= $<boolean\_function>$
| $<numeric\_function>$;

$< boolean\_function >$: := "=" | ">" | "<" | ">=" | "<=" | "**member**" ;

$< numeric\_function >$: := "+" | "-" | "*" | "/" | "%" | "**max**" | "**min**" ;

B.2 Domaine de planification de la mission 1

```
(Domain domain_mission1 (

;; ----------------------------------------------------------------

;; initialisation du rover
(Operator (!init_rover ?r)
 ((rover ?r)(central_waypoint ?o)(not(initialized ?r)))
 (
  (agent r ?r)(object o ?o)
  (setProperty(r.x,o.x))
  (setProperty(r.y,o.y))
  (setProperty(r.heading,0))
 )
 ()
 ()
 ((initialized ?r))
)

;; ----------------------------------------------------------------

(Method (do_collect ?r)
 ((has_enough_rock))
 ()
 ((is_full_store ?r)(central_waypoint ?wp))
 ((!empty_store ?r ?wp)(do_collect ?r))
 ()
 ((collect_or_analyse ?r)(do_collect ?r))
)

(Method (collect_or_analyse ?r)
 (:sort-by ?d < ((waypoint ?wp)(not(unaccessible ?wp))
  (heuristic(distance_from_waypoint ?r ?wp ?d))))
 ((verify_accessibility ?r ?wp))
)

(Method (verify_accessibility ?r ?wp)
 ()
 ((collect_or_analyse_wp ?r ?wp))
 ((not(analysed ?wp)))
 ((!!assert (unaccessible ?wp)))
)

(Method (collect_or_analyse_wp ?r ?wp)
 ((at_rock_sample ?wp ?q)(call > ?q 0))
 ((collect_rock ?r ?wp))
```

```
 ((not(analysed ?wp)))
 ((analyse_soil ?r ?wp 0))
)

(Method (collect_rock ?r ?wp)
 ((at_rock_sample ?wp ?qs)(call > ?qs 0)(store_level ?r ?qr)
  (max_store ?r ?qm)(assign ?qrfree (call - ?qm ?qr))
  (call > ?qrfree 0)(call >= ?qs ?qrfree))
 ((!collect_rock ?r ?wp ?qrfree))
 ((at_rock_sample ?wp ?qs)(call > ?qs 0)(store_level ?r ?qr)
  (max_store ?r ?qm)(assign ?qrfree (call - ?qm ?qr))
  (call > ?qrfree 0)(call < ?qs ?qrfree))
 ((!collect_rock ?r ?wp ?qs))
)

(Method (analyse_soil ?r ?wp ?n)
 ((max_analyse ?qn)(call < ?n ?qn))
 ((!analyse_soil ?r ?wp ?n)(analyse_soil ?r ?wp (call + ?n 1)))
 ((call > ?n 0))
 ((!!compute_rock_qty ?wp ?n))
)

;; ---------------------------------------------------------------------

(Operator (!analyse_soil ?r ?wp ?n)
 ((analyse_radius ?rad)(assign ?radius (call * (call +?n 1) ?rad)))
 ((agent r ?r)(object o ?wp)
  (distance(r, o) = ?radius)
  (rel_angle(r, o) = cos-and-sin(1.5708)))
 ((until(distance, mult(6.28318, ?radius)))
  (constant(distance(r, o))))
 ()()
)

(Operator (!collect_rock ?r ?wp ?q)
 ((at_rock_sample ?wp ?qs)(store_level ?r ?qr))
 ((agent r ?r)(object o ?wp)
  (distance(r, o) < 5)
  (rel_angle(r,o) = cos-and-sin(0)))
 ()()
 ((not(at_rock_sample ?wp ?qs))(at_rock_sample ?wp (call - ?qs ?q))
  (not(store_level ?r ?qr))(store_level ?r (call + ?qr ?q)))
)

(Operator (!empty_store ?r ?wp)
 ((store_level ?r ?qr)(central_rock_level ?q))
 ((agent r ?r)(object o ?wp)
```

```
  (distance(r, o) = 0))
 ()()
 ((not(central_rock_level ?q))(central_rock_level (call + ?q ?qr))
  (not(store_level ?r ?qr))(store_level ?r 0))
)

;; -----------------------------------------------------------------

(Operator (!!compute_rock_qty ?wp ?n)
 ((rock_quantity ?qr))
 ((at_rock_sample ?wp (call * ?n ?qr))(analysed ?wp))
)

(Operator (!!assert ?g)
  ()
  ?g
)

(:- (is_full_store ?r)
 ((store_level ?r ?qr)(max_store ?r ?qm)
  (assign ?pc (call * (call / ?qr ?qm) 100))
  (call >= ?pc 75))
)

(:- (has_enough_rock)
 ((central_rock_level ?qg)(mission_rock_qty ?qm)(call >= ?qg ?qm))
)

;; -----------------------------------------------------------------

))
```

B.3 Domaine de planification de la mission 2

```
(Domain domain_mission2 (

;; ------------------------------------------------------------------

;; initialisation du rover
(Operator (!init_rover ?r)
 ((rover ?r)(central_waypoint ?o)(total_energy ?q)
  (not(initialized ?r)))
 (
  (agent r ?r)(object o ?o)
  (setProperty(r.x,o.x))
  (setProperty(r.y,o.y))
  (setProperty(r.heading,0))
  (setProperty(r.energy_level,?q))
 )
 ()
 ()
 ((initialized ?r))
)

;; ------------------------------------------------------------------

(Method (do_mission ?goals)
 ((rover ?r))
 ((assert_goals ?goals nil)
  (choose_and_collect ?r))
)

(Method (choose_and_collect ?r)
 ((is_full_store ?r)(central_waypoint ?wp))
 ((!empty_store ?r ?wp)(!update_energy_level ?r)
  (choose_and_collect ?r))
 (:sort-by ?d < ( (at_rock_sample ?wp ?qs) (call > ?qs 0)
  (heuristic(distance_from_waypoint ?r ?wp ?d))))
 ((collect_rock ?r ?wp)(!update_energy_level ?r)
  (!!remove (goal(need_sample_of ?wp)))(choose_and_collect ?r))
 ((not(goal(need_sample_of ?loc)))(central_waypoint ?wp)
  (store_level ?r ?ql)(call > ?ql 0))
 ((!empty_store ?r ?wp)(!update_energy_level ?r))
 ((not(goal(need_sample_of ?loc)))(central_waypoint ?wp)
  (store_level ?r ?ql)(call = ?ql 0))
 ()
)
```

```
(Method (collect_rock ?r ?wp)
 ((at_rock_sample ?wp ?qs)(store_level ?r ?qr)
  (max_store ?r ?qm)(assign ?qrfree (call - ?qm ?qr))
  (call > ?qrfree 0)(call >= ?qs ?qrfree))
 ((!collect_rock ?r ?wp ?qrfree))
 ((at_rock_sample ?wp ?qs)(store_level ?r ?qr)
  (max_store ?r ?qm)(assign ?qrfree (call - ?qm ?qr))
  (call > ?qrfree 0)(call < ?qs ?qrfree))
 ((!collect_rock ?r ?wp ?qs))
)

;; ---------------------------------------------------------------

(Operator (!collect_rock ?r ?wp ?q)
 ((at_rock_sample ?wp ?qs)(store_level ?r ?qr)(needed_energy ?tot)
  (energy_sample_unit ?qe)(assign ?energy_sample (call * ?q ?qe)))
 ((agent r ?r)(object o ?wp)
  (distance(r, o) = 5)
  (rel_angle(r,o) = cos-and-sin(0)))
 ()
 ((conso_energy r ?energy_travel))
 ((not(at_rock_sample ?wp ?qs))(at_rock_sample ?wp (call - ?qs ?q))
  (not(store_level ?r ?qr))(store_level ?r (call + ?qr ?q))
  (not(needed_energy ?tot))(needed_energy (call + ?energy_sample ?energy_travel))
  (has_sample_of ?wp))
)

(Operator (!update_energy_level ?r)
((total_energy ?tot)(needed_energy ?q)
 (assign ?new_tot (call - ?tot ?q)))
((agent r ?r)(setProperty(r.energy_level, ?new_tot)))
()()
((not(total_energy ?tot))(total_energy ?new_tot))
)

(Operator (!empty_store ?r ?wp)
 ((store_level ?r ?qr)(central_rock_level ?q)(needed_energy ?tot))
 ((agent r ?r)(object o ?wp)
  (distance(r, o) = 0))
 ()
 ((conso_energy r ?energy_travel))
 ((not(central_rock_level ?q))(central_rock_level (call + ?q ?qr))
  (not(store_level ?r ?qr))(store_level ?r 0)
  (not(needed_energy ?tot))(needed_energy ?energy_travel))
)

;; ---------------------------------------------------------------
```

```
(:- (is_full_store ?r)
 ((store_level ?r ?qr)(max_store ?r ?qm)
  (assign ?pc (call * (call / ?qr ?qm) 100))
  (call >= ?pc 75))
)

(Method (assert_goals (list ?goal | ?goals) ?out)
 ()
 ((assert_goals ?goals (list (goal ?goal) | ?out)))
)

(Method (assert_goals nil ?but)
 ()
 ((!!assert ?but))
)

(Operator (!!remove ?g)
 ()
 (not ?g)
)

(Operator (!!assert ?g)
 ()
 ?g
)

(Method (print_results)
 ((total_energy ?q1)(central_rock_level ?q2))
 ((!print energy_left ?q1)(!print rock_collected ?q2))
)

(Operator (!print ?what ?val)()()()()())

;; -----------------------------------------------------------------

))
```

Annexe C

Interfaces Utilisateur

Dans cette annexe, nous présentons les interfaces utilisateur disponibles dans le logiciel que nous avons développé. Ces interfaces permettent de créer un nouveau projet, de le configurer puis de visualiser les résultats du processus de planification.

C.1 Création d'un nouveau projet

Lors de la création d'un nouveau projet de planification, la première interface (figure C.1) permet de configurer le nom du projet, la taille de l'environnement ainsi que le nombre de cellules constituant le découpage de cet environnement, les noms des différents fichiers constituant le projet et, finalement, le répertoire de destination du projet.

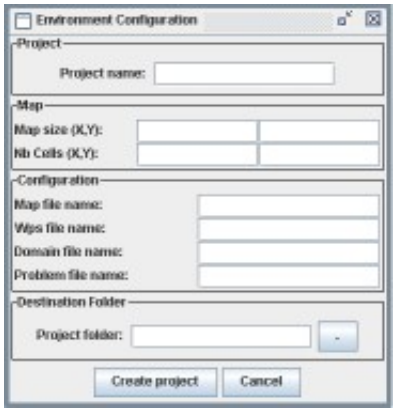


FIGURE C.1 – Création d'un nouveau projet

Après la création du projet, le logiciel propose un ensemble de fonctionnalités (figure C.2) permettant de définir le contenu de ce projet : dessiner l'envi-

ronnement, placer les objets, configurer les objets, configurer le robot, écrire le domaine de planification et écrire le problème associé.

FIGURE C.2 – Options de création d'un projet

La création de l'environnement (figure C.3) se fait par le dessin de polygones représentant les obstacles. Les polygones sont dessinés en définissant à l'aide de la souris leurs sommets. Lorsque le dernier sommet correspond au premier sommet, le polygone est fermé et l'obstacle est créé.

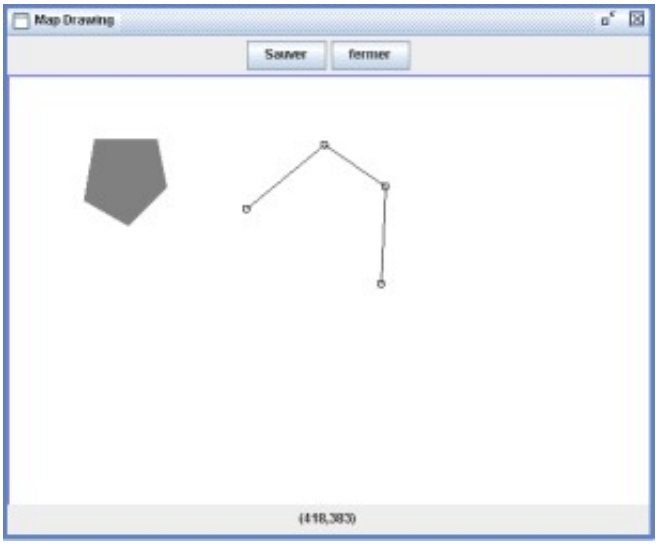


FIGURE C.3 – Dessin de la carte de l'environnement

Une fois l'environnement créé, l'utilisateur peut placer les objets à l'aide du pointeur de la souris. Ceux-ci sont représentés sur la figure C.4 par des points jaunes.

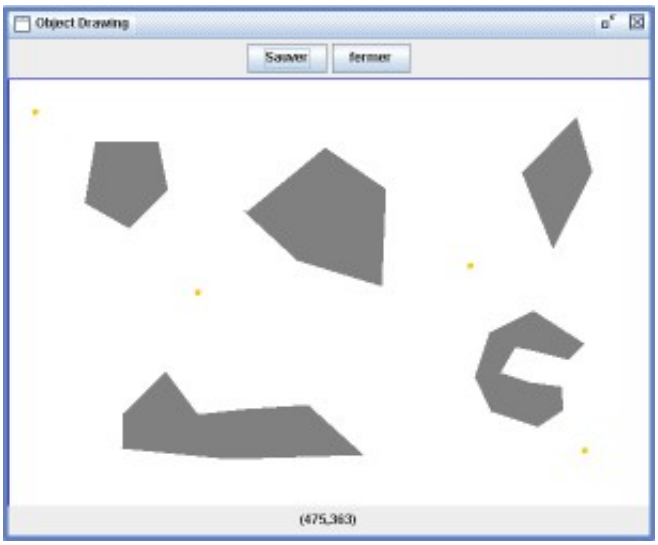


FIGURE C.4 – Placement des objets dans l'environnement

À chacun de ces objets, il est nécessaire d'associer une étiquette symbolique permettant de faire le lien entre le module de raisonnement symbolique et le module de raisonnement géométrique. La figure C.5 présente l'interface permettant d'attribuer ces étiquettes. Elle récapitule également les coordonnées de ces objets.

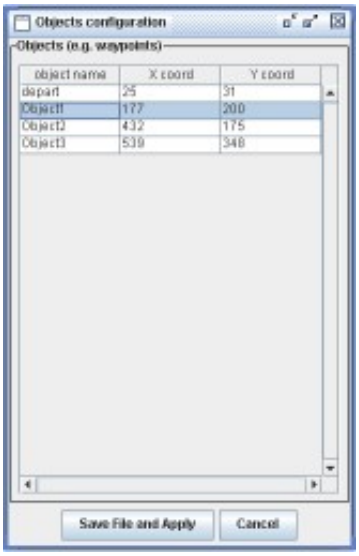


FIGURE C.5 – Configuration des étiquettes des objets

L'interface de configuration du robot (figure C.6) permet de définir l'angle de giration maximum du robot, sa vitesse, son niveau d'énergie et sa consommation d'énergie par unité de distance.

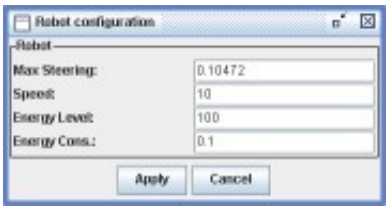


FIGURE C.6 – Configuration des paramètres du robot

Finalement, l'utilisateur doit définir le domaine et le problème de planification à l'aide d'un éditeur de texte (figure C.7).

FIGURE C.7 – Éditeur permettant de définir le domaine et le problème

C.2 Configuration d'un projet

Après avoir créé ou chargé un projet, l'utilisateur peut effectuer un certain nombre de configurations additionnelles.

Il peut, par exemple, ajuster les paramètres de configuration du planificateur CELL-RRT (figure C.8) : changement de découpage, options de définition

du corridor, configuration de génération des nouvelles configurations dans l'environnement, configuration du tirage aléatoire de ces configurations géométriques.

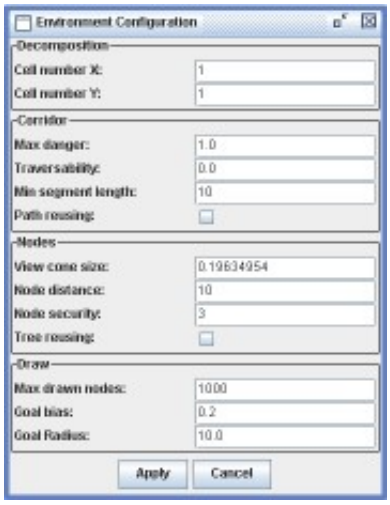


FIGURE C.8 – Configuration de CELL-RRT

Il peut aussi régler les paramètres d'affichage de CELL-RRT (figure C.9) : afficher le découpage de l'environnement, afficher les arbres RRT en cours de développement...

FIGURE C.9 – Options d'affichage de CELL-RRT

Le planificateur de tâches possède également une interface de configuration (figure C.10) dont les options sont :

- réglage du nombre maximum de solutions à trouver, fixé par défaut à une solution ;
- affichage du plan : 0 pour uniquement les opérateurs qui ont des contraintes géométriques, 1 pour tous les opérateurs et 2 pour les opérateurs et les méthodes développées ;
- réglage du niveau d'information durant le processus de planification (0 = pas de message, 9 = tous les messages).

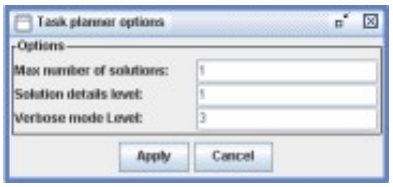


FIGURE C.10 – Options du planificateur de tâches

C.3 Visualisation des résultats

L'interface représentée sur la figure C.11 est l'interface principale du logiciel et permet, d'une part, de suivre la construction d'une solution au problème donné et, d'autre part, de visualiser cette solution à la fin du processus de planification. Cette interface est composée de 4 zones :

- une zone de visualisation du résultat graphique, *i.e.,* les déplacements du robot ;
- une zone de visualisation du processus de planification symbolique ;
- une zone de visualisation des messages, requêtes et conseils échangés entre les deux modules de raisonnement ;
- une zone de messages systèmes indiquant notamment les erreurs survenus, le temps de calcul...

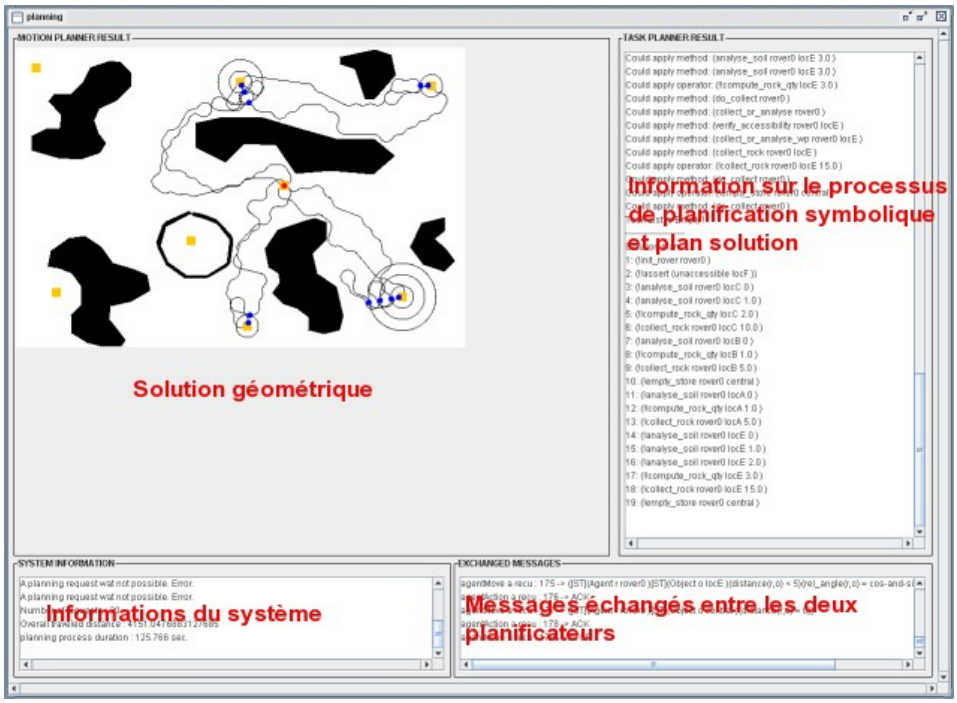


FIGURE C.11 – Fenêtre principale d'affichage des résultats

Bibliographie

M. Akinc, K. Bekris, B. Chen, A. Ladd, E. Plaku, and L. Kavraki. Probabilistic roadmaps of trees for parallel computation of multiple query roadmaps. In *Robotics Research : The 11th International Symposium*, pages 80–89, 2003.

R. Alami, J.P. Laumont, and T. Simeon. Two manipulation planning algorithms. In *workshop on Algorithmic foundations of robotics*, pages 109–125, 1995.

R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research*, 17 :31–337, 1998.

J. Baltié, E. Bensana, P. Fabiani, J.L. Farges, S. Millet, P. Morignot, B. Patin, G. Petitjean, G. Pitois, and J.C. Poncet. Multi-vehicle missions : architecture and algorithms for distributed online planning. In *Artificial Intelligence for Advanced Problem Solving Techniques*, pages 1–22. Information Science Reference, 2008.

S. Biundo, R. Holzer, and B. Schattenberg. Dealing with continuous resources in AI planning. In *International Workshop on Planning and Scheduling for Space*, pages 213–218, 2004.

A. L. Blum and M.L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90 :281–300, 1997.

R. Bonasso and D. Kortenkamp. Using a layered control architecture to alleviate planning with incomplete information. In *the AAAI Spring Symposium, Planning with Incomplete Information for Robot Problems*, pages 1–4, 1996.

B. Bonet and H. Geffner. - HSP : Heuristic search planner - in the AIPS-98 planning competition. *AI Magazine*, 21 :13–33, 2000.

B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129 :5–33, 2001.

A. Botea, M. Müller, and J. Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1 :7–28, 2004.

R. Brooks. A robust layered control system for a mobile robot. *IEEE Journal on Robotics and Automation*, 2(1) :14–23, 1986.

J. Bruce and M. Veloso. Real-time randomized path planning for robot navigation. In *IEEE International Conference on Robots and Systems*, 2002.

B. Burns and O. Brock. Single-query motion planning with utility-guided random trees. In *IEEE International Conference on Robotics and Automation*, 2007.

S. Cambon, R. Alami, and F. Gravot. A hybrid approach to intricate motion, manipulation and task planning. *The International Journal of Robotics research*, 28 :104–126, 2009.

S. Cambon. *Planifier avec les contraintes géométriques du mouvement et de la manipulation*. PhD thesis, Université de Toulouse, 2005.

A. Cesta and C. Stella. A time and resource problem for planning architectures. In *4th European Conference on Planning*, pages 117–129, 1997.

E. Chanthery. *Planification de mission pour un véhicule aérien autonome*. PhD thesis, ENSAE, 2005.

P. Cheng and S.M. LaValle. Reducing metric sensitivity in randomized trajectory design. In *IEEE International Conference on Intelligent Robots and Systems*, pages 43–48, 2001.

J. Choi and E. Amir. Combining planning and motion planning. In *IEEE International Conference on Robotics and Automation*, 2009.

K. Currie and A. Tate. O-Plan : The open planning architecture. *Artificial intelligence*, 52 :49–86, 1991.

E. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1 :269–271, 1959.

B. Drabble and A. Tate. The uses of optimistic and pessimistic resource profiles to inform search in an activity based planner. In *2nd International Conference on Artificial Intelligence Planning Systems*, pages 243–248, 1994.

L. E. Dubins. On curves of minimal length with a constraint on average curvature, and with prescribed initial and terminal positions and tangents. *American Journal of Mathematics*, 79 :497–516, 1957.

A. El-Kholy and B. Richards. Temporal and resource reasoning in planning : the parkPLAN approach. In *12th European Conference on Artificial Intelligence*, pages 614–618, 1996.

G. Ernst, A. Newell, and H. Simon. GPS : A case study in generality and problem solving. Academic Press, 1969.

K. Erol, J. Hendler, and D. Nau. UMCP : A sound and complete procedure for hierarchical task-network planning. In *Artificial Intelligence Planning Systems*, pages 249–254, 1994.

K. Erol, J. Hendler, and D. Nau. HTN planning : Complexity and expressivity. In *AAAI 12th National Conference on Artificial Intelligence*, pages 1123–1128, 94.

T. Estlin, R. Volpe, I. Nesnas, D. Mutz, F. Fisher, B. Engelhardt, and S. Chien. Decision-making in a robotic architecture for autonomy. In *6th International Symposium on Artificial Intelligence, Robotics and Automation in Space*, 2001.

D. Ferguson, N. Kalra, and A. Stenz. Replanning with RRTs. In *IEEE International Conference on Robotics and Automation*, pages 1243–1248, 2006.

R.E. Fikes and N.J. Nilsson. STRIPS : A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2 :189–208, 1971.

A. Finzi, F. Ingrand, and N. Muscetolla. Model-based executive control through reactive planning for autonomous rovers. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 879–884, 2004.

M. Fox and D. Long. PDDL 2.1 : an extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20 :61–124, 2003.

E. Gat. Integrating planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots. In *the Tenth National Conference on Artificial Intelligence (AAAI)*, pages 809–815, 1992.

E. Gat. On three-layer architectures. *Artificial Intelligence and Mobile Robots*, 1 :195–210, 1997.

A. Gerevini and D. Long. Plan constraints and preferences in PDDL3. Technical report, University of Brescia, Italy, 2005.

M. Ghallab, D. Nau, and P. Traverso. *Automated planning : Theory and Practice*. Morgan Kaufmann, 2004.

F. Gravot, S. Cambon, and R. Alami. aSyMov : a planner that delas with intricate symbolic and geometric problems. *Robotics Research*, 15 :100–110, 2005.

F. Gravot. *aSyMov : Fondation d'un planificateur robotique intégrant le symbolique et le géométrique.* PhD thesis, Université de Toulouse, 2004.

C. Green. Application of theorem-proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, pages 219–239, 1969.

J. Guitton and J.-L. Farges. Planification de mission pour un ensemble de drones de combat aériens. In *2e Journées francophones de Planification, Décision et Apprentissage*, pages 73–78, 2007.

J. Guitton and J.-L. Farges. Geometric and symbolic reasoning for mobile robotics. In *3rd National Conference on Control Architecture of Robots*, pages 76–97, 2008.

J. Guitton and J.-L. Farges. Vers un modèle de planification hybride pour la planification d'action et de déplacement. In *3e Journées francophones de Planification, Décision et Apprentissage*, pages 23–32, 2008.

J. Guitton and J.-L. Farges. Taking into account geometric constraints for task-oriented motion planning. In *Workshop on Bridging the Gap between Task and Motion Planning*, pages 26–33, 2009.

J. Guitton and J.-L. Farges. Towards a hybridization of task and motion planning for robotic architecture. In *Int. workshop on Hybrid Control of Autonomous Systems*, pages 21–24, 2009.

J. Guitton, J.-L. Farges, and R. Chatila. A planning architecture for mobile robotics. In *1st Mediterranean Conference on Intelligent Systems and Automation*, pages 162–167, 2008.

J. Guitton, J.-L. Farges, and R. Chatila. Cell-RRT : Decomposing the environment for better plan. In *IEEE/RSJ International Conference on Intelligent RObots and Systems (IROS)*, pages 5776–5781, 2009.

J. Guitton. Cell-RRT : décomposer l'environnement pour mieux planifier. In *Rencontre des Jeunes Chercheurs en Intelligence Artificielle*, pages 89–106, 2009.

E. Guéré and R. Alami. Let's reduce the gap between task planning and motion planning. In *IEEE International Conference on Robotics and Automation*, pages 15–20, 2001.

K. Hammond. CHEF : A model of case-based planning. In *AAAI*, pages 267–271, 1986.

P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. In *IEEE Transactions on Systems Science and Cybernetics*, volume 4, pages 100–107, 1968.

J. Hoffmann and B. Nebel. The FF planning system : Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14 :253–302, 2001.

J. Hoffmann, S. Edelkamp, S. Thiebaux, R. Englert, F.Liporace, and S. Trüg. Engineering benchmarks for planning : the domains used in the deterministic part of IPC-4. *Journal of Artificial Intelligence*, 26 :453–541, 2006.

A. Howe and E. Dahlman. A critical assessment of benchmark comparison in planning. *Journal of Artificial Intelligence Research*, 17 :1–33, 2002.

F. Ingrand. Architectures logicielles pour la robotique autonome. In *Journées Nationales de Rechercher en Robotique*, 2003.

M. Kalisiak and M. von de Panne. RRT-blossom : RRT with a local flood-fill behavior. In *IEEE International Conference on Robotics and Automation*, 2006.

S. Kambhampati, M.R. Cutkosky, J.M. Tenenbaum, and S.H. Lee. Integrating general purpose planners and specialized reasoners :case study of a hybrid planning architecture. In *IEEE transactions on Systems, Man and Cybernetics*, volume 23, pages 1503–1518, 1993.

L. Kavraki and J.-C. Latombe. Randomized preprocessing of configuration for fast path planning. In *IEEE Int. Conf. on Robotics and Automation*, volume 3, pages 2138–2145, 1994.

L. Kavraki, P. Svestka, J.-C. Latombe, and M.H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. In *IEEE Transactions on Robotics and Automation.*, volume 12, pages 566–580, 1996.

J. Koehler. Planning under resource constraints. In *ECAI'98, 19th European Conference on Artificial Intelligence*, pages 489–493, 1998.

S. Koenig, M. Likhachev, and D. Furcy. Lifelong planning A*. *Artificial Intelligence*, 155 :93–146, 2004.

J. J. Kuffner and S. M. LaValle. RRT-Connect : An efficient approach to single-query path planning. In *IEEE International Conference on Robotics and Automation*, pages 995–1001, 2000.

P. Laborie and M. Gallab. Planning with sharable resource constraints. In *International Joint Conference on Artificial Intelligence*, 1995.

P. Laborie and M. Ghallab. IxTeT : an integrated approach for plan generation and scheduling. In *INRIA/IEEE Symposium on Emerging Technologies and Factory Automation*, volume 1, pages 485–495, 1995.

P. Laborie. *IxTeT : Une approche intégrée pour la gestion de ressources et la synthèse de plans*. PhD thesis, Ecole Nationale Supérieure des Télécommunications, 1995.

P. Laborie. Algorithms for propagating resource constraints in AI planning and scheduling : Existing approaches and new results. *Artificial Intelligence*, 143 :151–188, 2003.

B. Lamare and M. Ghallab. Integrating a temporal planner with a path planner for a mobile robot. In *AIPS Workshop Integrating planning, scheduling and execution in dynamic and uncertain environments*, pages 144–151, 1998.

J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, 1991.

S.M. LaValle and J.J. Kuffner. Randomized kinodynamic planning. In *IEEE International Conference on Robotics and Automation*, volume 1, pages 473–479, 1999.

S. M. LaValle. Rapidly-exploring random trees : A new tool for path planning. Technical report, Computer Science Dept., Iowa State University, 1998.

T. Lozano-Perez, J. Jones, E. Mazer, and P. O'Donnell. Task-level planning of pick-and-place robot motions. *IEEE Computer*, 22(3) :21–29, 1989.

D. McDermott, M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld, and D. Wilkins. PDDL - the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165 1.2, Yale Center for Computational Vision and Control, 1998.

N. Muscettola, G. Dorais, C. Fry, R. Levinson, and C. Plaunt. IDEA : Planning at the core of autonomous reactive agents. In *3rd International NASA Workshop on Planning and Scheduling for Space*, 2002.

D. Nau, Y. Cao, and H.mu noz Avila. SHOP : Simple hierarchical ordered planner. In *International Joint Conference on Artificial Intelligence*, pages 968–973, 1999.

D. Nau, T.-C. Au, O. Ilghami, U. Kuter, J. W. Murdock, D. Wu, and F. Yaman. SHOP2 : An HTN planning system. *Artificial Intelligence Research*, 20 :380–404, 2003.

P. Naur. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 3 :299–314, 1960.

N. Nilsson. *Principles of Artificial Intelligence*. Springer-Verlag, 1982.

E. Pednault. ADL and the state-transition model of action. *Journal of logic and computation*, 4 :467–512, 1994.

E. Plaku, K.E. Bekris, B.Y Chen, A.M. Ladd, and L. Kavraki. Sampling-based roadmap of trees for parallel motion planning. *IEEE Transactions on Robotics*, 21(4) :597–608, 2005.

E. Plaku, L. Kavraki, and M. Y. Vardi. Discrete search leading continuous exploration for kinodynamic motion planning. In *Proceedings of Robotics : Science and Systems*, 2007.

E. Plaku, L. Kavraki, and M. Y. Vardi. Impact of workspace decompositions on discrete search leading continuous exploration (DSLX) motion planning. In *IEEE International Conference on Robotics and Automation*, pages 3751–3756, 2008.

G. Ramkumar. An algorithm to compute the minkowski sum outer-face of two simple polygons. In *twelfth annual symposium on Computational geometry*, pages 234–241, 1996.

S. Rodriguez, X. Tang, J-M. Lien, and N. M. Amato. An obstacle-based rapidly-exploring random tree. In *IEEE International Conference on Robotics and Automation*, pages 895–900, 2006.

J. Rosenblatt. DAMN : A distributed architecture for mobile navigation. In *AAAI String Symposium on Lessons Learned from Implemented Software Architectures for Physical Agents*, 1995.

E. Sacerdoti. Planning in a hierarchy of abstraction spaces. *Artificial Intelligence*, 5 :115–135, 1974.

E. Sacerdoti. The nonlinear nature of plans. In *International Joint Conference on Artificial Intelligence*, pages 206–214, 1975.

M. Stefik. Planning and meta-planning (MOLGEN : Part 2). *Artificial Intelligence*, 16 :141–170, 1981.

A. Stentz. Optimal and efficient path planning for partially-known environments. In *International Conference on Robotics and Automation*, pages 3310–3317, 1994.

A. Stentz. The focussed D* algorithm for real-time replanning. In *the International Joint Conference on Artificial Intelligence*, 1995.

M. Strandberg. Augmenting RRT-planners with local trees. In *IEEE International Conference on Robotics and Automation*, pages 3258–3262, 2004.

Third International Planning Competition. Hosted at the Artificial Intelligence Planning and Scheduling (AIPS) conference. http ://planning.cis.strath.ac.uk/competition/, 2002.

R. Volpe, I. Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. CLARAty : Coupled layer architecture for robotic autonomy. Technical report, JPL Technical Report D-19975, 2000.

R. Volpe, I Nesnas, T. Estlin, D. Mutz, R. Petras, and H. Das. The CLARAty architecture for robotic autonomy. In *IEEE Aerospace Conference*, 2001.

D.E Wilkins. *Practical Planning : Extending the Classical AI Planning Paradigm*. Morgan Kaufmann, 1988.

A. Yershova, L. Jaillet, T. Simeon, and S. M. LaValle. Dynamic-domain RRTs : Efficient exploration by controlling the sampling domain. In *IEEE International Conference on Robotics and Automation*, pages 3856–3861, 2005.

F. Zacharias, C. Borst, and G. Hirzinger. Bridging the gap between task planning and path planning. In *IEEE International Conference on Intelligent Robot ans Systems*, pages 4490–4495, 2006.

L. Zhang and D. Manocha. An efficient retractation-based RRT planner. In *IEEE International Conference on Robotics and Automation*, pages 3743–3750, 2008.

Architecture hybride pour la planification d'actions et de déplacements

L'autonomie d'un robot mobile se caractérise par sa capacité à agir et à se déplacer dans l'environnement sans intervention humaine. La planification de mission fait intervenir un raisonnement symbolique pour le choix des actions permettant d'accomplir la mission et un raisonnement géométrique pour le calcul des déplacements du robot afin de réaliser ces actions. Dans un premier temps, nous comparons différentes approches permettant de coupler un planificateur de tâches et un planificateur de mouvements. Puis nous proposons une architecture de planification hybride mettant en œuvre un planificateur de tâches et un planificateur de mouvements dont les exécutions sont entrelacées. Nous avons été amenés à étendre le concept d'opérateur de planification afin de permettre l'expression et la prise en compte de préconditions géométriques ainsi que d'effets géométriques. Ces préconditions, définissant géométriquement la manière de réaliser les actions, sont envoyées au module de raisonnement géométrique sous la forme de requêtes de planification. Un chemin est ensuite calculé entre la configuration actuelle du robot et la configuration solution à l'aide d'un algorithme de planification de mouvement appelé Cell-RRT. Cet algorithme est un algorithme probabiliste incrémental qui s'appuie sur le principe de l'algorithme RRT. Il est couplé avec une phase de réduction de l'espace de recherche. Les échanges font également appel à la notion de conseil afin de permettre le guidage de la construction du plan par des heuristiques géométriques et de permettre des phases d'optimisation du plan. Cette architecture hybride est finalement testée sur des scénarios de mission pour un robot mobile.

Mots-clés : planification hybride, planification d'actions, planification de déplacements, raisonnement symbolique et géométrique, robotique mobile, architecture robotique.

Hybrid Architecture for Task and Motion Planning

Autonomy of mobile robots is caracterized by the ability to act and move in their environment without human intervention. Mission planning for a mobile robot involves different reasoning: a symbolic reasoning to choose the actions allowing the robot to accomplish its assigned mission, and a geometric reasoning to compute the robot motions in order to achieve these actions. First, we compare different approaches to couple a task planner and a motion planner. Then, we propose a hybrid planning architecture implementing a task planner and a motion planner whose executions are interleaved. We were brought to extend the concept of planning operator to allow expression and use of geometric preconditions as well as geometric effects. These preconditions, defining geometrically how to perform actions are then sent to the geometric reasoning module in the form of planning requests. A path is thereafter computed between the current robot configuration and the final configuration using a motion planning algorithm called Cell-RRT. This algorithm is an incremental probabilistic algorithm based on the Rapidly-exploring Random Trees (RRT) algorithm. It is coupled with a search space reduction phase. Exchanges between both planners are using the concept of advice to help guide the plan construction by using some heuristics as well as allowing the optimization of the plan. This hybrid architecture is finally tested on mission scenarios for a mobile robot.

Keywords : hybrid planning, task planning, motion planning, symbolic and geometric reasoning, mobile robotics, robotic architecture.