École doctorale n° ED-224 : Cognition Langage Interaction

THÈSE

pour obtenir le grade de docteur délivré par

l'Université de Paris 8

Spécialité doctorale "Informatique Robotique"

présentée et soutenue publiquement par

Halim Djerroud

le 6 décembre 2021

Architecture Robotique pour la Navigation Parmi les Obstacles Amovibles pour un Robot Mobile

Jury

M. Raja Chatila,	Professeur, Sorbonne Université	Rapporteur
Mme. Nicoleta Rogovschi,	HDR, Université Paris Descartes	Rapporteur
Mme. Lynda Seddiki,	MCF, Université Paris 8	Examinateur
M. Hanene Azzag,	HDR, Université Sorbonne Paris Nord	Examinateur
M. Zreik Khaldoun,	Professeur, Université Paris 8	Président
M. Arab Ali Cherif,	Professeur, Université Paris 8	Directeur

Université Paris 8
Laboratoire d'informatique avancée de Saint-Denis (LIASD)
Laboratoire Paragraphe
2 rue de la Liberté - 93526 Saint-Denis cedex - France

Remerciements

Je remercie chaleureusement toutes les personnes qui m'ont aidé pendant l'élaboration de ma thèse et notamment mon directeur Monsieur Arab Ali Cherif, pour son intérêt, son soutien et ses nombreux conseils durant la rédaction de ma thèse.

Je remercie également M. Raja Chatila et Mme. Nicoleta Rogovschi qui m'ont fait l'honneur d'être rapporteurs de ma thèse et Mme. Lynda Seddiki, Mme. Hanene Azzag et M. Zreik Khaldoun d'avoir accepté de participer à mon jury de thèse.

Ce travail n'aurait pas été possible sans le soutien de l'Université Paris 8, de l'UFR STN et le laboratoire LIASD, qui m'ont permis, grâce à un poste ATER et diverses aides financières, de me consacrer sereinement à l'élaboration de ma thèse.

Ce travail n'aurait pu être mené à bien sans la disponibilité et l'accueil chaleureux que m'ont témoignés le personnel du laboratoire LIASD, Anna Pappa qui m'a aidé à la rédaction de mon premier article, Nicolas Jouandeau pour la relecture de mes articles. Je te tiens aussi à remercier tous les enseignants-chercheurs du laboratoire LISAD qui m'ont apporté leur aide, leur soutien et m'ont enrichi avec les nombreuses discussions et conseils qui m'ont accompagné tout au long de mon cursus.

Un grand merci aussi à tous les membres l'UFR STN pour leur gentillesse et leur bienveillance. Je remercie particulièrement Mehdy Tounsi qui m'a beaucoup aidé et soutenu durant cette thèse.

Il m'est impossible d'oublier Rym et Léa qui me sont chères et que j'ai quelque peu délaissées durant les longues années qu'a duré cette thèse. Les attentions et encouragements de Rym m'ont accompagnée tout au long de ces années. Je lui reste redevable pour son soutien moral et sa confiance indéfectible dans mes choix. Enfin, j'ai une pensée toute particulière pour ma grand-mère .

Résumé

Les travaux décrits dans le cadre de cette thèse portent sur la réalisation d'un robot mobile pour la navigation en milieu domiciliaire dans un milieu congestionné par de multiples objets/obstacles qui peuvent être en mouvement ; dans cette proposition, l'environnement et le contexte sont inconnus. L'objectif consiste à proposer une architecture robotique permettant la navigation parmi des obstacles fixes, amovibles et interactifs. L'objectif du robot est de rejoindre une position définie par l'utilisateur, tout en évitant les obstacles fixes, déplacer les obstacles amovibles s'ils gênent le passage ou bien demander à des obstacles interactifs (humain, robots, etc.) de céder le passage. Dans la littérature, cette problématique est connue sous le nom de la *la navigation parmi les obstacles amovibles*. Dans le cadre de notre recherche, nous avons élaboré les trois contributions suivantes :

La première contribution est l'élaboration d'une architecture robotique hiérarchique baptisée VICA (VIcarious Cognitive Architecture), dont le niveau décisionnel est couplé à une architecture cognitive. Pour cela, nous nous sommes inspiré des travaux sur la simplixité de Alain Berthoz qui décrivent comment le vivant prépare l'action et anticipe les réactions, notamment dans les situations complexes (gestion des obstacles). L'architecture robotique se compose d'un planificateur global permettant la navigation dans un environnement inconnu et d'un planificateur local dédié à la gestion des obstacles.

La seconde met en œuvre un planificateur global dont le but est de rapprocher autant que possible le robot de son objectif. La solution proposée se base sur un algorithme que nous avons développé, connu sous le nom de H^*.

La troisième propose un planificateur local pour la gestion des obstacles. La solution proposée consiste à utiliser la simulation multi-agents dans le but d'anticiper le comportement des obstacles en relation avec les actions du robot afin de déterminer par avance la meilleure action à entreprendre. Les résultats de la simulation sont comparés aux données récoltées pendant l'action dans le but de vérifier si l'action effectuée est conforme aux simulations tout en élaborant une base d'apprentissage pour les actions futures.

Les expérimentations réalisées ont pour but de montrer les possibilités et les limites de notre vision de la version informatique de la simplixité. Par la suite, elles ont permis de valider les approches proposées dans le cadre de la navigation parmi les obstacles amovibles. L'implémentation de cette solution est réalisée dans l'architecture VICA (VIcariance Cognitive Architecture) développée sous ROS (Robot Operating System). En parallèle, nous avons développé un robot expérimental pour valider nos résultats.

Mot-clefs : Robotique mobile, navigation parmi les obstacles amovibles (NAMO : Navigation Among Movable Obstacles), architectures cognitives, architectures robotique, planification de mouvement, systèmes multi-agents.

Abstract

The work carried out in this thesis aims at the implementation of a mobile robot for home navigation, for domestic and daily use, congested by multiple objects / obstacles that may be in motion; In this proposal the environment and the context are unknown. The objective is to propose a robotic architecture allowing navigation among fixed, removable and interactive obstacles. The goal of the robot is to reach a user-defined position, while avoiding fixed obstacles, moving removable obstacles if they obstruct the passage, or asking interactive obstacles (humans, robots, etc.) to give way.

In this thesis, it is a question of determining a hierarchical robotic architecture, whose decisional level is coupled with a cognitive architecture. This is inspired by Alain Berthoz's work on simplicity, which describes how living beings prepare the action and anticipate their consequences, especially in complex situations (obstacle management). The robotic architecture consists of a global planner allowing navigation in an unknown environment and a local planner dedicated to the management of obstacles.

The second contribution proposes a local planner for the management of obstacles. The proposed solution consists of using multi-agent simulation in order to anticipate the behaviour of the obstacles in relation to the actions of the robot in order to determine in advance the best action to be taken. The results of the simulation are compared with the data obtained during the action in order to verify whether or not the action taken is in line with the expectations.

The last contribution proposes a global planner in order to bring the robot as close as possible to its goal. The proposed solution is based on an algorithm that we have developed, known under the name of H^*.

The implementation of this solution is carried out in the VICA architecture (VIcariance Cognitive Architecture) developed under ROS (Robot Operating System). In parallel, an experimental robot was developed to validate the experiments. The experiments carried out are intended to show the possibilities and limits of our vision of the computer version of simplixity. Subsequently, they validate the proposed approaches in the context of navigation among removable obstacles.

;o

Table des matières

1 Introduction

1.1 Problématique générale

Depuis la création du premier robot mobile autonome[1], la navigation en milieu congestionné tel que les milieux domiciliaires, reste un problème ouvert, malgré les avancées dans ce domaine. Nous avons donc décidé de nous intéresser dans cette thèse à l'élaboration d'une architecture robotique permettant à un robot de naviguer dans un environnement typiquement domiciliaire. Un tel environnement présente les caractéristiques suivantes : congestionné[2], dynamique[3], fréquenté par des humains[4] ou des animaux domestiques[5] et il est inconnu[6]. L'un de nos objectifs est de proposer une architecture robotique autonome permettant à un robot mobile de rejoindre une position (but) définie par l'utilisateur en suivant un chemin optimal en toute sécurité. L'architecture robotique proposée dans cette thèse offre au robot la capacité de déterminer un chemin optimal dans un tel environnement tout en évitant ou en déplaçant les obstacles[7] qui gênent le passage.

La problématique que nous abordons ici relève de la « Navigation parmi les obstacles amovibles » connue sous l'acronyme **NAMO** pour *Navigation Among Movable Obstacles*. Pour valider nos résultats nous proposons un cas pratique dans lequel nous mettons en œuvre un robot mobile qui évolue dans un milieu de bureau, nous donnons pour objectif au robot d'aller d'une position initiale A pour rejoindre une position finale B définie par l'utilisateur. Nous ajoutons comme contrainte, un environnement complexe, composé d'obstacles statiques et dy-

[1] Le premier robot mobile autonome nommé Shakey a été développé dans les laboratoires de l'université de Stanford en 1967, il est capable de se déplacer de manière autonome d'un point à un autre en évitant les obstacles.

[2] Forte présence d'obstacles susceptibles de bloquer le chemin du robot.

[3] Obstacles en mouvement, la configuration de l'environnement évolue dans le temps.

[4] Obstacle dynamique, fragile et interactif : le robot doit éviter tout contact avec les humains afin de se prémunir de tout risque de blessure, mais il est possible d'interagir avec les humains.

[5] Obstacles dynamiques et fragiles et sans possibilités de communication.

[6] Le robot ne dispose pas de carte de l'environnement au préalable.

[7] Le robot déplace les obstacles en les poussant.

namiques (robots, humains, d'autres obstacles tels que du mobilier de bureau). Le cas qui nous intéresse dans notre démarche, est que l'objectif B ne soit pas accessible par simple évitement d'obstacles comme on peut le constater dans les travaux classiques de navigation. En revanche, l'objectif est souvent plus compliqué à atteindre, il faut parfois déplacer des objets ou demander à d'autres agents de laisser le passage. Dans ces situations, le robot doit être autonome, c'est-à-dire être en mesure de spécifier lui-même ses sous-objectifs ainsi que les moyens pour y parvenir. Afin de réaliser un tel système, nous nous inspirons des architectures cognitives pour concevoir une architecture robotique. Donc, notre objectif est de développer des modèles et des techniques informatiques inspirées par le comportement humain souvent appelées « Architectures cognitives » pour permettre à un robot mobile de se déplacer de manière autonome dans un environnement domiciliaire congestionné.

L'objectif défini par l'utilisateur (**but**) est un élément déterminant dans le processus d'exécution d'instructions que le robot doit mener. Il permet d'initier tous les processus et de les paramétrer afin qu'il puisse répondre à un besoin. Dans notre étude, nous avons fait le choix suivant : les instructions de l'utilisateur sur le but à atteindre, est composée de deux informations, 1) une indication de direction (cap) et 2) une indication d'objet à trouver dans cette direction. À terme, cette technique permettra à l'utilisateur de montrer du doigt une direction et d'indiquer le nom de l'objet vers lequel le robot doit se diriger. Il est possible aussi d'indiquer un cap et une distance (coordonnées polaires).

Les domaines d'application de la NAMO sont nombreux, elle peut par exemple trouver son application :

- Dans les maisons de retraites pour assister les personnes en situation de mobilité réduite pour trouver et/ou d'apporter des objets, malgré un environnement qui est susceptible d'être encombré par différents objets ou personnes.
- Améliorer l'efficacité des robots aspirateurs autonomes, pour les doter de capacités de bouger les obstacles qui bloqueraient leur passage.
- Améliorer l'autonomie des robots en milieux hostiles tel que les sites contaminés par la radio-activité, la recherche de personnes dans les décombres, déminage en environnement intérieur, etc.

Dans la suite de ce chapitre, nous allons présenter le domaine de la robotique ainsi que les travaux qui traitent de la même problématique. Nous terminerons ce chapitre par une conclusion détaillant nos motivations portant sur la navigation parmi les obstacles amovibles et leurs applications dans un environnement domiciliaire.

1.2 Introduction à la robotique de service

Au début des années 1960, les robots à valeur ajoutée, on fait leur apparition dans les usines pour effectuer des tâches simples et à fort risque (Figure 1.1), ces robots sont généralement confinés dans un espace réservé, où l'environnement est connu et contrôlé, ils n'interagissent pas directement avec des agents humains pour des raisons de sécurité. De nos jours, les robots ont été de plus en plus utilisés comme par exemple sur les chaînes de montage pour automatiser les tâches et réduire les coûts. Ces deux dernières décennies ont vu l'apparition d'un nouveau

genre de robots, contrairement à la robotique industrielle, cette nouvelle robotique a pour but d'assister les humains dans leurs tâches et non pas à exécuter des tâches de façon automatique et purement isolée, c'est la **robotique de service**.

FIGURE 1.1 : Le premier robot industriel **Unimates** installé dans l'usine de General Motors en 1961, permet d'effectuer le déchargement d'une machine de moulage sous pression de pièces en acier.

Depuis, la robotique industrielle n'a cessée de gagner du terrain jusqu'à l'apparition d'usines complètement robotisées (Figure 1.2). La réussite de la robotique industrielle contrairement à la robotique de service est due à la simplicité de l'environnement dans lequel les robots évoluent et aux budgets consacrés par les industriels à cette discipline qui sont assez conséquents. Construire un robot dans un environnement confiné et contrôlé relève plus du domaine de l'automatisation que de la robotique, car les différentes configurations du robot son dénombrables et les configurations de l'environnement sont contrôlées et prévisibles comme on peut le voir par exemple dans les Figures 1.3 et 1.4.

La **robotique de service** vise à créer des robots qui fonctionnent dans un environnement non (ou partiellement) contrôlé, dans le but d'aider des êtres humains dans un travail pénible (dangereux, répétitif et ennuyeux, sale, etc.). La Fédération internationale de la robotique (www.ifr.org) donne une définition ainsi que quelques exemples d'utilisation de robots de service[8]. Elle les classe en deux catégories :

- **Robot de service à usage professionnel**, défini comme suit : « Un robot de service professionnel ou un robot de service à usage professionnel est un robot de service utilisé pour une tâche commerciale, généralement exploité par un opérateur qualifié. ».
  Exemple d'utilisation :
  - Robot de nettoyage pour lieux publics.
  - Robot de distribution dans les bureaux ou les hôpitaux.
  - Robot de lutte contre les incendies.

[8]ISO 8373 : 2012 Robots and robotic devices - Vocabulary (http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=55890.).

FIGURE 1.2 : L'usine **Changying Précision Technologie** en Chine, entièrement robotisée, l'usine compte 650 ouvriers avant la robotisation et ne compte plus que 60 ouvriers après, soit une réduction 90% des effectifs. Le processus de fabrication est automatisé à 80% (2015).

FIGURE 1.3 : Entrepôt de l'épicier **Ocado**, les robots mobiles circulent grâce à un algorithme centralisé de contrôle de trafic pour éviter les collisions. Les robots ont pour mission de préparer les commandes en-ligne.

  - Robot de rééducation.
  - Robot de chirurgie dans les hôpitaux, etc.

  Ces robots présentent généralement une interface pour qu'une personne qualifiée puisse les piloter et les arrêter en cas d'urgence, quelques exemples illustrés dans la Figure 1.5.

- **Robot de service à usage personnel** défini comme suit : « Un robot de service personnel ou robot de service à usage personnel est un robot de service utilisé pour une tâche non-commerciale, généralement par des profanes. ».
  Exemple d'utilisation :

  - Robot domestique, compagnon.
  - Fauteuil roulant automatisé.
  - Robot d'assistance à la mobilité personnelle, etc.

FIGURE 1.4 : Premier centre de distribution robotisé de la société **Amazone** à Brétigny-sur-Orge en France (2014), les robots ont permis de réduire le temps d'acheminement des colis à 13 minutes contre une heure et demie en moyenne auparavant.

Ces robots sont généralement autonomes et nécessitent une intervention minime de l'utilisateur, quelques exemples illustrés dans la Figure 1.6.

En 2013, le pôle interministériel de prospective et d'anticipation des mutations économiques (PIPAME) a conduit une étude sur le développement futur de la robotique de service personnel [PIPAME, 2012] en France. Les prévisions énoncées par le PIPAME prévoient un fort développement de ce secteur comme l'indique le graphe illustré dans la Figure 1.7, l'unité est indiquée en valeur base 100 en 2008[9]. Cependant, il est difficile de confirmer les chiffres prédits au moment de la rédaction de ce rapport en raison du peu d'études statistiques récentes. Mais il semble que le secteur n'a pas connu les résultats escomptés, en raison de limites technologiques.

L'objectif de la robotique de service personnelle vise à l'élaboration d'une catégorie de robots permettant d'accomplir des tâches complexes, avec une intervention minimale de l'utilisateur. Ces robots partagent le même espace physique que leurs utilisateurs, donc il est impératif que ces robots soient dotés d'un niveau élevé de sécurité afin d'éviter les collisions avec les personnes ou endommager les objets qui se trouvent sur leurs trajectoires. Les solutions apportées par la plupart des robots actuellement présents sur le marché (par exemple les robots aspirateurs, fauteuils roulants, etc.) se limitent généralement à l'évitement d'obstacles et ils sont souvent utilisables uniquement dans des environnements préalablement cartographiés et connus, néanmoins les recherches dans ce domaine restent très actives, dans le but d'améliorer leur interaction avec les obstacles et d'optimiser leur fonctionnement dans des environnements non contrôlés. La Figure 1.8 montre quelques exemples de robots de services expérimentaux.

[9]Définition INSEE : L'indice d'une grandeur est le rapport entre la valeur de cette grandeur au cours d'une période courante et sa valeur au cours d'une période de base. Il mesure la variation relative de la valeur entre la période de base et la période courante. Souvent, on multiplie le rapport par 100 ; on dit : indice base 100 à telle période.

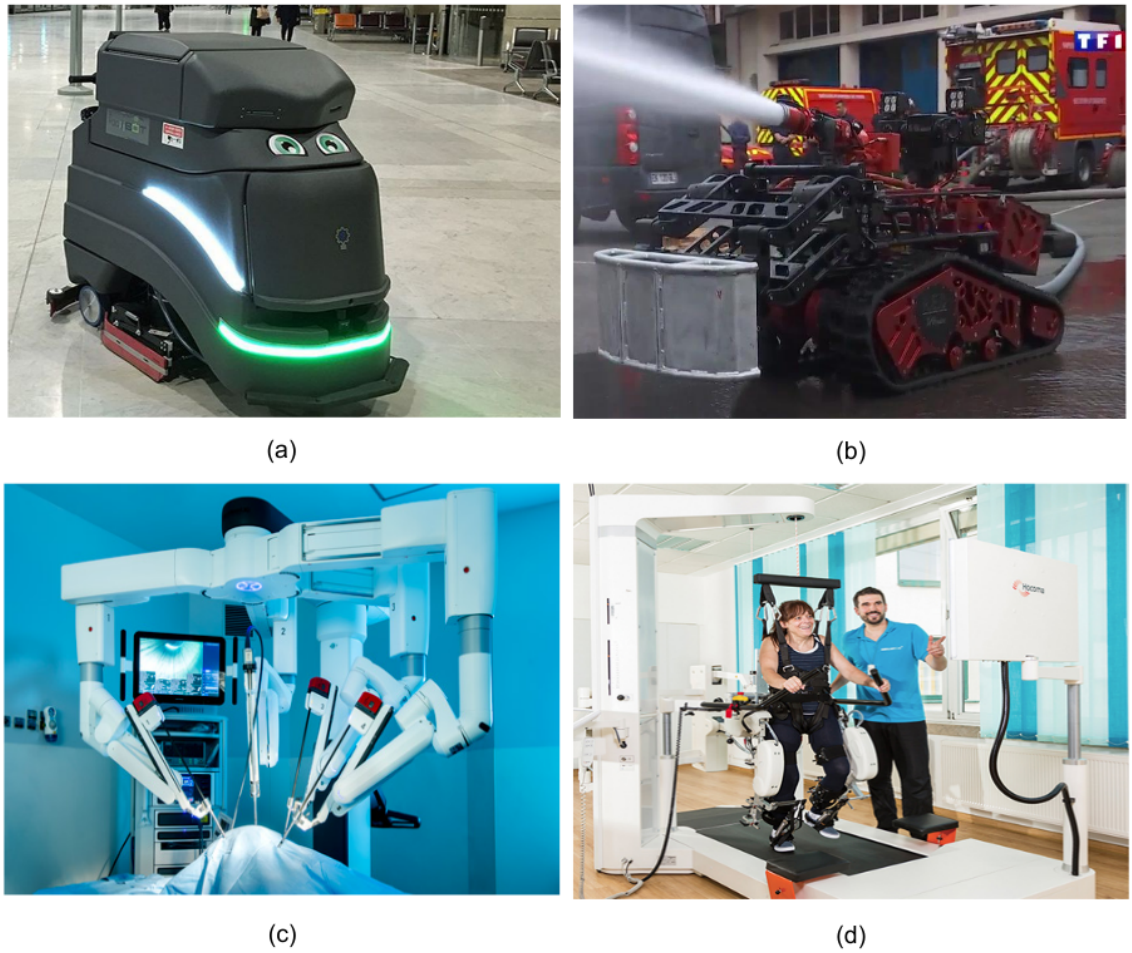


FIGURE 1.5 : Quelques exemples de robots de services professionnels : (a) Robot de nettoyage dans l'aéroport Roissy-CDG, le nettoyage se fait la nuit pour minimiser les interactions avec les voyageurs (b) Robot des pompiers de Paris **Colossus**, intervenu lors de l'incendie de Notre-Dame de Paris (c) Robot de chirurgie **Da Vinci XI** à l'Infirmerie Protestante à Lyon (d) Robot pour réapprendre à marcher **Lokomat**.

L'un des défis important que tente de relever la robotique de service personnelle est la navigation dans un environnement typiquement domiciliaire avec des obstacles de nature différente qui peuvent être fixes, en mouvement (tel que les humains ou des robots), fragiles (on risque d'abîmer et de blesser si on les manipule), amovibles (on peut les déplacer), etc. Dans le milieu de la recherche, ce domaine est connu sous le nom de la navigation parmi les obstacles amovible connu aussi en anglais sous l'acronyme **NAMO** : *Navigation Among Movable Obstacles*, dans la suite de ce manuscrit, nous allons utiliser cet acronyme pour se référer à cette discipline.

La robotique de service personnelle a pour objectif de faire quitter aux robots les laboratoires pour qu'ils s'installent dans nos maisons. Les utilisateurs s'attendent alors à ce que les robots réagissent à des instructions de haut niveau tel que « S'il vous plaît, ouvrez la porte » ou « S'il vous plaît, venez à la cuisine ». Compte tenu de la commande, il est prévu que le robot soit autonome et exécute le mouvement en toute sécurité dans un environnement domiciliaire, on attend aussi que son comportement soit proactif, c'est-à-dire que le robot doit résoudre tout seul les problèmes qui l'empêcheraient d'atteindre son objectif. Le domaine de la NAMO,

FIGURE 1.6 : Quelques exemples de robots de services à usage personnel : (a) Robot aspirateur Roomba, (b) Prototype d'une fauteuil roulant robotisé présenté par la société japonaise TOYOTA en 2017, (c) Le robot humanoïde NAO commercialisé par la société SoftBank Robotics participe chaque année à la coupe du monde de robotique (RobotCup), (d) le robot Pepper qui assiste le présentateur *Jérôme Bonaldi* dans l'émission *Mag de la Science*.

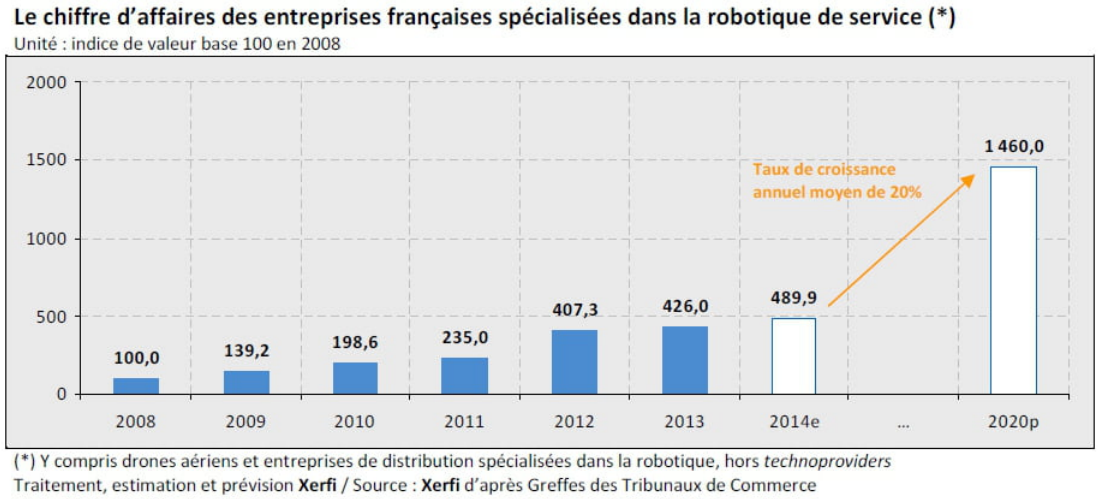


FIGURE 1.7 : Marché de la robotique de service.

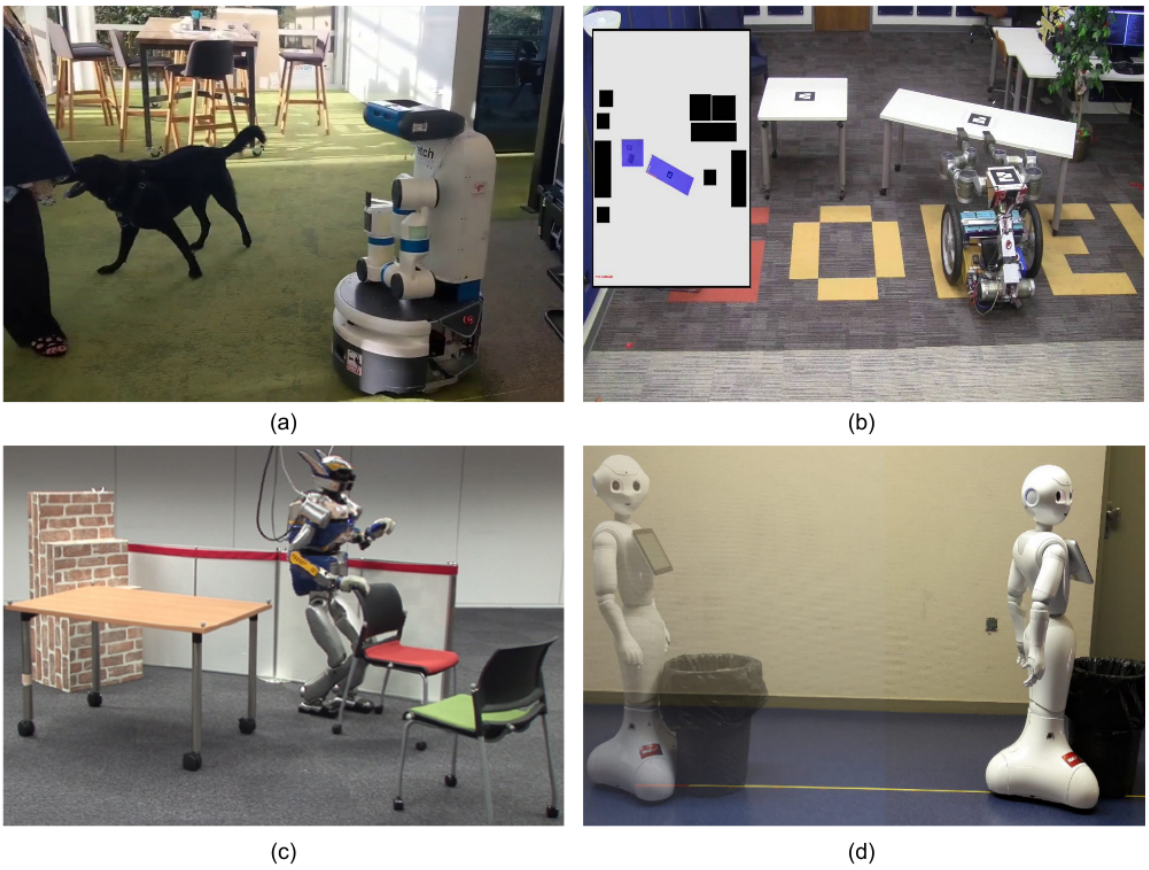


FIGURE 1.8 : Quelques exemples de robots de services expérimentaux : (a) Navigation d'un robot dans un environnement avec des obstacles dynamiques, le robot utilise l'apprentissage artificiel pour planifier son mouvement [Faust et al., 2018], (b) Robot qui manipule les obstacles pour se frayer un passage, le robot est assisté par une caméra en hauteur qui lui indique une vue globale de l'environnement [Scholz et al., 2016], (c) Robot humanoïde HRP-2 dans un environnement domiciliaire, qui tente d'écarter la chaise qui lui bloque le passage, on peut constater que les obstacles sont placés sur des roulettes pour faciliter leur déplacement [Stilman et al., 2007], (d) Robot Pepper développé dans le cadre du choix du placement social d'objets, ici, on peut voir le robot qui tente de mettre la poubelle dans la cuisine par exemple au lieu de la mettre dans le salon, ce qui est socialement acceptable par un humain [Renault et al., 2019].

tente d'apporter une réponse afin qu'un robot puisse trouver les solutions aux problèmes qui l'empêcheraient d'exécuter les instructions de l'utilisateur. Par exemple, si un humain veut quitter le salon pour se rendre à la cuisine, mais qu'une chaise bloque l'unique passage, il la déplacera au lieu de déclarer qu'il n'est pas possible d'aller dans la cuisine. Hors, la plupart des robots de nos jours se contentent d'indiquer qu'il est impossible d'effectuer la tâche. De plus, même s'il existait un autre chemin pour aller à la cuisine, les humains pourraient déplacer la chaise plutôt que de faire un long détour, une fois encore, contrairement aux robots. La décision humaine de savoir quand déplacer la chaise lors du précédent exemple plutôt que de faire un détour est guidée par l'effort nécessaire dans l'une ou l'autre option. Par exemple, cette notion peut être décrite par une fonction de coût, ayant un coût pour la manipulation et un autre pour la navigation. Sur la base de cette fonction de coût, il convient de choisir les options optimales ou les efforts les plus faibles. La réalisation de tels robots nécessite un traitement

logiciel complexe, en raison de la multiplicité des tâches de différents types qu'elle implique (navigation, traitement des obstacles, interaction avec l'utilisateur, etc.). Généralement lors de la présence de divers fonctionnalités elle sont réalisées dans un ensemble nommé *architecture robotique*.

1.3 Motivations

La robotique personnelle de service a pour objectif de contribuer à améliorer le cadre de vie dans les espaces de travail et dans nos maisons, en particulier pour assister les personnes âgées ou avec une autonomie réduite. L'étude menée par le PIPAME a montré que cette branche de la robotique est prometteuse, mais qu'il reste un vide à combler en terme de recherche, nous pensons qu'il est important d'explorer ce domaine et d'apporter notre contribution.

La problématique posée par la robotique personnelle de service implique différentes compétences auxquelles le robot doit répondre, on peut citer par exemple : La navigation, la manipulation des objets, la communication avec les utilisateurs. Dans ce travail, nous nous intéressons uniquement à la première brique, que nous pensons fondamentale : *La navigation.*

1.4 Contribution de la thèse

Le travail présenté dans ce manuscrit porte sur la réalisation d'une architecture robotique pour la NAMO. L'objectif étant de proposer et développer une solution permettant à un robot mobile de se déplacer dans un environnement domiciliaire. Le but étant de partir d'une position initiale afin de rejoindre une position finale donnée par un utilisateur sous forme d'une direction et d'un objectif à atteindre. Par exemple, l'ordre de l'utilisateur peut être de la forme suivante : « trouver le téléphone dans cette direction » ou tout simplement appeler le robot pour qu'il puisse rejoindre l'utilisateur, dans ce cas le robot doit être capable de détecter approximativement la direction de l'endroit d'où est venu l'ordre.

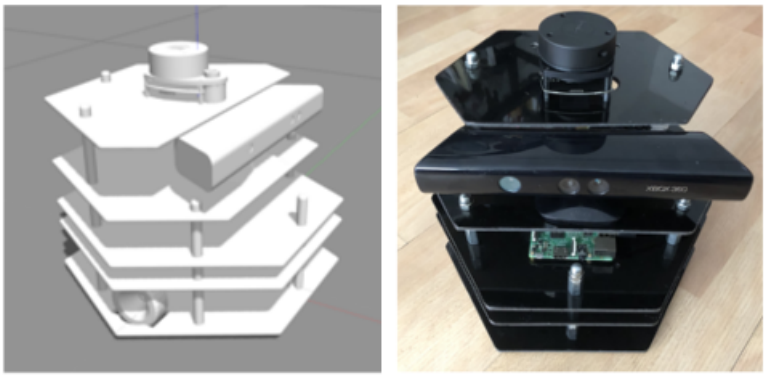


FIGURE 1.9 : Version du robot mobile utilisée lors des premières simulations dans le cadre de cette thèse, à gauche une version logicielle du robot pour les simulations à l'aide du logiciel *Gazebo* et à droite le robot que nous avons conçu pour les expérimentions en environnement réel.

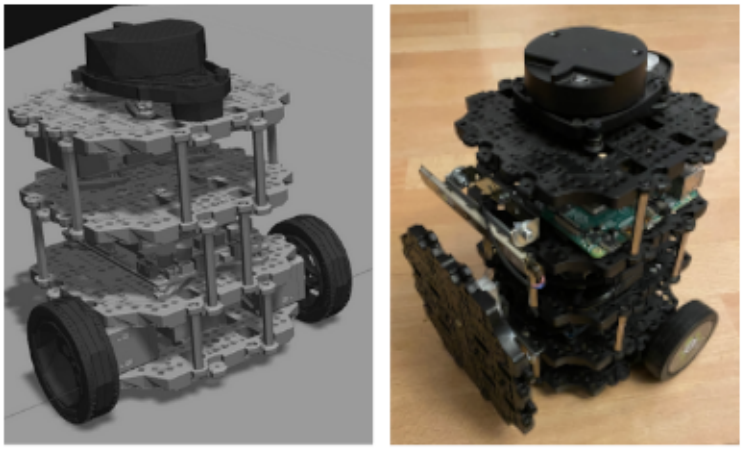


FIGURE 1.10 : Version du robot mobile utilisée plus récemment. Il s'agit d'une plateforme robotique commerciale connue sous le nom *Turtlebot Burger*.

Le robot mis en œuvre dans cette thèse (Figure 1.9 et 1.10) a pour but d'évoluer dans un environnement domiciliaire qui nous impose les contraintes suivantes :

- *1) Environnement dynamique et inconnu* : c'est-à-dire que le robot ne dispose pas de connaissances préalables sur l'environnement, même si le robot dispose d'une connaissance partielle sur l'environnement cette dernière peut évoluer dans le temps.
- *2) Environnement congestionné :* un certain nombre d'obstacles peuvent obstruer le passage du robot, la position finale (but) n'est pas toujours accessible sans déplacer certains obstacles dans certaines configurations.
- *3) Environnement fréquenté par des humains et des animaux :* le robot ne peut pas déplacer tous les obstacles en raison de leurs natures (fragilité), il doit être capable de communiquer pour demander par exemple à un humain de se déplacer afin de lui laisser le passage.

La Figure 1.11 illustre un exemple typique du résultat qu'on cherche à atteindre dans ce travail et les différentes étapes que le robot doit passer pour atteindre son but dans un environnement de bureau avec des obstacles de différentes natures.

Cette thèse présente trois contributions, la première consiste à proposer un planificateur global permettant à un robot mobile de trouver un chemin le rapprochant le plus possible de son but, dans un environnement dynamique et inconnu. La seconde contribution se concentre sur la gestion des obstacles, afin de trouver un passage dans des situations de blocage, par exemple, le chemin est obstrué par des obstacles, pour ce faire nous nous inspirons des sciences cognitives et de l'intelligence artificielle pour réaliser un système basé sur la simulation du mouvement avant son exécution. Le robot réalise en interne une représentation de la scène observée dans un Système Multi-Agents, chaque obstacle est représenté par un agent, par la suite, une simulation des actions possibles est réalisée afin de choisir la meilleure action, celle qui lui permettra de se rapprocher un peu plus près de son but. Lors de la réalisation de l'action, le robot utilise ses capteurs afin de comparer avec les résultats prédis par la simulation à des fins d'apprentissage. La troisième contribution, consiste à développer une architecture robotique hiérarchique qui

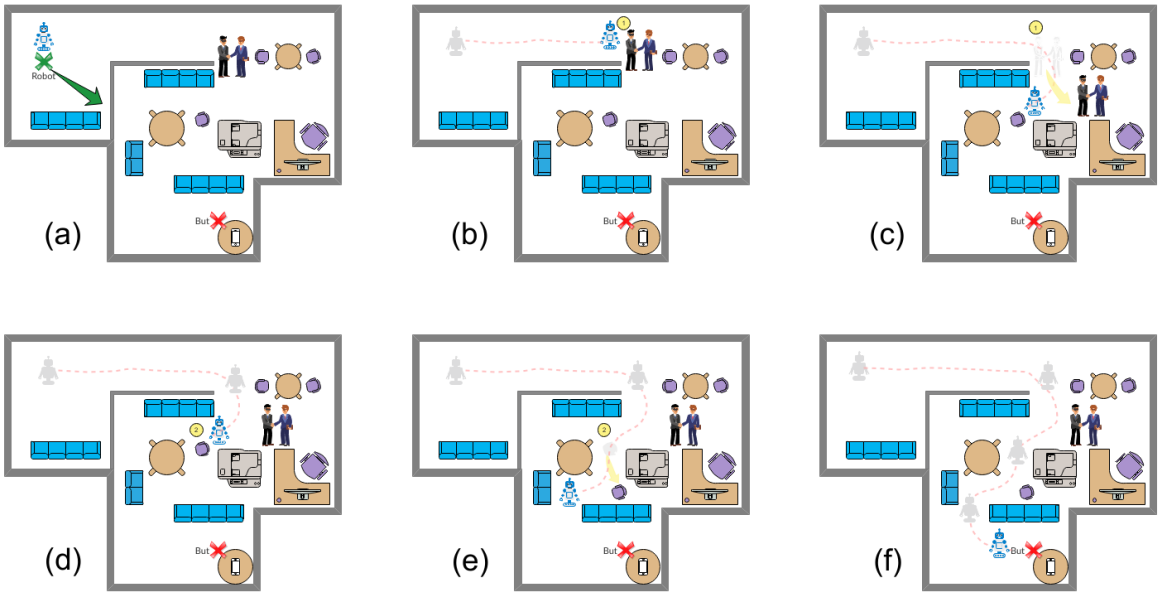


FIGURE 1.11 : Exemple (a) Le robot reçoit les instructions de l'utilisateur, une direction approximative et l'objet à trouver (ici le téléphone sur la table). (b) Le robot doit emprunter le seul chemin qui le rapproche le plus possible de son but. (c) La gestion des obstacles ici consiste à demander aux personnes de lui céder le passage. (d) Le robot est de nouveau confronté à des obstacles. (e) Le choix du robot doit être de déplacer la chaise plutôt que la table. (f) Le robot a atteint son objectif.

regroupe les deux premières contributions pour présenter un ensemble cohérent et effectuer des simulations. Cette architecture robotique est implémentée sous ROS (Robot Operating System), les expérimentations sont conduites dans un premier temps avec le simulateur de robot Gazebo, par la suite, une implémentation d'un robot physique est réalisée afin d'effectuer des expérimentations grandeur nature.

1.5 Conclusion

Nous avons débuté ce chapitre par un historique de la robotique qui a abouti à une séparation de cette discipline en deux parties : la robotique industrielle et la robotique personnelle de service. Dans les années à venir, les chercheurs de ce domaine ont pour objectif de développer des robots capables d'opérer dans un environnement conçu pour l'homme et d'interagir avec celui-ci. Le PIPAME a identifié de nombreuses contraintes parmi lesquelles on peut citer par exemple :

- 1) Les robots doivent agir en présence de l'humain sans solliciter son attention.
- 2) Une certaine autonomie dans les déplacements.
- 3) Robustesse et sécurité permettant le fonctionnement en présence d'un public éventuellement large, et souvent non-professionnel.

Notre démarche s'inscrit dans cette optique et tente de répondre à une partie de ces besoins. La première difficulté apparente pour un robot mobile dans un tel environnement est la na-

vigation. Malgré de nombreux travaux dans ce domaine, à notre connaissance, actuellement, il n'existe pas de solutions robustes et efficaces pour la navigation parmi les obstacles amovibles. Pour cela, nous nous intéressons dans ce travail à l'élaboration d'un modèle et d'une implémentions permettant à un robot de naviguer dans un tel environnement.

Dans le domaine de la navigation parmi les obstacles amovibles, les êtres vivants dotés d'une intelligence, semblent résoudre le problème de la navigation avec peu d'efforts, nous souhaitons dans ce travail s'inspirer des mécanismes mis en œuvre par le vivant, pour les implémenter dans un robot. De nombreux travaux en sciences cognitives tentent d'expliquer le fonctionnement de la navigation chez les êtres vivants. Dans ce travail, nous tentons de proposer une implémentation d'un modèle de fonctionnement qui se repose sur la simulation de mouvements, ce modèle est proposé par le physiologiste Alain Berthoz. Pour ce faire, nous avons utilisé les Systèmes Multi-Agents pour réaliser une modélisation de l'environnement et la simulation des mouvements du robot.

Le robot mis en œuvre dans le cadre de cette thèse utilise principalement deux mécanismes, le premier lui permet de naviguer en évitant les obstacles pour se rapprocher de son objectif, le second permet au robot de gérer les obstacles, c'est-à-dire, déterminer la nature des obstacles immédiats et engager une stratégie pour dégager un passage, soit en poussant les obstacles ou en demander de céder le passage. Les deux mécanismes sont supervisés par un système permettant de déterminer lequel des deux planificateurs engager pour assurer une navigation optimale.

Les résultats que nous avons obtenus, ont permis de franchir une nouvelle étape, dans la modélisation de l'état interne des robots, qui est l'*abstraction*, on peut même dire que le robot imagine le mouvement avant d'agir. Ainsi, il est capable de prédire l'état futur de l'environnement et agir en conséquence. La prédiction se fait en simulation et non pas en utilisant des mécanismes d'apprentissage artificiel qui nécessitent un nombre important de données d'apprentissage.

Le travail proposé dans cette thèse reste une ébauche d'une solution de navigation robuste et efficace, mais il reste encore à résoudre les problèmes liés à la reconnaissance du rôle de chaque objet dans leur environnement (l'affordance[10]), ce qui permet encore plus d'autonomie.

Organisation du rapport

Ce rapport est organisé en 6 chapitres :

- Le Chapitre 1 constitue une introduction générale qui présente la robotique personnelle de services, les domaines d'applications et l'intérêt de ce domaine pour la société.
- Le Chapitre 2 donner un état de l'art non-exhaustif de la *Navigation Parmi les Obstacles Amovibles* qui tente de répondre à ces questions dans un environnement fréquenté par des humains, et présenter les travaux déjà effectués dans ce domaine. Par la suite, il décrit comment les composants logiciels sont organisés entre eux pour apporter une réponse satisfaisante aux questions soulevées par la NAMO en présentant les architectures robotiques et cognitives susceptibles de répondre à ce besoin. Pour finir, il présente le concept

[10]L'affordance est l'ensemble des caractéristiques d'un objet qu'on peut utiliser pour réaliser une action.

de la *Vicariance* et de *Simplixité*, utilisés dans ce travail pour élaborer un planificateur pour la NAMO.

- Le Chapitre 3 Commence par poser la problématique de la NAMO d'une façon formelle, et présente les différents matériels et méthodes que doit implémenter un robot mobile pour répondre au mieux à la problématique de la NAMO en environnement domiciliaire. Par la suite, il propose une architecture cognitive inspirée des travaux effectués en physiologie pour trouver une application en robotique mobile de service et propose un modèle d'une architecture robotique implémentable.

- Le Chapitre 4 expose les stratégies de navigation présentées dans la littérature afin de déterminer la meilleure stratégie applicable à la NAMO. Par la suite, il présente notre planificateur de mouvement global pour déterminer les chemins et les trajectoires adaptées à la NAMO.

- Le Chapitre 5 décrit un planificateur local pour la gestion des obstacles ainsi que la représentation interne que fait le robot de son l'environnement.

- Le Chapitre 6 présente une conclusion et les perspectives à venir.

2
État de l'art

Introduction

Concevoir un robot capable de naviguer dans un environnement congestionné et déplacer les obstacles qui lui bloquent le passage pour se frayer un chemin, est un domaine auquel s'intéressent les roboticiens depuis plusieurs décennies, ce domaine est connu sous le nom de la navigation parmi les obstacles amovibles (*NAMO : Navigation Among Movable Obstacles*). La NAMO est un domaine de recherche important dans la planification de mouvements en milieu congestionné, car il donne aux robots mobiles une meilleure capacité de raisonnement sur l'environnement et les possibilités de choisir les obstacles à manipuler, afin de se frayer un passage [Stilman and Kuffner, 2005]. La NAMO permet donc de résoudre des problèmes qui sont difficiles ou voir impossibles à résoudre avec une méthode d'évitement d'obstacles classiques. La première partie de ce chapitre est consacrée à la présentation des différentes approches et solutions proposées par la NAMO dans la littérature.

La gestion des obstacles n'est pas une finalité pour un robot, mais le but est d'atteindre une position finale à moindre coût[1]. Il est donc essentiel que le robot fasse une planification globale de sa trajectoire et de faire appel aux différentes techniques adéquates aux types de problèmes rencontrés. Pour permettre cela, il est important de mettre en place un logiciel de haut niveau qui permet de raisonner sur un aspect global et de choisir les planificateurs adéquats, en même temps interagir avec les capteurs et les actionneurs qui utilisent des logiciels bas niveau. Dans le but de faciliter cette tâche et de séparer les couches bas-niveau / haut-niveau, on utilise généralement les architectures robotiques. La seconde partie de ce chapitre est consacrée à la présentation des différents types d'architectures robotiques.

La partie décisionnelle d'une architecture robotique, permet d'établir les choix de stratégies, ainsi l'efficacité de celle-ci dépend entièrement de la stratégie choisie. Le choix décisionnel d'un agent autonome dépend essentiellement de sa capacité à modéliser son environnement pour une

[1]La notion de coût peut être différente selon les applications, elle est exprimée généralement en quantité de temps, de distance, d'énergie, etc.

prise de décision efficace. Les architectures cognitives sont développées principalement dans ce but, dans notre travail, nous proposons de coupler la partie décisionnelle de l'architecture robotique avec une architecture cognitive. La dernière partie de ce chapitre sera consacrée à l'étude de ces modèles et leurs utilisations en robotique.

2.1 La NAMO

2.1.1 Analyse des travaux existants sur la NAMO

Les robots seraient beaucoup plus utiles et efficaces s'ils pouvaient déplacer les obstacles qui gênent leur passage, comme nous l'avons montré dans le premier chapitre de ce rapport. Les études sur la problématique de navigation des robots mobiles impliquant des obstacles mobiles remontent au début des années 90 [Wilfong, 1991]. Cette problématique et connue actuellement sous l'acronyme **NAMO**.

Les travaux actuels dans le domaine NAMO peuvent être divisés en deux grandes catégories : *planification hors ligne* et *planification en ligne*. La planification hors ligne suppose que l'ensemble des informations de l'espace dans lequel se déplace le robot est connu à l'avance, contrairement à la planification en ligne, le robot ne dispose que d'une connaissance partielle de son environnement, et qu'il peut modifier son plan initial en fonction de nouvelles informations acquises lors de son déplacement. Dans la plupart des travaux antérieurs qui se sont concentrés sur les NAMO hors ligne, les résultats montrent que cette approche n'est pas efficace. La tendance actuelle se tourne vers la NAMO en ligne, qui semble plus prometteuse et mieux adaptée.

Le premier article traitant de la NAMO hors ligne [Chen and Hwang, 1991] permet de gérer plusieurs objets mobiles dans un environnement à informations complètes. Il se base sur un planificateur heuristique pour générer dans un premier temps une série de sous-objectifs, puis résout les sous-objectifs séparément par un planificateur local. Cependant, ce planificateur n'a pas abordé le problème dans lequel l'ordre des manipulations des objets décide de la solution. Plus précisément les auteurs proposent une méthode basée sur une grille qui représente l'environnement complet. Pour trouver un chemin vers un objectif, ils utilisent un planificateur global et un planificateur local. Le planificateur global se charge de planifier le chemin global pour atteindre un objectif défini à l'avance. Par la suite, le chemin est décomposé en une série de sous-objectifs. Le planificateur local est appelé pour évaluer la faisabilité de chaque sous-objectif. L'algorithme ne permet pas de résoudre une grande variété de problèmes et n'est pas optimal. Les hypothèses faites par l'algorithme sont simples (par exemple, déplacer un obstacle une seule fois).

Un second article traitant toujours de la NAMO hors ligne [Okada et al., 2004] présente un robot humanoïde avec deux sous-ensembles de planificateurs, un pour le déplacement en espace libre et un second pour la manipulation des obstacles. Le robot construit un graphe de tâches de manipulation des obstacles à exécuter dans un ordre donné. Chaque nœud du graphe représente une tâche, par la suite, chaque tâche est décomposée en sous-tâches, qui sont résolues à l'aide du planificateur de chemin. L'algorithme n'est pas efficace et ne prend pas en compte les obstacles mobiles qui bloquent indirectement le chemin vers l'objectif. Beaucoup d'autres articles traitent de la navigation hors ligne parmi les plus intéressants, on trouve par exemple : [Stilman and Kuffner, 2008] [Stilman and Kuffner, 2005] [Nieuwenhuisen et al., 2008].

Les travaux sur la NAMO en ligne ne sont pas nombreux[2], l'article [Wu et al., 2010] est l'un des premiers à aborder la NAMO dans un environnement inconnu. Il présente un algorithme qui se base sur de fortes suppositions, par exemple, le robot ne pousse les objets qu'une seule fois, que l'environnement est une grille plane, que les objets sont alignés dans la grille, avec la différence que la grille est inconnue au départ (c.f. section 2.1.2). L'algorithme se base sur les nouvelles informations recueillies durant le trajet du robot, il est capable d'identifier les nouvelles données qui n'affectent pas les calculs du chemin établi précédemment. Cependant les solutions locales résultantes de nouvelles informations du robot sur l'environnement immédiat ne permettent pas de résoudre tous les cas.

L'article [Levihn et al., 2013c] propose un planificateur qui fonctionne en environnement réel et utilisable dans la pratique, il prend en compte le faite que l'environnement n'est pas déterministe et que les espaces de configurations et d'actions ne sont pas discrets, selon les auteurs, ce sont ces deux paramètres qui empêchent les autres planificateurs d'être utilisés dans la pratique avec des robots réels. La méthode fournit une solution théorique de décision à la sélection d'action pour la NAMO appliquée à un robot à commande continue. L'algorithme proposé dans cet article combine des planificateurs basés sur des processus décisionnels Markovien, ainsi que la simulation à l'aide de la méthode Monte-Carlo.

Un autre article [Levihn, 2011] présente un algorithme pour la NAMO dans un environnement inconnu avec des obstacles non nécessairement alignés dans une grille, avec un robot équipé de capteurs pour collecter des informations sur les obstacles proches pour les pousser ou les tirer, mais une seule fois uniquement, comme dans le précédent article du même auteur [Wu et al., 2010]. Le robot planifie un chemin provisoire vers son objectif à l'aide de la recherche $A*$, puis le modifie et l'applique de manière incrémentale pour les segments de chemin bloqués par des objets mobiles. L'algorithme proposé s'avère être optimal sous certaines hypothèses, bien qu'il reste encore de nombreux cas qu'il ne peut résoudre.

L'article [Renault et al., 2019] donne un état de l'art sur la NAMO relativement récent et présente une comparaison (voir Tableau 2.1.1) des différents techniques utilisées, cette étude est fondée sur les quatre critères suivants :

1. *La représentation de l'environnement utilisé :* il est indiqué qu'une bonne représentation de l'environnement est importante, car cela permet de distinguer les obstacles les uns des autres et de raisonner sur leur placement futur [Ota, 2004] et les actions possibles sur les objets (affordance)[Kim et al., 2006].

   Le besoin le plus fondamental est la **mobilité**, en plus de la position et de la forme des obstacles. Dans la littérature, les obstacles individuels sont simplement supposés avoir un attribut booléen d'être mobiles ou non. Cet attribut était jusqu'à présent donné en entrée dans la plupart des techniques (1-7,11,12,15,16) présentées dans le Tableau 2.1.1, généralement la mobilité d'un obstacle est déterminée directement à partir des résultats de reconnaissance visuelle d'obstacles ou par tentative de manipulation (ces techniques sont utilisées en simulation uniquement) (8-10,17-19). Afin d'être plus réaliste, d'autres

[2]Le Tableau 2.1.1 donne les différentes approches existantes, la colonne *Mobilité* renseigne le type de la NAMO utilisée. Les approches *en ligne* sont celles dont la mobilité n'est pas donnée.

informations sémantiques sont utilisées dans des approches plus avancées, comme la cinématique et la physique des objets (masse, centre d'inertie, etc.)(1,3,4,16). Les résultats obtenus dans les implémentations réelles (pas en simulation)(17) restent mitigés, car ces caractéristiques sont difficiles à déterminer avec les capacités de détection actuelles des robots. D'autres types d'obstacles mobiles ou immobiles, comme les humains ou les objets en mouvement autonome n'ont jamais été considérés dans la littérature NAMO : une hypothèse standard est que le robot est le seul agent autonome dans l'environnement (voir colonne « Mobilité » du Tableau 2.1.1).

2. *La notion de coût et d'optimisation :* dans la NAMO comme dans la navigation classique (évitement d'obstacles) il est possible d'utiliser différentes façons d'estimer le coût, distance, temps, énergie, nombre d'obstacles déplacés, probabilité de succès, parfois combinés ou utilisés alternativement (voir colonne « coût » du Tableau 2.1.1). Le choix d'un coût qui ne prend en compte que la distance de déplacement peut être motivé par l'hypothèse que le poids des obstacles mobiles est négligeable au regard des capacités physiques du robot. Il est cependant évident que si la manipulation d'un obstacle entraîne un changement significatif des besoins en vitesse et en énergie par rapport à une seule tâche de navigation, le temps et l'énergie deviennent des choix bien plus appropriés.

3. *Les caractéristiques de manipulation :* Dans [Stilman et al., 2007] Stilman a formalisé trois classes principales de procédures de manipulation d'obstacles :

   (a) La saisie (Contact contraint)(1),

   (b) La poussée (Mouvement contraint)(1,7,16) et

   (c) Les primitives de manipulation (repose sur la simulation de la dynamique des objets, un glissement de translation ou de rotation peut se produire)(2-4,6,8-13).

D'après les résultats exposés dans le Tableau 2.1.1, Colonne « Manipulations », la saisie est la classe la plus utilisée, probablement parce qu'elle est la plus fiable. La poussée a également été envisagée, car les grands objets ne peuvent pas nécessairement être saisis. Les primitives de manipulation ont également été expérimentées, mais les implémentations dans le monde réel nécessitent des caméras externes pour fonctionner.

Il existe globalement trois stratégies de manipulation, la première considère qu'un seul obstacle peut être manipulé à la fois (pas d'effet de cascade sur les mobiliers proches). La seconde consiste à considérer qu'un ensemble limité de points de contact avec les objets, ainsi on facilite la recherche en arrière de la posture du robot pour la manipulation, c'est encore plus vrai pour certains obstacles qui ont des points de contact spécifiques (par exemple dessus de chaise, les objets avec des poignées, etc.). La troisième stratégie consiste à limiter la manipulation dans des directions spécifiques (par exemple pousser des objets uniquement en face.)

Cet article fait aussi référence à la sécurité et au confort des personnes. Il indique qu'un robot doit favoriser les classes de manipulation les plus fiables lorsque cela est possible pour réduire la complexité de la manipulation et ainsi réduire les chances d'échec qui pourrait mettre en danger les humains ou les objets.

4. *Les algorithmes de planification :* Dans certaines solutions (1,15,16), on propose des algorithmes qui ne distinguent pas clairement les différents aspects de la NAMO (manipulation des obstacles amovibles, actions possibles et calculs de chemin), on peut généralement distinguer un planificateur de décision de haut niveau et deux sous-programmes de planification de chemin. Ces sous-programmes peuvent être vaguement identifiés comme des planificateurs de chemin et de manipulation d'obstacles. Plus clairement la plupart des solutions embarquent deux planificateurs, une pour calculer le chemin global et un autre planificateur pour la gestion des obstacles.

   La plupart des solutions proposées se basent sur des algorithmes existants comme $Dijkstra$ (1), DFS (3,4,6,16), $A*$ (15), $Chaines\ de\ Markoves$ (11,12,17), $Monte-Carlo$ (18), $BHPN$[3] (13). Mais d'autres semblent avoir développé leur approches à partir de zéro (2,5,7-10,14,15,18,19,etc.) dans le but de réduire le temps de calcul. La plus courante méthode consiste à utiliser un planificateur de chemin heuristique (3,4,6,15,18,19) qui ignore les obstacles mobiles pour trouver des obstacles « bloquants ». Ensuite, on sélectionne le dernier objet bloquant ou la distance euclidienne minimale pour passer l'obstacle. D'autres techniques de sélection d'obstacles existent, on trouve par exemple les files de priorité utilisées par exemple dans [Meng et al., 2018].

Une synthèse des principaux critères de comparaison est donnée dans le Tableau 2.1.1 suivant :

[3]Belief Hierarchical Planning in the Now

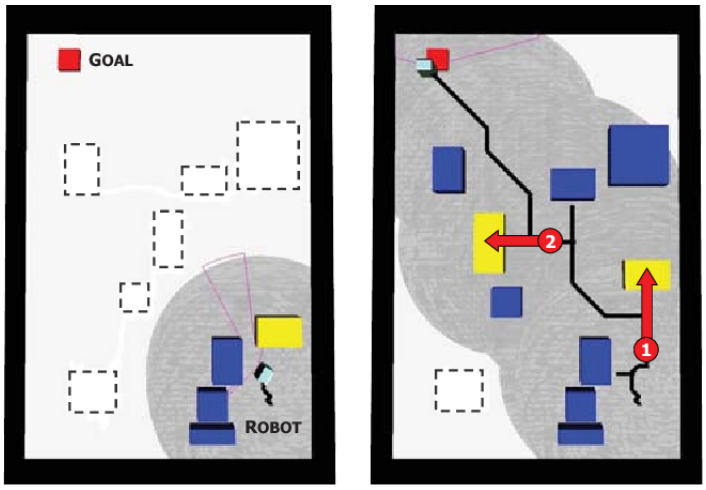
Auteur	Référence	Prior	Mobilité	Incertitude	Comp.	Opt.	Cout	Espace Conf.	Tâche P.	Transition P.	Transfert P.	Manipulation.	Env. réel
(1)Chen	[Chen and Hwang, 1990]	Comp.	Donné	Non	-	-	D	Disc.	Dij.+GD	N/A	N/A	Prim.	Non
(2)Okada	[Okada et al., 2004]	Comp.	Donné	Non	-	-	D‖E	Disc.	Pers.	NG	NG	Prise	Non
(3)Stilman	[Stilman and Kuffner, 2005]	Comp.	Donné	Non	RC	-	E+NMO	Disc.	DFS	A*	BTS	Prim.	Non
(4)Stilman	[Stilman et al., 2007]	Comp.	Donné	Pos.	RC	-	E+NMO	Disc.	DFS	A*	BTS	Prim.	Oui
(5)Nieuwenhuisen	[Nieuwenhuisen et al., 2008]	Comp.	Donné	Non	PC	-	D+PS	Cont.	Pers.	RRT	RRT	Prise	Non
(6)Stilman	[Stilman and Kuffner, 2008]	Comp.	Donné	Non	-	-	D+NMO	Disc.	DFS	A*	BTS	Prise	Non
(7)Van den Berg	[Van Den Berg et al., 2009]	Comp.	Donné	Non	PC	-	(D)	Disc.	Pers.	N/A	N/A	Prise	Non
(8)Kakiuchi	[Kakiuchi et al., 2010]	Non	Manip.	Pos. Mov.	-	-	(D+NMO)	Cont.	Pers.	RRT	N/A	Pous.	Oui
(9)Wu	[Wu et al., 2010]	Non	Manip.	Non	-	-	(D+T+E)	Disc.	Pers.	A*	DFS	Pous.	Non
(10)Levihn	[Levihn, 2011, Levihn et al., 2014]	Non	Manip.	Non	-	LO	(D+T+E)	Disc.	Pers.	D* Lite	DFS	Prise	Non
(11)Levihn	[Levihn et al., 2013b]	Comp.	Donné	Pos. Mov.	-	-	PS	Disc.	MDP+MCTS	N/A	N/A	Prim.	Non
(12)Levihn	[Levihn et al., 2013c]	Comp.	Donné	Pos. Mov. Kin.	-	-	T+E	Cont.	MDP+MCTS	PRM	RRT	Prim.	Non
(13)Levihn	[Levihn et al., 2013a]	Partiel	RecV.	Pos. Mov.	-	-	(D+T+E)	Cont.	BHPN	RRT	RRT	Prise	Oui
(14)Mueggler	[Mueggler et al., 2014]	Comp.	RecV.	Non	-	-	T	Disc.	Pers.	A*	Dji.	Prise	Oui
(15)Castaman	[Levihn et al., 2014, Castaman et al., 2016]	Comp.	Donné	Pos.	-	-	T	Disc.	KPIECE+A*	N/A	N/A	Prise‖*Pouss.*	Non
(16)Moghaddam	[Moghaddam and Masehian, 2016]	Comp.	Donné	Non	CO	-	E	Cont.	DFS	Dij.+GV	Dij.+GV	Prise	Non
(17)Scholz	[Scholz et al., 2016]	Comp.	RecV.	Pos. Mov. Kin.	-	-	T+E	Cont.	MDP+MCTS	PRM	RRT	Prim.	Oui
(18)Sun	[Sun et al., 2017]	Partiel	RecV.	Pos.	-	-	(D)	Cont.	Pers.	RRT	RRT	Prise‖*Pouss.*	Oui
(19)Meng	[Meng et al., 2018]	Partiel	RecV.	Pos.	-	-	D	Cont. MP	Pers.	RRT	RRT	Prise‖*Pouss.*	Oui

Tableau 2.1 : () = Non spécifié mais probable ; '+' = Combinaison ; '‖' = *Alternative*; *Manip.* = *Trouvé par manipulation*; *RecV.* = *Trouvé par reconnaissance visuelle*; *Pos.* = *Gère l'incertitude sur la position*; *Mov.* = *Idem pour lamobilité*; *Kin.* = *Idem sur la cinématique des objets*; '−' = *Selon les colonnes, non optimal ou incomplet*; *CO* = *Complet*; *RC* = *Résolution complète*; *PC* = *Probabiliste Complet*; *LO* = *Localement optimal*; *D* = *Distance*; *E* = *Énergie*; *T* = *Temps*; *NMO* = *Nombre d'obstacles déplacés*; *PS* = *probabilité de succès*; *Disque.* = *Discret*; *Cont.* = *Continu*; *MP* = *Multi − Planification*; *Dij.* = *Dijkstra*; *GD* = *Distance généralisée*; *GV* = *Graphique de visibilité*; *NG* = *Non donné*; *N/A* = *Non applicable*; *Prim.* = *Primitives de mouvement*

2.1.2 NAMO en environnement domiciliaire

Les travaux sur la NAMO dans un environnement inconnu ne sont pas très nombreux, néanmoins cet axe de recherche reste très important en raison de la nature du problème. Une avancée majeure dans ce domaine aura des conséquences majeures sur l'ensemble de la robotique mobile. La difficulté réside dans la nature de l'environnement, typiquement un environnement domiciliaire congestionné évolue constamment à cause des obstacles amovibles et de nature différente (dynamiques, statiques, interactifs, etc.). Les conséquences d'un environnement changeant, est de produire un nouvel environnement presque inconnu au robot après chaque passage.

M. Levihn et al. proposent dans [Wu et al., 2010] un algorithme pour NAMO permettant à un robot de naviguer dans un environnement inconnu (Figure 2.1). Il propose un scénario de navigation avec un robot non-holonome dans un espace de travail bidimensionnel contenant des objets mobiles et statiques. L'objectif est donné au robot avant le départ. La position de départ du robot est donnée dans les coordonnées du monde en 2D, mais aucune connaissance préalable de la position, de la taille, ou la mobilité des objets n'est fournie. Le robot obtient des informations sur la position et la taille des objets grâce à l'utilisation du télémètre laser, mais la mobilité ne peut être déterminée que par interaction avec l'objet. L'étude se limite aux interactions possibles avec les objets lors de poussées alignées sur un des axes x ou y dans un plan orthonormé. L'environnement est discrétisé dans une grille de taille NxN et les objets ainsi que le robot sont modélisés sous forme de rectangles.



(a) Initial State (b) Final Execution

FIGURE 2.1 : Résolution d'un problème NAMO dans un environnement à informations partielles. Les objets jaunes (légers) sont mobiles. Les bleus (lourds) ne le sont pas. (a) Les lignes pointillées représentent des objets inconnus. (b) L'évolution de la carte interne des robots.

Le robot proposé dispose d'une carte interne de l'environnement qui est mise à jour lors de la détection de nouveaux obstacles et de nouvelles informations sur des objets précédemment connus, telles que la taille et la mobilité. Chaque cellule inconnue sur la carte est supposée être un espace libre et chaque objet est considéré comme déplaçable avant l'exécution d'une action *push*. La solution proposée consiste à calculer des plans pour les actions possibles sur tous les

objets connus à chaque mise à jour du plan. Cette approche est décrite dans les algorithmes 1 et 2 et illustrée dans la Figure 2.2.

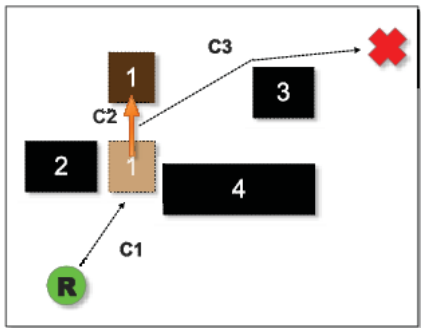


FIGURE 2.2 : Visualisation des étapes que le robot effectue pour pousser l'objet numéroté 1. R, représente la position initiale du robot. C1, C2 et C3 représentent les trajectoires du robot. 2,3 et 4 représentent les obstacles fixes. L'obstacle mobile 1, est représenté deux fois, en marron claire le robot pousse l'obstacle pour le déplacer, la flèche orange indique le sens de déplacement de l'obstacle.

L'algorithme est initialisé en calculant un plan avec une recherche $A*$ de R_{Init} à R_{Goal} en utilisant la distance euclidienne comme une heuristique admissible. Si aucune nouvelle observation n'est faite, ce plan reste inchangé jusqu'à ce que l'objectif soit atteint. Toutefois, si une nouvelle observation est effectuée, toutes les actions possibles sont évaluées pour tous les objets connus. Ceci est montré dans l'algorithme 2.

Algorithme 1 Base (R_{Init}, R_{Goal})

$R \leftarrow R_{Init}$
$O \leftarrow \emptyset$; { ensemble des objets}
$P_{opt} \leftarrow A*(R_{Init}, R_{Goal})$
while $R \neq R_{Goal}$ **do**
  $O_{new} \leftarrow$ GET-NEW-INFORMATION()
  if $O_{new} \neq \emptyset$ **then**
    $O = O \cup O_{new}$
    for $o \in O$ **do**
      for each possible push direction d on o **do**
        $p \leftarrow$ EVALUATE-ACTION(o, d)
        if $p.cost$ inf $P_{opt}.cost$ **then**
          $p_{opt} = p$
        end if
      end for
    end for
  end if
  $R \leftarrow$ Next step in p_{opt}
end while

L'article présente aussi des optimisations de cet algorithme en utilisant essentiellement trois techniques :

Algorithme 2 EVALUATE-ACTION(o, d)

$P_{o,d} \leftarrow \emptyset$
$c_1 = |A * (R, o.init)|$
$o.position = o.init$
while push on o in d possible **do**
  $o.position = o.position + one\_push\_in\_d$
  $c_2 = (o.position - o.init)$
  $c_3 = |A * (o.position, R_{Goal})|$
  $p = c_1 + c_2 + c_3$
  $p.cost = c_1 * moveCost + c2 * pushCost + c_3 * moveCost$
  $Po, d \leftarrow P_{o,d} \cup \{p\}$
end while
return $P \in P_{o,d}$ with min $p.cost$

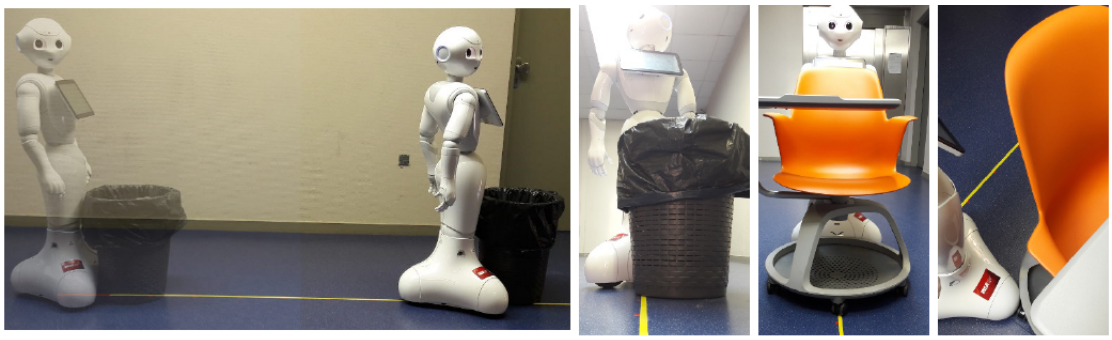
1. Initialement le calcul est relancé automatiquement à chaque détection de nouveaux objets, ou à la mise à jour d'informations sur les objets déjà existants aux mêmes positions. L'optimisation consiste à recalculer uniquement si le plan actuel devient invalide en raison de collisions avec des obstacles nouvellement détectés.

2. Pour éviter de faire trop de calcul, le nombre de fois où le robot peut pousser un objet est limité.

3. La dernière technique consiste à réduire les objets candidats, c'est-à-dire que les objets précédemment poussés ne sont plus considérés.

L'algorithme présenté ici permet de résoudre le problème NAMO dans un environnement inconnu. Malheureusement, cette solution implique uniquement des obstacles de forme géométriques, stricts (rectangles ou carrés) et les déplacements d'objets sont possibles uniquement sur deux axes (x ou y). L'application de cette solution n'est pas envisageable dans un environnement domiciliaire. L'un des problèmes majeurs de la mise en œuvre de cette solution dans un environnement domiciliaire, c'est qu'elle ne permet pas d'assurer la sécurité, car le test des obstacles (si les obstacles sont fixes ou amovibles) est effectué en essayant de les pousser. Dans un tel environnement certains obstacles (par exemple : objets fragiles, animaux domestiques, humains, etc.) risquent de s'abîmer ou de les blesser s'ils s'agit d'êtres vivants.

L'apparition de la NAMO à rapidement intéressé d'autres domaines de la robotique tel que la robotique sociale (Social Robotics), plus précisément la Navigation de Robot avec Conscience Sociale ou *Socially-Aware Navigation (SAN)* [Kruse et al., 2013] [Levihn et al., 2014] [Scholz et al., 2016]. La méthode présentée précédemment a été récemment améliorée dans [Renault et al., 2019] avec le but de réorganiser les différents objets qu'on trouve en milieu domiciliaire selon les règles sociales des humains. Les auteurs proposent un nouvel acronyme S-NAMO pour Social-NAMO. On peut voir par exemple dans l'expérimentation proposée par les auteurs dans la Figure 2.3, (a) et (b) le robot Pepper déplacer la poubelle dans la cuisine et (c) et (d) mettre la chaise à côté de la table, ce qui est socialement acceptable pour un humain.



(a) (b) (c) (d)

FIGURE 2.3 : Le robot Pepper décrit dans [Renault et al., 2019], pousse la poubelle pour la mettre dans la cuisine et la chaise à côté de la table.

La technique mise en œuvre ici s'inspire et améliore la méthode décrite précédemment [Wu et al., 2010], l'algorithme repose sur deux procédures : une procédure principale `make-and-execute-plan()`. Selon le contexte, cette dernière appelle une sous-procédure de planification de trajectoire combinée avec la procédure `make-plan-for-obstacle()`.

La procédure principale, `make-and-execute-plan(`w`,` q_{init}`,` q_{goal}`)`, construit et exécute le plan de navigation optimal p_{opt} à partir de la connaissance du monde w (carte métrique 2D avec des entités de formes polygones) et des configurations du robot $\{q_{init}, q_{objectif}\}$. Le plan p_{opt} est soit un chemin évitant tous les obstacles, soit un plan construit à partir de trois composants de chemin (voir Figure 2.4) : $c1$, $c2$, $c3$, sont respectivement des chemins de q_r à q_{manip}, de q_{manip} à q_{sim} où le robot arrête de déplacer l'obstacle o, et de q_{sim} à q_{but}. Il essaie dans un premier temps de trouver le meilleur plan à éviter. La connaissance du robot est mise à jour après chaque étape d'exécution, et si p_{opt} n'est plus valide (future collision avec d'autres obstacles par un robot ou un obstacle manipulé, échec de la manipulation, ou interruption de la mise à jour de la géométrie de l'obstacle manipulé), une nouvelle planification est déclenchée.

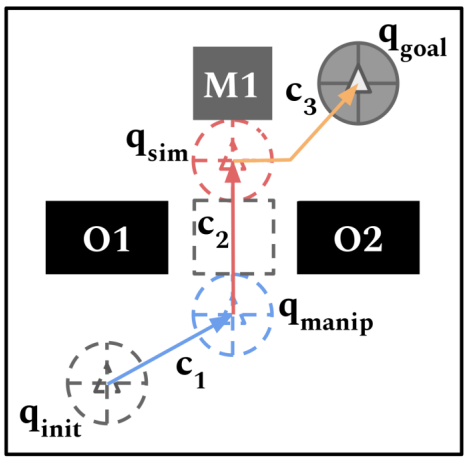


FIGURE 2.4 : Le robot (disque gris) exécute un plan p_{opt} en trois étapes pour déplacer M.

La sous-procédure, `make-plan-for-obstacle` (w, q_r, q_{goal}, o, p_{opt}), est appelée lors de l'itération sur les obstacles, et retourne le meilleur plan p impliquant au mieux la manipulation de l'obstacle o. Il itère sur les actions act qui peuvent être faites sur o, en supposant qu'il n'y a qu'une seule configuration de robot q_{manip} pour chaque paire o, act (milieu du côté o). Les composants du plan sont calculés séquentiellement, en commençant par $c1$. Si $c1$ est trouvé, les actions unitaires successives de longueur constante sont simulées (comptage des temps) dans une copie de w jusqu'à ce que le déplacement devient impossible (collision avec un autre obstacle). Pour éviter des calculs inutiles de $c2$ et $c3$, la simulation est arrêtée dès qu'un coût sous-estimé (C) et/ou le coût du plan actuellement évalué devient supérieur à celui de p_{opt}. C est la somme du coût de $c1$, une estimation $c2$ (produit du nombre par unité de longueur) et une estimation $c3$ (distance euclidienne minimale entre o et q_{but}). De plus, une évaluation complète n'est effectuée que si une nouvelle ouverture locale a été créée autour de o.

L'approche initiale décrite précédemment dans [Wu et al., 2010] suppose que tout obstacle est mobile à moins qu'une tentative de manipulation n'échoue, dans ce cas l'obstacle est mis sur liste noire (pour indiquer que cet obstacle n'est pas mobile). Cependant, dans S-NAMO (décrite ici), ce comportement n'est pas acceptable, car il pourrait conduire à des manipulations d'objets non autorisées (humain, objets fragiles, objet non-manipulable, etc.). Dans cette approche, le robot détient une liste des obstacles mobiles (amovibles), si les objets détectés ne sont pas dans cette liste, alors ils sont considérés comme étant non-mobiles. Mais le mobile repose souvent sur plusieurs capteurs, et leurs champs de vision respectifs ne sont pas nécessairement égaux. Un obstacle peut avoir été détecté géométriquement, mais pas encore identifié, conduisant à trois états possibles : (1) Inconnu (2) mobile et (3) non-mobile. Comme dans l'algorithme initial, ici, les auteurs supposent que le robot est équipé de deux types de capteurs, un géométrique (télémètre laser haute résolution) (voir le disque bleu dans la Figure 2.5) et une caméra RGB avec un système d'identification des objets (voir la partie en vert dans la Figure 2.5), ces deux capteurs sont supposés fonctionner parfaitement. Le champ de vision des deux capteurs est différent, le capteur laser recouvre une zone sur 360° alors que la caméra RGB a un champ de vison restreint à quelques dizaines de degrés. La procédure `make-plan-for-obstacle()` décrite dans l'Algorithme 3 a été adaptée pour fonctionner sous ces hypothèses. Lorsque l'obstacle o est identifié comme étant mobile, l'algorithme reste inchangé. Lorsque l'obstacle o est inconnu, l'algorithme vérifie d'abord si le calcul de c_1 peut fournir une certitude d'observation, sinon, il essaye de trouver un autre chemin qui garantit une meilleure observation. Pour ce faire, dans la fonction `compute-c0-c1()`, l'algorithme détermine la liste des configurations du robot qL qui permettrait l'observation : tout d'abord, il obtient toutes les configurations sans collisions dans la zone entre les polygones d'obstacles. Ensuite, il exécute l'algorithme $A*$ entre q_{manip} et chaque configuration dans qL. La même chose est faite avec la configuration actuelle du robot q_r à tous les éléments de qL. Enfin, il retourne la meilleure paire de chemins $\{c_0, c_1\}$ (voir Figure 2.5).

Cette technique est implémentée dans un simulateur basé sur ROS, les auteurs nous indiquent qu'il s'agit seulement d'une première étape en vue d'une implantation sur un vrai robot (**Pepper**), au moment de la rédaction de ce rapport aucune implémentation réelle n'a été proposée par les autres. L'implémentation proposée jusque ici est largement simplifiée, les obstacles mobiles sont supposés être des polygones convexes. Tous les calculs sont effectués sur le modèle vectoriel 2D, à l'exception de la planification de trajectoire, qui est implémentée comme un algorithme $A*$ de recherche dans une grille.

Algorithme 3 Extension de l'approche de Wu Levihn

▷ lorsque le plan est invalidé, faire un plan en évitant tous les obstacles, puis essaie de l'améliorer en répétant les obstacles et en appelant la fonction `make-and-execute-plan`

function MAKE-AND-EXECUTE-PLAN(w, q_{init}, q_{goal})
  `make-plan-for-obstacle`$(w, q_r, q_{goal}, o, p_{opt})$
end function
function MAKE-PLAN-FOR-OBSTACLE(w, q_{init}, q_{goal})
  $p_{best} \leftarrow \emptyset$
  for each `is-unknown`(o) **do**
    $q_{manip}, c_1 \leftarrow q - for(o, act), A * (w, q_r, q_{manip})$
    if $c_1 \neq \emptyset$ **then**
      $q_{look} \leftarrow get - last - look - q(w, o, c_1)$
      if $q_{look} \neq \emptyset$ **then**
        $c_0, c_1 \leftarrow compute - c0 - c1(w, o, q_r, q_{manip})$
        $c_0 \leftarrow \emptyset$
      end if
    end if
    if $is - movable(o) or (is - unknown(0) and c_0 \neq \emptyset and c_1 \neq \emptyset)$ **then**
      $w_{sim} \leftarrow copy(w)$
      $count, q_{sim} \leftarrow 1, sim - one - step(w_{sim}, act, o, q_r)$
      while $C_{est}(W_{sim}, c_1, count, act) \leq cost(p_{opt}) and is - step - success(q_r, q_{sim}, count, act)$ **do**
        if $check - new - opening(W, W_{sim}, o) and not - in - taboo(w, o))$ **then**
          $c_2 \leftarrow line(q_{manip}, q_{sim})$
          $c_3 \leftarrow A * (W, W_{sim}, o)$
          if $c_3 \neq \emptyset$ **then**
            $p \leftarrow plan(c_0, c_1, c_2, c_3, o, act)$
            if $cost(p < cost(p_{best}))$ **then**
              $p_{best} \leftarrow p$
            end if
            if $cost(p_{best} < cost(p_{opt}))$ **then**
              $p_{opt} \leftarrow p_{best}$
            end if
          end if
        end if
        $count, q_{sim} \leftarrow count + 1, sim - one - step(W_{sim}, act, o, q_{sim})$
      end while
    end if
  end for
  return p_{best}
end function
function COMPUTE-C0_c1(W, O, Q_r, q_{manip})
  $qL \leftarrow get - ql(w, o)$
  $paths - qL - q_{manip} \leftarrow multigoal - A * (w, q_{manip}, qL)$
  $paths - qL - q_{manip} \leftarrow multigoal - A * (w, q_r, qL)$
  return $shorest - c_0 - c1(paths - q_r - qL, paths - qL - q_{manip})$
end function

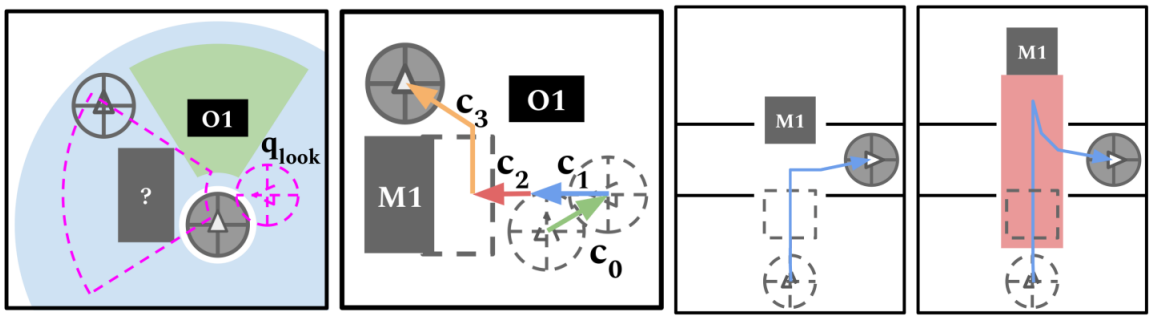


FIGURE 2.5 : (a) Le capteur laser (bleu) a détecté deux obstacles, le capteur RGB, n'a identifié que l'obstacle non-amovible $O1$. Le robot est trop proche de l'autre obstacle pour l'observer. Il est nécessaire de passer par une meilleure configuration d'observation intermédiaire : le meilleur chemin final avec $c0$ est indiqué en (b). (c) Représente deux pièces en vis-à-vis séparées par un couloir. Dans la NAMO (c) typique, le robot poussera $M1$ juste assez pour passer, bloquant l'autre porte. Dans la proposition S-NAMO, la zone critique (rouge) empêche le blocage, mais peut se retrouver avec un plan plus long.

2.1.3 Formulation du problème

Le problème de la NAMO, peut être abordé de différentes façons. Traditionnellement, on utilise deux approches, la première consiste à utiliser la planification de mouvement géométrique qui consiste à approcher le problème en utilisant une définition mathématique de l'environnement et du robot, les actions de déplacement du robot sont vues comme des applications de fonctions. La seconde méthode utilise une approche visuelle, de nos jours, les recherches en navigation robotique se penchent vers cette approche en raison du faible coût lié directement à la puissance de calcul abondante. Cette méthode permet généralement de produire une carte de l'environnement qui sert de base pour l'application de divers techniques tel que les graphes, division cellulaire, etc. Dans la suite de ce chapitre, nous allons formaliser la problématique de la NAMO en utilisant ces deux différentes techniques afin de mettre en évidence leurs avantages et inconvénients, et finalement nous allons mettre en évidence notre choix pour aborder ce problème.

Planification de mouvements en NAMO

La planification de mouvements géométrique [Latombe, 2012] suppose que la géométrie et la cinématique de l'environnement et du robot sont connues, suppose également qu'il n'y a pas d'incertitudes dans la détection et les effets des actions du robot. La représentation des objets et des liens du robot sont sous forme de polyèdres. On pose que les objets d'environnement sont classés en objets fixes, mobiles ou interactifs (objets que le robot peut déplacer avec un acte de communication (demander de laisser le passage)). Les actes de communication du robot avec les autres entités de l'environnement sont considérés comme des actions nulles. Formellement, l'environnement est modélisé comme un espace euclidien 2D ou 3D qui contient les éléments suivants :

- $O_f = \{F_1, \ldots, F_f\}$ - Un ensemble *d'obstacles fixes* que le robot doit éviter.

- $O_m = \{M_1, \ldots, M_m\}$ - Un ensemble *d'obstacles amovibles* que le robot peut déplacer en les poussant.
- $O_i = \{I_1, \ldots, I_i\}$ - Un ensemble *d'obstacles interactifs* avec lesquels le robot peut interagir pour demander de libérer le passage.
- R - Robot à n degrés de liberté représentés par un polyédrique unique (sans liens).

  Bien que les chemins ne puissent pas être explicitement paramétrés par le temps, nous utiliserons la variable t pour faire référence à un ordre chronologique des états et des opérations. À tout moment, l'état de l'environnement W^t définit la position et l'orientation du robot et de chaque objet. Nous représentons l'état de l'environnement comme suit :

  $$W^t = (t, r^t, q_1^t, \ldots, q_m^t)$$

  Avec une configuration initiale W^0 du robot r^0 et chaque obstacle mobile ou interactif q_i^o, l'objectif est de réaliser une configuration finale r^f pour le robot.

Afin d'atteindre cet objectif, le robot peut modifier sa propre configuration et éventuellement la configuration d'un obstacle à n'importe quel pas de temps t. Nous pouvons interpréter donc tout « changement » comme une action qui suit un chemin ou une trajectoire.

Nous pouvons distinguer deux opérateurs ou actions primitives : *Naviguer* et *Manipuler*. Chaque action est paramétrée par un chemin : $\tau\,(r_i, r_j)$ qui définit le mouvement du robot entre deux configurations : $\tau : [0, 1] \to r$ où $\tau\,[0] = r_i$ et $\tau\,[1] = r_j$.

L'opérateur *Navigation* fait référence au mouvement sans collision. Alors que le robot peut être en contact avec un objet, son mouvement ne doit être en contact avec aucun autre objet (collision ou frottement). On déplace simplement le robot comme spécifié par τ.

$$Navigation : (W^t, \tau(r^t, r^{t+1})) \to W^{t+1}$$

Lorsque le mouvement du robot affecte l'environnement en déplaçant un objet, O_m, nous appelons l'action *Pousser*. L'opérateur *Pousser* comprend deux chemins : un pour le robot et un pour l'objet O^m. L'objet n'étant pas autonome, le chemin de l'objet est paramétré par le chemin du robot et le contact initial $P_m \in P(O_m)$. L'ensemble $P(O_m)$ est constitué des transformations associées entre le robot et l'objet constituant le contact.

$$Pousser : ((W^t, O_m, P_m, \tau(r^t, r^{t+1})) \to W^{t+1}$$

Chaque P_i conduit à différents mouvements d'un objet, étant donné la même trajectoire final. On pose $\tau_{om} = PousserChemin(P_m, \tau)$ l'interprétation de l'objet lorsque le robot pousse l'objet O_m en fonction des contraintes imposées par le contact. L'opérateur *Pousser* permet depuis un état du monde, un contact et un chemin, d'arriver vers un nouvel état du monde W^{t+1} où le robot et l'objet O_m ont été déplacés. L'action est valide lorsque ni le robot ni l'objet en question ne se heurtent ni ne déplacent d'autres objets.

Lorsque le robot demande à une entité de l'environnement de laisser le passage, il affecte l'environnement avec un acte de langage, alors il affecte les obstacles O_i, nous appelons cette action *Interaction* noté $I_i \in I(O_i)$. L'opérateur *Interaction* affecte uniquement l'objet O^i. L'objet est autonome, le robot n'effectue pas de déplacement.

$$Interaction : ((W^t, O_i, I_i)) \rightarrow W^{t+1}$$

Les problèmes liés à la **NAMO** même dans un environnement à informations complètes posent un défi important. Wilfong [Wilfong, 1991] a d'abord prouvé que la planification du mouvement dans un environnement avec des obstacles mobiles est *NP-difficile*. Demaine [Demaine et al., 2000] a également prouvé que même la version simplifiée de ce problème, dans laquelle seuls les obstacles carrés sont pris en compte, est également *NP-difficile*.

Dans le contexte de la planification géométrique, comme nous venons de le voir, il est difficile de trouver une solution en un temps raisonnable. Généralement, d'autres méthodes sont appliquées, on peut citer par exemple les méthodes d'échantillonnage telles que les PRM (*Probabilistic RoadMap Planning*) et les arbres aléatoires à exploration rapide RRT (*Rapidly-exploring Random Tree planner*). Elles ont été appliquées aux problèmes liés aux espaces de recherche exponentiellement grands [Kavraki et al., 1994] [LaValle and Kuffner Jr, 2000]. Ces méthodes sont particulièrement efficaces dans les espaces étendus où il est facile d'échantillonner des points qui élargissent considérablement l'arbre de recherche [Hsu et al., 1999]. Les problèmes de NAMO dans un environnement domiciliaire imposent au robot d'évoluer dans un environnement partiellement inconnu en raison de l'état changeant de l'environnement, ce qui rend ces méthodes difficilement applicables.

2.1.4 Une approche empirique pour la NAMO

Le problème de la NAMO n'est pas très loin du problème classique de la planification de mouvement (*en anglais : motion planning*), connu aussi sous le nom du **problème du déménageur de piano** (voir Chapitre 4), on peut dire que la NAMO est une extension du domaine de la planification de mouvement, car en plus de la navigation avec évitement d'obstacles, on ajoute au robot la capacité de déplacer certains obstacles, si son objectif est difficile d'accès.

Le problème de la NAMO peut être décomposé en deux parties : (1) La navigation dans les espaces libres et (2) la gestion des obstacles. La première phase consiste à planifier un chemin pour rapprocher le robot de son objectif, si des obstacles se dressent sur le chemin du robot alors il faut faire appel au second planificateur pour gérer les obstacles. Il suffit alors d'enchaîner ces deux planificateurs jusqu'à atteindre l'objectif.

Pour obtenir des résultats pertinents, le planificateur de chemin dans les espaces libres ne doit pas ignorer les capacités du robot à gérer les obstacles, c'est-à-dire, même si un chemin avec évitement d'obstacle existe entre la position de départ et la position finale, il ne doit pas être choisi systématiquement, mais doit être décidé selon une fonction de coût. Par exemple, si le chemin avec évitement d'obstacle est plus long (on considère ici que le coût est calculé selon la distance parcourue) alors qu'il existe un chemin beaucoup plus court, mais qui nécessite un déplacement d'obstacles, alors ici la seconde solution n'est pas sans pertinence et elle doit être choisie. Mais il faut aussi prendre en compte le coût de la gestion des obstacles qui n'est

pas forcément exprimé en terme de distance, mais souvent en terme d'énergie ou de temps nécessaire pour écarter les obstacles.

Pour un algorithme de planification de chemin, il est **difficile** de juger du coût d'un chemin bloqué par des obstacles, car le coût de la manœuvre totale (de la position de départ à la position d'arrivée) est calculée avec une combinaison de deux coûts : (1) le déplacement du robot et (2) la gestion des obstacles. Pour cela, nous pensons qu'un planificateur qui combine ces deux approches prendra la forme d'un superviseur (architecture de contrôle) pour gérer le séquencement des deux planificateurs tout en renseignant l'un et l'autre sur le coût nécessaire pour l'accomplissement d'une tâche de l'un et de l'autre. Or, si on agit sur un obstacle, le coût de cette action ne permet pas d'améliorer à priori le coût d'un chemin, c'est pourquoi nous proposons une solution basée sur une simulation préalable pour prédire les coûts.

Un tel superviseur doit être capable par lui-même d'indiquer aux planificateurs quelle situation simuler, car il existe un nombre infini de situations. De plus, il faudra aussi distinguer les situations de simulation et les situations réelles. La conception d'un tel système nécessite forcément un logiciel de haut niveau pour gérer les planificateurs, il doit aussi être capable d'interagir avec les capteurs et les actionneurs du robot soit pour alimenter en informations les planificateurs ou bien exécuter les plans d'actions issus des planificateurs. La seconde partie de ce chapitre s'intéresse aux architectures de contrôle robotiques et cognitives capables de réaliser de telles tâches.

2.2 Les architectures de contrôle pour la robotique et les architectures cognitives

Un robot mobile est composé d'éléments mécaniques et électroniques, pour contrôler l'ensemble de ces éléments et avoir un comportement cohérent tout en gardant une certaine facilité lors de la conception et le développement, il est indispensable d'empiler des couches logicielles. Les couches les plus basses fournissent une interface pour le matériel et les couches les plus hautes fournissent une interface pour l'utilisateur, ces deux couches sont plus au moins communes pour tous les robots et tendent à se standardiser. Entre ces deux couches se trouve essentiellement la partie intelligente du robot. L'intelligence du robot dépend en grande partie du niveau d'abstraction de son en environnement, car un niveau d'abstraction élevé permet l'application d'algorithmes (raisonnements) génériques sans se soucier des détails. Pour avoir des niveaux d'abstractions différents, il faut empiler des couches logicielles, mais au détriment du temps de calcul, donc il faut trouver un compromis entre les niveaux d'abstractions à mettre en œuvre et le temps de réponse qui s'allonge.

C'est autour de l'organisation de ces trois fonctions (gestion de matériel - abstraction - raisonnement) que sont déclinées les principales architectures de contrôles génériques de la robotique mobile autonome. Mais dans la pratique elles sont différentes pour chaque type de robot, on peut dire qu'elle sont conçues sur-mesure. Mais elles doivent avoir la particularité d'utiliser des modules génériques [Arkin, 1998], d'une façon plus générale elles obéissent à des règles de conception et d'implémentations[4]. On peut distinguer trois grandes catégories d'architec-

[4]Dans la réalité ces exigences ne sont pas toujours respectées, mais les tendances actuelles semblent converger vers des implémentations génériques.

tures de contrôle pour la robotique :Lles contrôleurs hiérarchiques, les contrôleurs réactifs et les contrôleurs hybrides. Ces architectures ne diffèrent pas forcément par les méthodes élémentaires employées, mais plutôt par les agencements et relations entre les modules.

Afin qu'un robot puisse répondre correctement aux exigences d'autonomie, il doit intégrer des processus de perception, de planification, de navigation, de décisions, d'actions et d'interactions [Murphy, 2000]. Certains de ces processus exigent beaucoup de temps de calcul, alors que d'autres nécessitent une réactivité rapide, donc le robot doit effectuer des arbitrages afin de réguler ces processus. Ces arbitrages sont réglés par un ensemble logiciels appelés **architectures de contrôle** du robot. Ces processus sont généralement divisés en trois grandes catégories : (1) la **perception** qui est la fonction sensorielle par laquelle le robot acquiert la connaissance du monde et de son environnement, (2) la **planification** qui est la fonction décisionnelle par laquelle le robot décide de l'enchaînement de ses actions sous la forme d'un plan, elle regroupe les processus de planification, de localisation, des décisions, etc. Et finalement (3) l'**action** qui est la fonction produisant les commandes sur les actionneurs faisant agir le robot dans son environnement.

La section suivante propose dans un premier temps une vue globale sur les architectures pour la robotique existantes ainsi que leurs limites, par la suite, on va discuter des liens possibles avec les architectures cognitives qui permettent d'avoir un niveau décisionnel de haut niveau.

2.2.1 Les architectures de contrôle pour la robotique

La robotique est un domaine multidisciplinaire, il fait intervenir la mécanique pour les éléments de structure, de l'électronique pour les capteurs et les actionneurs, et finalement l'informatique pour contrôler ces deux derniers éléments. Il est essentiel d'apporter une attention particulière à la partie logicielle qui permettra au robot d'agir plus au moins intelligemment.

Donc, l'autonomie des robots dépend de leurs capacités à répondre de façon cohérente, sûre et efficace aux situations rencontrées, cette tâche revient à la partie logicielle qui contrôle le robot. Les architectures robotiques sont intervenues très tôt dans le développement de la robotique, elles permettent de préciser autant que possible les composants logiciels et matériels utilisés pour mettre en place un robot, et précise les modalités d'interaction des composants. Les premiers robots mobiles dérivés des recherches en intelligence artificielle prévalentes entre 1960 et 1990, utilisent un enchaînement cyclique de trois processus : Perception, Planification et Action, connues sous le nom **Architecture de contrôleurs hiérarchiques** qui vient de l'organisation de la planification en couches interconnectées. Le fonctionnement repose essentiellement sur la transmission des instructions du haut vers le bas. La partie haute de l'architecture contient les fonctionnalités comportementales et décisionnelles définissant la mission du robot et se basent sur une modélisation interne de l'environnement. La partie basse de l'architecture décrit les fonctions réactives du robot et les algorithmes de contrôle des capteurs et des actionneurs. Erann Gat dans son article *Three Layer Architecture* (*T*3) [Gat et al., 1998] décrit en détail leur fonctionnement. On peut citer comme exemple *4D/RCS*, un modèle théorique d'une architecture robotique hiérarchique pour véhicule autonome, décrite dans [Albus, 2002].

Ce type d'architecture a été plus au moins abandonné depuis le milieu des années 80, car les roboticiens ont remarqué que le processus de planification est un processus complexe et non

adapté à un environnement dynamique. Effectivement, le processus de décision se base sur la planification réalisée dans les niveaux précédents et les nouvelles informations de l'environnement ne sont prisent en compte que lors du prochain cycle. Pour palier à ce problème et répondre aux situations où le robot doit agir rapidement un autre type d'architectures robotiques a vu le jour sous le nom *d'architectures réactives*.

Les architectures réactives [Arkin, 1995] utilisent seulement deux niveaux : *perception* et *action*, dans ce type d'architectures aucune représentation du monde n'est utilisée, le robot se base uniquement sur les informations reçues à l'instant t_n pour agit directement à l'instant t_{n+1}. Ce type d'architecture est adapté aux environnements où le robot doit agir rapidement, par exemple dans le cas des environnements dynamiques. D'autres types d'architectures connus sous le nom *d'architectures basées sur le comportement* souvent classées dans la même catégorie que les *architectures réactives*. Ces deux types sont souvent confondus dans la littérature [Arkin, 1995], car généralement, un comportement est défini comme une activité primitive qui prend en entrée des données capteurs et retourne une action. Dans ce type d'architecture, il arrive que plusieurs comportements réagissent en même temps, dans ce cas, plusieurs politiques ont été adoptées afin d'assembler les comportements et d'assurer une coordination entre les perceptions et les actions. On trouve par exemple : la coordination basée sur la priorité [Brooks, 1986], coordination basée sur le vote [Rosenblatt, 1997], coordination basée sur la sélection des actions majoritaires [Maes, 1989], etc.

Il est évident que la perte d'informations inhérente aux architectures réactives diminue leur efficacité malgré leur réactivité importante, un juste milieu entre ces dernières et les architectures hiérarchiques a été défini à la fin des années 80, dans les *architectures hybrides*. Elles utilisent le paradigme *planification* et *Perception-Action*. La planification est réalisée en une seule étape, la perception et l'action sont effectuées simultanément. Les architectures hybrides sont dotées d'un système réactif pour le contrôle de bas niveau et un système délibératif de haut niveau pour la prise de décisions. L'une des premières architectures hybride à avoir vue le jour est **AuRA**(Autonomous Robot Architecture) [Arkin, 1987].

La classification proposée ici n'est pas la seule possible, d'autres auteurs [Ingrand, 2003] divisent les architectures robotiques en trois grandes catégories : (1) les architectures réactives, les architectures subsomption [Brooks, 1986] [Arkin, 1989] et les architectures purement délibératives [Albus et al., 1987].

Les classifications proposées ici ne prennent en compte que les aspects réaction / délibération, mais d'autres classifications sont possibles, selon par exemple qu'elles soient mono ou multi-robots, ou selon les environnements de développement, ou encore selon le type de la plateforme robot cible (mobile, fixe, UAV, etc.) etc.

L'architecture LAAS

L'architecture LAAS [Alami et al., 1998] [Ingrand, 2003] a été pensée pour offrir une solution générique de développement de robots autonomes, qui soit en mesure de s'intégrer aux outils et aux méthodes de développement (spécification, intégration, tests et validation, etc.) existants. Elle est dotée de trois niveaux (c.f. Figure 2.6), (1) le niveau décisionnel, (2) le niveau fonctionnel et (3) le niveau de contrôle des requêtes.

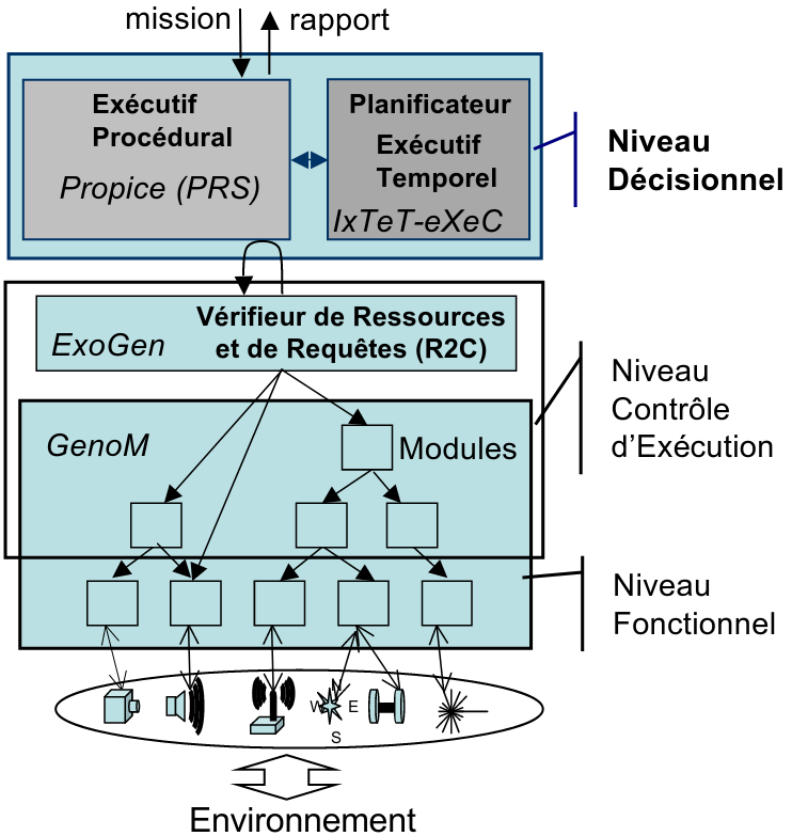


FIGURE 2.6 : Architecture LAAS.

- Le niveau décisionnel : Implémente les capacités délibératives du système, il comprend aussi les capacités de produire un plan de tâches et de superviser son exécution tout en étant à la fois réactif aux événements du niveau inférieur. Ce niveau peut être décomposé en deux ou plusieurs couches basées sur le même concept, mais en utilisant différentes représentations abstraites ou différents outils algorithmiques. Ce choix dépend principalement de l'application. Sur des applications particulières, il peut intégrer d'autres capacités de délibération plus complexes, qui sont appelées par le superviseur lorsque cela est nécessaire. Les propriétés temporelles du superviseur est définie comme la garantie du temps de réaction (c'est-à-dire le temps écoulé entre l'arrivée d'un événement et le moment où il le voit.).

- Le niveau fonctionnel : Il comprend toutes les capacités de base pour les actions et la perception du robot. Ces fonctions de traitement et boucles de contrôle (contrôle de mouvement d'évitement d'obstacles, traitement d'image, etc.) sont encapsulées dans des modules. Ces modules sont contrôlables par des requêtes, afin de rendre ce niveau aussi indépendant que possible du matériel et donc portable d'un robot à un autre. Chaque module est interfacé avec le capteur (ou les capteurs) qu'il gère via un niveau logique. Chaque module fournit un certain nombre de services et de traitements disponibles via les requêtes qui lui sont adressées. À la fin ou en cas de terminaison anormale, des rapports (avec statut) sont renvoyés à l'émetteur de la requête. Les modules sont entièrement contrôlés à partir du niveau décisionnel via le Vérificateur de Ressources et de Requêtes « Requests and Resources Checker (R2C) ».

Les modules contiennent également des « affiches », se sont des données produites par les modules, telles que la position et la vitesse actuelle (à partir du module de locomotion) ou la trajectoire actuelle (à partir du module de planification de mouvement) qui peuvent être vues par d'autres modules et les niveaux supérieurs. Les exigences temporelles des modules dépendent du type de traitements qu'ils effectuent. Les modules exécutant une boucle d'asservissement (qui doivent être exécutés à une fréquence et à un intervalle précis sans aucun retard) auront une exigence temporelle plus élevée qu'un planificateur de mouvement ou un algorithme de localisation.

- Le niveau de contrôle d'exécution : Permet de contrôler et coordonner l'exécution des fonctions réparties dans les modules en fonction des exigences de chaque tâche. Concrètement il permet de vérifier les requêtes envoyées par le niveau fonctionnel et vérifie aussi l'utilisation des ressources. Cette vérification est assurée par le R2C. Le niveau de contrôle d'exécution permet de synchroniser les modules fonctionnels sous-jacents, dans le sens où il vérifie toutes les requêtes qui leur sont envoyées, et tous les rapports qui en reviennent. Il agit comme un filtre qui autorise ou interdit le passage des requêtes. Ces modules sont développés à l'aide du framework *GenoM*[5] pour (Generator of Modules).

L'architecture LAAS est utilisée sur tous les robots du LAAS, ainsi que sur deux robots de la NASA (Gromit et K9), il existe actuellement plus d'une centaine de modules développés et maintenus, certains sont génériques et d'autres spécifiques pour certains robots. Nous n'avons trouvé aucun travail sur la NAMO utilisant cette architecture, probablement en cause de sa complexité et son vieillissement, car elle apparaît de moins en moins dans les récents travaux de recherche.

L'architecture IDEA

IDEA (Intelligent Distributed Execution Architecture) [Muscettola et al., 2002] [Ingrand, 2003] propose une nouvelle méthode pour structurer les architectures robotiques, elle se base sur le concept des systèmes multi-agent. Elle a été développée par la NASA pour la mise en place d'un planificateur/contrôleur d'exécution pour la sonde *Deep Space 1*[6].

Dans l'architecture IDEA, le système est décomposé en un ensemble d'agents (voir Figure 2.7) qui se basent tous sur un même modèle (voir Figure 2.8). Chaque agent est composé d'un planificateur(algorithme) et d'un exécutif(qui trouvent les prochaines actions à exécuter en fonction de l'état courant(automate)). Chaque agent peut posséder des planificateurs ou des exécutifs différents, à partir du moment où il respecte les spécifications définis par l'architecture IDEA. De cette façon, un agent qui gère un équipement de bas niveau peut réagir rapidement alors qu'un agent de planification peut prendre plus de temps pour délibérer. Dans dans ce système, on s'affranchit de la contrainte temporelle, c'est-à-dire que le système est à la fois réactif et délibératif. Malgré un concept très intéressant, ce système n'est pas très présent dans les robots mobiles.

[5] https://homepages.laas.fr/mallet/orocos/genom.pdf

[6] Une mission spatiale de la NASA, qui a pour but de tester de nouvelles technologies, le système IDEA est chargé de surveillance des pannes de la sonde. Lancée en 1998, la mission à durée jusqu'à décembre 2001.

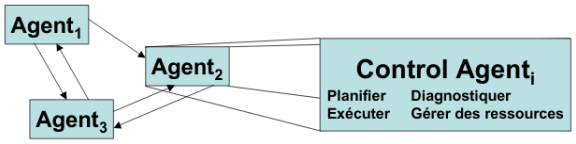


FIGURE 2.7 : Collection d'agents IDEA. Chaque agent est conçu avec une interface normalisée de tel sorte à ce qu'il puisse communiquer avec n'importe quel autre agent.

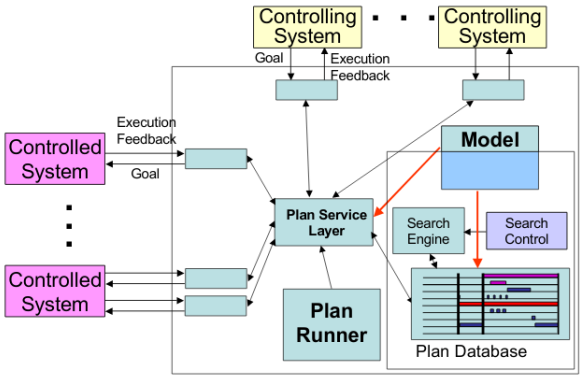


FIGURE 2.8 : Structure interne d'un agent IDEA. On trouve principalement trois sous-systèmes importants, le planificateur réactif, l'exécutif, et la base de plans.

Les architectures proposées ici, couvrent tout le spectre des compétences que le robot doit avoir pour mener à bien sa mission. Or, cela implique des compétences très variées (bas niveau / haut niveau), ce qui engendre un degré complexité élevé dans leur conception et utilisation. Nous pensons qu'il est essentiel de séparer la partie délibérative (raisonnement) de la partie de contrôle et fonctionnelle. Cela peut être fait grâce à la délégation de cette partie à des architectures plus spécialisées dans le domaine du raisonnement tel que les architectures cognitives.

2.2.2 Les Architectures cognitives

Dans le chapitre précédent, nous avons présenté quelques applications en cours de développement dans le domaine de la robotique mobile et plus particulièrement des robots qui se déplacent dans un milieu congestionné avec des obstacles amovibles (voir Figure 1.8). Dans ces cas, le robot est confronté au défi de se déplacer d'un point à un autre en écartant du passage les obstacles qui gênent son déplacement, pour assurer la sécurité et un déplacement efficace, le robot doit aussi gérer l'interaction avec les humains, les autres robots, les animaux domestiques, etc. De nombreuses solutions et travaux tentent de relever le défi en utilisant des algorithmes pour gérer uniquement les obstacles amovibles et/ou fixes, mais l'interaction avec d'autres types d'obstacles n'est pas prise en compte, comme on peut le voir dans les algorithmes présentés dans la première partie de ce chapitre. En effet, le développement d'un robot permettant à un robot de gérer la complexité des interactions avec des objets est un défi en soi, car de nombreuses fonctionnalités sont nécessaires : perception, gestion de la mémoire, raisonnement, affordance, gestion du dialogue et des aspects non-verbaux de l'interaction, etc. De plus, un état interne du robot est quasi essentiel pour gérer l'information issue de ces fonctionnalités. Les **architectures**[7] **cognitives** sont une solution apportée par les chercheurs pour concevoir un tel système [Djerroud and Cherif, 2016].

Malgré les différentes contributions dans le domaine des architectures cognitives et plus spécifiquement celles dédiés à la robotique [Pellier et al., 2018] [Lallée et al., 2012]

[7] "Le terme architecture implique une approche qui tente de modéliser les propriétés internes du système cognitif représenté et non seulement le comportement extérieur." Wikipedia https://fr.wikipedia.org/wiki/Architecture_cognitive

[Lemaignan et al., 2011], la plupart des architectures existantes, actuellement, sont génériques et peu d'entre elles peuvent vraiment gérer la complexité des interactions avec les objets de l'environnement dans le but d'atteindre des objectifs tels que la manipulation des objets à proprement parler.

Les domaines des architectures cognitives ne sont pas nouveaux, mais remontent à plusieurs décennies. L'état de l'art et l'évolution de ces différentes architectures cognitives est montré par exemple dans [Chong et al., 2007] [Thórisson and Helgasson, 2012]. On peut distinguer trois grandes familles d'architectures cognitives : (1) Les architectures cognitives bio-inspirées. (2) Les architectures cognitives dédiées à l'intelligence artificielle. (3) Les architectures cognitives inspirées des théories psychologiques, ce type d'architectures se base sur des théories psychologiques pour élaborer un système proche de celui d'un humain, le but est double, il permet d'élaborer des systèmes efficaces et proches de celui d'un humain et permet aussi en théorie d'étudier la psychologie humaine à travers un système informatique[8]. Dans ce qui suit nous allons détailler chaque grande famille, et détailler quelques implémentations existantes (les plus importantes et les plus connues).

(1) Les architectures cognitives bio-inspirées

Elles ont pour objectif de reproduire le comportement d'un système cognitif naturel (souvent le cerveau humain) afin de fournir une implémentation. Parmi les plus connues, on trouve par exemple : ACT-R, CLARION et ASMO.

L'architectures cognitive ACT-R

ACT-R(Adaptive Control of Thought—Rational) a été décrite dans [Anderson, 2013] [Anderson et al., 1997] est une d'architecture cognitive qui a pour objectif la modélisation du comportement humain, elle a connue un développement continu depuis le début des années 1970. ACT-R est organisée en un ensemble de modules, dont chacun traite un type d'information différent. Il s'agit notamment des modules sensoriels pour le traitement visuel, des modules de moteur d'action, un module intentionnelle des objectifs, et un module déclaratif pour la connaissance déclarative à long terme. Chaque module dispose d'une mémoire tampon associée qui contient une structure relationnelle déclarative (souvent appelé morceaux ou (*chunks*), mais différents de ceux de *SOAR(c.f. 2.2.2)*). Pris ensemble, ces tampons comprennent la mémoire à court terme de l'ACT-R.

ACT-R est composée principalement de trois mémoires : (1) La mémoire de travail (WM) : appelée aussi la mémoire à court terme, elle est en relation directe avec ce qui se passe dans l'environnement à un moment donné. Elle permet d'encoder la représentation de l'environnement. Cette mémoire est en relation avec les deux autres mémoires (DM et PM). (2) La mémoire déclarative (DM) : représente la mémoire à long terme, elle permet d'encoder la connaissance déclarative (*chunks*). Cette mémoire sert à stocker toutes les connaissances rencontrées ainsi qu'à sauvegarder les éléments de la mémoire de travail dans le but de les restituer plus tard. (3) La mémoire procédurale(PM) : elle permet de stocker ou de restaurer dans la WM les connaissances procédurales (*rules*). Les lois sont stockées sous forme d'un ensemble de productions.

[8]Dans notre travail nous nous intéressons pas à cet aspect.

ACT-R est organisée en un ensemble de modules, dont chacun traite un type d'information différent, et chacun ayant un équivalent chez l'humain (c.f. Figure 2.9 : *Visual Module, perception module, goal module, Declarative Memory module, manual module* et *Motor module*). La coordination est assurée par un système central. Chaque module est doté des trois mémoires précédemment citées (WM,DM et PM).

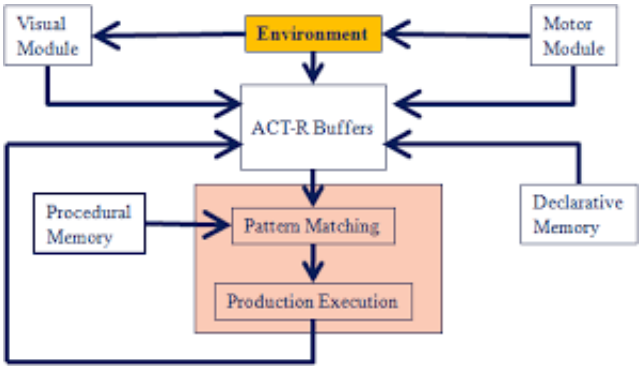


FIGURE 2.9 : Organisation des modules de ACT-R.

L'architectures cognitive CLARION

CLARION (Connectionist Learning with Adaptive Rule Induction On-line) a été décrite dans [Sun, 2007] [Sun, 2016]. Elle a pour objectif principale la recherche autour des mécanismes de la cognition humaine (prise de décision, apprentissage raisonnement, etc.) tout en développant également des agents artificiels.

CLARION est une architecture intégrative, composée d'un certain nombre de sous-systèmes distincts, avec une structure de représentation double dans chaque sous-système (représentations implicites et explicites) (c.f. Figure 2.10). Ses sous-systèmes comprennent le sous-système centré sur l'action (ACS), le sous-système non centré sur l'action (NACS), le sous-système motivationnel (MS) et le sous-système métacognitif (MCS). Le rôle du sous-système centré sur l'action est de contrôler les actions, qu'elles soient destinées à des mouvements physiques externes ou à des opérations mentales internes. Le rôle du sous-système sans action est de maintenir des connaissances générales, implicites ou explicites.

Le rôle du sous-système de motivation est de fournir des motivations sous-jacentes pour la perception, l'action et la cognition, en termes d'impulsion et de rétroaction (par exemple, en indiquant si les résultats sont satisfaisants ou non). Le rôle du sous-système métacognitif est de surveiller, de diriger et de modifier dynamiquement les opérations du sous-système centré sur l'action ainsi que les opérations de tous les autres sous-systèmes. Chacun de ces sous-systèmes en interaction se compose de deux niveaux de représentation (c-à-d. une structure de représentation double) : Généralement, dans chaque sous-système, le niveau supérieur code les connaissances explicites et le niveau inférieur code les connaissances implicites. La distinction des connaissances implicites et explicites a été largement argumentée auparavant sur la base de données psychologiques [Sun, 2001]. Les deux niveaux interagissent, par exemple, en coopérant dans des actions, en combinant les recommandations d'actions des deux niveaux respectivement, ainsi qu'en coopérer à l'apprentissage à travers un processus ascendant et descendant. Il s'agit essentiellement d'une théorie de l'esprit à double processus.

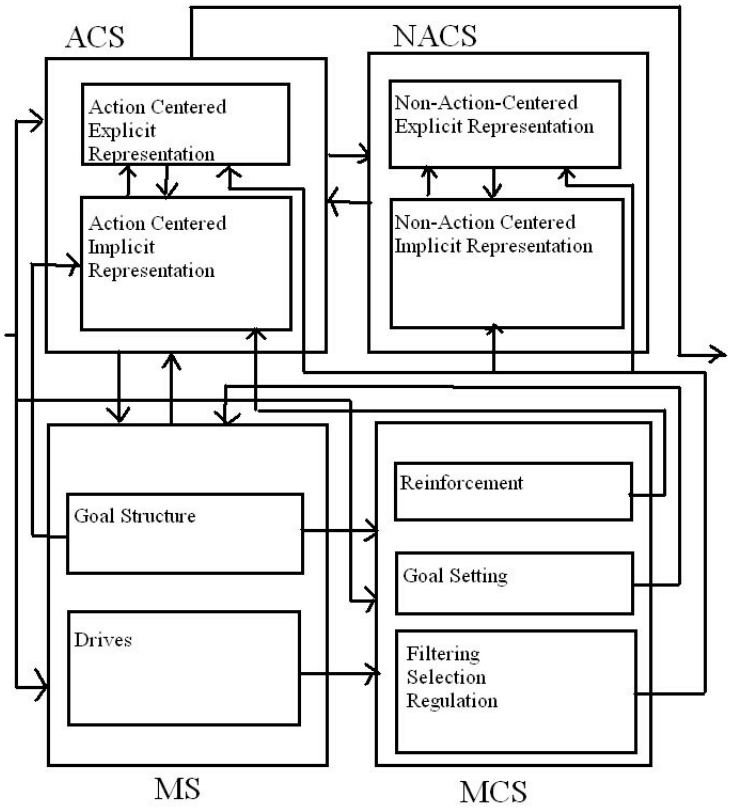


FIGURE 2.10 : Organisation de l'architecture CLARION.

(2) Les architectures cognitives dédiées à l'intelligence artificielle

Les architectures pour la résolution de problèmes d'Intelligence Artificielle (IA), souvent basées sur des IA symboliques qui mettent l'accent sur l'apprentissage et la résolution de problèmes. On trouve par exemple parmi les plus connues dans cette famille : SOAR et ICARUS.

L'architectures cognitive SOAR

SOAR (State, Operator And Result) [Laird, 2012] est une architecture purement "IA symbolique", qui met l'accent sur l'apprentissage et la résolution de problèmes. SOAR créée des représentations de la connaissance en utilisant des formes appropriées de la connaissance selon le contexte du problème (procédurale, déclarative, épisodique, et éventuellement iconique) (voir 2.11). Elle dispose d'une mémoire à court-terme (ou mémoire de travail), et une mémoire à long-terme qui se décompose en mémoire procédurale, mémoire sémantique et mémoire épisodique. L'apprentissage par renforcement est déclenché quand les connaissances ne permettent pas de prendre une décision.

Le but de SOAR est concevoir un système intelligent, mais on ne peut affirmer que c'est le cas, il semble qu'elle manque encore de certains aspects importants de l'intelligence notamment les émotions. SOAR a été étendue avec des émotions qui interviennent au niveau de l'apprentissage.

Cette architecture a été utilisée dans quelques implémentations robotique notamment pour la navigation des robots mobiles tel que décrit dans [Laird et al., 2012].

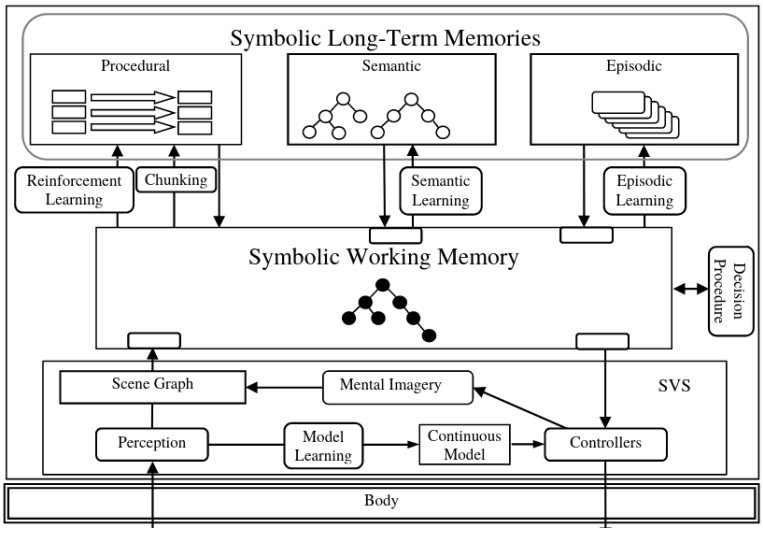


FIGURE 2.11 : Diagramme de l'architecture SOAR.

L'architectures cognitive ICARUS

ICARUS [Langley et al., 1991] est une architecture cognitive pour contrôler un agent intelligent dans un environnement physique complexe. Cette architecture a été conçue pour atteindre des objectifs grâce à la manipulation d'autres objets et à la navigation entre des positions. Par exemple, l'agent prend une tasse et la met sur la table (manipulation), ou l'agent passe d'une pièce à une autre en passant par le couloir.

ICARUS est une architecture cognitive plus récente qui stocke deux formes distinctes de concepts (voir Figure 2.12). Dans l'article [Choi and Langley, 2018] les auteurs décrivent les classes de situations environnementales en termes de concepts et de percepts, alors que les compétences spécifient comment atteindre les objectifs en les décomposant en sous-objectifs. Les concepts et les compétences impliquent des relations entre les objets, et les deux imposent une organisation hiérarchique de la mémoire à long terme.

(3) Les architectures cognitives inspirées des théories psychologiques

Elles se basent sur des théories philosophiques et psychologiques. Elles s'intéressent à l'ensemble des problèmes qui se posent dans l'activité fonctionnelle du cerveau liée à la sensation, au raisonnement, à l'action volontaire et l'intentionnalité. Par exemple dans cette famille d'architectures cognitives, on trouve : les architectures BDI (Belief,Desire, Intention) [Rao and Georgeff, 1991], dans cette théorie, les croyances et les désirs sont la cause de l'intention d'action. On peut citer aussi : PRS (Procedural Reasoning System) [Wooldridge, 2009] pour les agents rationnels, LIDA, CARAMEL, etc.

Les agents BDI

Regroupe une famille d'architectures basées sur des systèmes multi-agents, où chaque agent perçoit le monde avec ses propres moyens pour acquérir des connaissances, fait des supposition (croyances) et agit pour atteindre des buts (intention). L'architecture interne d'un agent

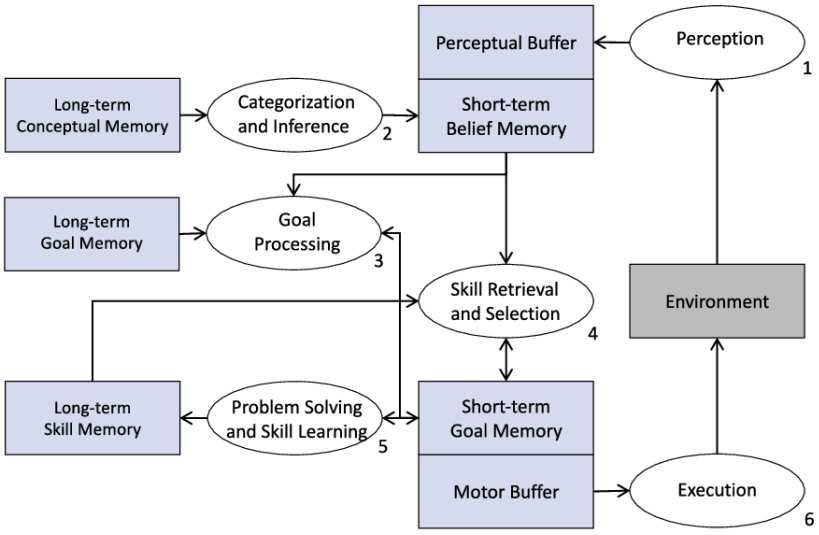


FIGURE 2.12 : Diagramme de l'architecture ICARUS, image extraite de [Choi and Langley, 2018]

est montrée dans la Figure 2.13, les cylindres montrent les données de l'agent, les rectangles montrent les traitements. L'architecture BDI repose sur une théorie du raisonnement pratique et essaie de d'expliquer comment un agent raison rationnellement pour prendre des décisions. Ce modèle montre le rôle important des intentions et le raisonnement, car elles limitent les choix possibles qu'un agent peut faire à un certain moment donné.

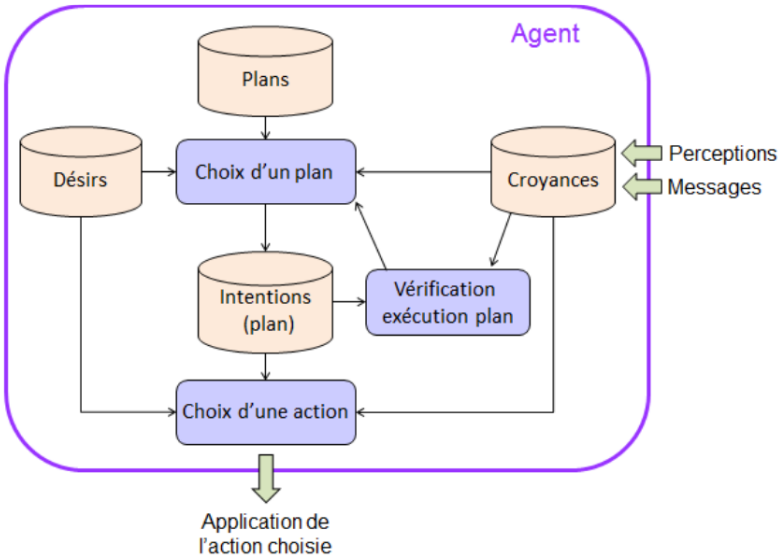


FIGURE 2.13 : Architecture interne d'un agent BDI. (Image extraite de [Taillandier et al., 2012])

1. **Croyances :** sont les informations que l'agent possède ou déduit sur l'environnement et sur les autres agents. Ces informations peuvent être incorrectes, incomplètes ou incertaines de plus, elles évoluent dans le temps. Les croyances sont différentes des connaissances, car ces dernières sont toujours vraies.

2. **Désirs :** Représentes sont but (ou les buts) que souhaite atteindre l'agent.

3. **Intention :** Correspond au plan choisi par l'agent pour atteindre un de ses buts.

Nous allons illustrer l'architecture BDI à travers un exemple[9] :
≪ L'agent Pierre a la croyance que, si quelqu'un passe son temps à étudier, cette personne peut faire une thèse de doctorat. En plus, Pierre a le désir de voyager beaucoup, de faire une thèse de doctorat et d'obtenir un poste d'assistant à l'université. Le désir de voyager beaucoup n'est pas consistant avec les deux autres et Pierre, après réflexion, décide de choisir, parmi ces désirs inconsistants, les deux derniers. Comme il se rend compte qu'il ne peut pas réaliser ses deux désirs à la fois, il décide de faire d'abord une thèse de doctorat. En ce moment, Pierre a l'intention de faire une thèse et, normalement, il va utiliser tous ses moyens pour y parvenir. Il serait irrationnel de la part de Pierre, une fois sa décision prise, d'utiliser son temps et son énergie, notamment ses moyens, pour voyager autour du monde. En fixant ces intentions, Pierre a moins de choix à considérer, car il a renoncé à faire le tour des agences de voyage pour trouver l'offre de voyage qui le satisferait au mieux. ≫

L'architectures cognitive LIDA

LIDA (Learning Intelligent Distribution Agent) [Ramamurthy et al., 2006] [Friedlander and Franklin, 2008], est une architecture cognitive basée sur la théorie psychologique Globar Workspace de Bernard Baars [Baars, 2005]. Cette théorie propose un tableau noir sans contrôle, un ensemble de processus sont soit en compétition (en parallèles) soit en collaboration (en série) afin d'écrire sur le tableau noir. Le cycle cognitif implémenté dans LIDA est divisé en trois phases, compréhension, attention et sélection de l'action et d'apprentissage. Ces phases se répètent indéfiniment.

Le modèle que propose Baars, est un modèle computationnel, c'est à dire qu'il est possible de faire une implémentation informatique et qu'il ne présente pas de limites algorithmiques. Le projet LIDA pour (Learning Intelligent Distribution Agent) [Baars, 1993] [Friedlander and Franklin, 2008] propose une implémentation de l'espace de travail global.

Le cycle cognitif que propose LIDA est divisé en trois phases, compréhension, attention (conscience) et sélection de l'action et d'apprentissage. Ces phases se répètent indéfiniment. La figure 2.14 montre les différents processus qui composent cette architecture.

Le système LIDA a été déployé pour la planification et l'affectation des soldats (US Navy) à leurs nouveaux postes, l'interaction avec les sujets se fait par e-mail. Le système reçoit les demandes et essai de répondre au mieux aux préférences, parfois, il propose même des plans pour palier à des situations inédites. D'après nos recherches, il n'existe pas d'applications en robotique connues pour cette architecture.

L'architecture CARAMEL

[9]Cet exemple est extrait du cours sur les SMA des autres suivants : Adina Florea, Daniel Kayser et Stefan Pentiuc, il est disponible sur le lien suivant : http://turing.cs.pub.ro/auf2/html/chapters/chapter2/chapter_2_2_2.html :

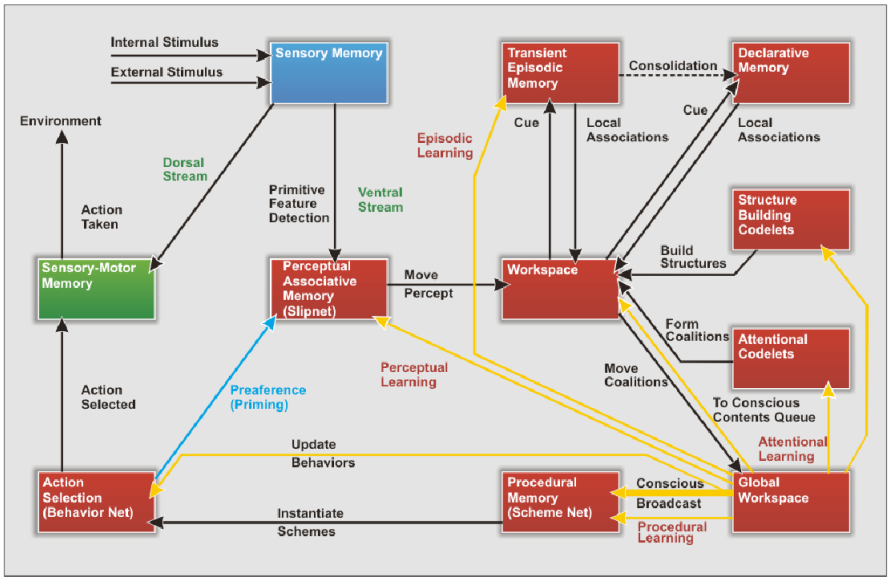


FIGURE 2.14 : Digramme de l'architecture LIDA.

Une architecture cognitive proposée par Gérard SABAH, il s'appuie sur le modèle du système réflexif classique (système capable d'appliquer à lui même ses propres capacités d'action) [Sabah, 1990] de l'intelligence artificielle auquel il adjoint deux extensions. Dans ce système l'agent se fait une représentation symbolique de celui avec lequel il va interagir. Dans ce modèle, chaque agent est contrôlé par un seul méta-système. D'après Sabah ce système est une représentation partielle de la conscience. Pour lui, ce modèle peut être étendu afin qu'un méta-système puisse contrôler plusieurs agents. Ainsi l'auteur propose d'étendre ce système par deux ramifications, (1) le fait qu'un méta-système puisse contrôler plusieurs agents et (2) le fait de considérer ces méta-systèmes comme des agents usuels et leurs appliquer récursivement des méta-représentations. Cette dernière extension fait que chaque agent est contrôlé par un méta-système qui est lui-même un agent sou contrôle d'un autre. Ce qui nous conduit à un système réflexif.

SABAH a implémenté en langage LISP sont modèle baptisé CARAMEL [Sabah and Briffault, 1993] (Conscience, Automatismes, Réflexivité et Apprentissage pour un Modèle de l'Esprit et du Langage) dans un premier temps, l'architecture a pour élément central la compréhension du langage naturelle, puis il apporte plusieurs idées à son système (CARAMEL 2) dont une conscience. Les carnets d'esquisse (tableau noir) et des méta-agents de contrôle. La conscience est un processus contrôlé qui établit le lien entre le traitement automatisé "inconscient" de l'information et la réflexion de plus haut niveau. Un carnet d'esquisse est un tableau noir modifié qui permet un retour d'information d'un niveau supérieur vers les processus plus simples. Par exemple, lorsque le niveau le plus bas interprète une série de sons comme étant les lettres 'a', 'v', 'h', 'r ' et 't', la couche supérieure va détecter que ceci n'est pas un mot valide et demander à la couche inférieure de lui faire une nouvelle proposition. Finalement, CARAMEL est un système multi-agents et multi-experts qui présente des méta-agents qui contrôlent un ou plusieurs autres agents, offrant ainsi de l'introspection.

2.2.3 Lien entre les architectures : robotiques - cognitives

Les roboticiens inventent un nouveau domaine dit de « Robotique Cognitive » [Réguigne-Khamassi and Doncieux, 2016] pour qualifier les recherches concernant des tâches que doit effectuer le robot, semblables a celles existantes chez l'humain, et qui semble nécessiter chez "l'humain" l'appel aux fonctions cognitives, par exemple la motricité, l'apprentissage, l'interaction sociale, la cognition spatiale, la navigation, etc.

D'un autre côté, il existe les architectures cognitives qui ont pour but de comprendre comment un système biologique implémente une fonctionnalité cognitive (fonction particulière chez l'homme ou l'animal) à travers une implémentation informatique. Le but étant de simuler informatiquement le comportement d'un système cognitif puis de mettre en confrontation les résultats des simulation et les résultats cliniques. Par une approche expérimentale et incrémentale, alors on confirme (ou non) l'hypothèse cognitive de départ, qui a servi à l'implémentation du système informatique.

Dans [Réguigne-Khamassi and Doncieux, 2016] Khamassi et al. ont montré que la tendance actuelle est vers une hybridation des approches. Il nous paraît évident que les architectures robotiques et les architectures cognitives sont facilement agençables, car toutes les deux possèdent une implémentation informatique. Les architectures cognitives sont au moins aussi anciennes que les architectures robotiques et ont connu un développement parallèle. Elles abordent les questions sur les capacités qui forment la cognition d'un être vivant, et par extension d'un système artificiel. Dans [Langley et al., 2009] Langley et al. cite par exemple, la capacité de reconnaître et de catégoriser, de prendre des décisions, de prévoir à long terme ou d'agir dans le monde. Plus récemment dans [Réguigne-Khamassi and Doncieux, 2016] Khamassi et al. identifie les points communs entre les problématiques de la robotique et celles considérées dans les sciences du vivant. Les architectures robotiques, initialement voulues pour être fonctionnelles, tendent naturellement vers les architectures cognitives comme le montrent les travaux qui adressent la question de l'architecture cognitive des robots [Kurup and Lebiere, 2012, Vernon et al., 2007a, Vernon et al., 2007b].

L'idée d'utiliser les architectures cognitives en robotique a été explorée dans de nombreux articles, on peut citer par exemple dans [Laird, 2009] l'auteur John E. Laird qui fait parti de l'équipe qui ont développé l'architecture cognitive SOAR, qui indique que l'objectif à long terme été de développer des systèmes robotiques autonomes qui ont les capacités cognitives des humains, et donc dans cet article, il explore les possibilités d'utiliser un tel système sans pilote. Peu après le même auteur propose dans [Laird et al., 2012] d'appliquer l'architecture SOAR à la robotique mobile.

D'autres applications de même type existent en utilisant d'autres architectures cognitives que SOAR, on peut par exemple citer un autre modèle basé sur l'architecture ACT-R [Trafton et al., 2013]. Cet article propose un système dans lequel un robot peut être un coéquipier d'un humain pour effectuer des tâches ensemble, dans ce cas le robot doit avoir une idée sur les croyances, les intentions et objectifs de son coéquipier (humain). Il propose dans un scénario dons lequel un humain et un robot patrouille dans un couloir bordé de bureaux, l'humain et le robot avance côte à côte, et chacun des deux, vérifie les bureaux de son côté seulement, car le robot a déduit que son coéquipier surveille un seul côté.

Malgré les applications montrées ici, aucune n'est assez efficace et sûre pour être réellement appliquée dans un environnement réel. Notre analyse nous a conduit à conclure que les architectures cognitives utilisées sont très génériques, et elles ont connu pendant très longtemps un développement qui ne tient pas compte des besoins réels en robotique.

Nous pensons qu'une architecture cognitive qui doit être appliquée à la navigation en robotique par exemple, doit se construire autour des fonctions cognitives qui sont dédiées à cette fonction. Or, les architectures comme SOAR et ACT-R, sont trop génériques. Plusieurs chercheurs [Berthoz and Petit, 2014] [Laumond, 2014] convergent vers cette conclusion et se tournent vers la théorie de l'esprit (Theory of Mind - ToM) [Davies and Stone, 1995] pour trouver des sources d'inspiration dans les sciences cognitives pour les appliquer à la robotique [Warnier, 2012] [Jones et al., 2014]. Notre travail s'inscrit dans cette lignée. Pour notre part, nous avons été fortement inspirés par les travaux du physiologiste Alain Berthoz. Il décrit le cerveau comme un prédicteur et un simulateur d'action. Le cerveau a pour fonction d'anticiper les événements futurs de l'environnement et simuler le mouvement adéquat pour répondre à un besoin. L'auteur appelle ce principe la **Simplexité**. Pour cela, nous proposons dans la section suivante, d'étudier cette théorie développée par le physiologiste Alain Berthoz, qui décrit le cerveau comme un prédicteur et un simulateur d'actions. Le cerveau a pour fonction d'anticiper les événements futurs de l'environnement et simuler le mouvement adéquat. Nous pensons que ce mode de fonctionnement est adapté pour la navigation.

2.3 Théorie de la simplexité

Le physiologiste Alain Berthoz, décrit le cerveau comme prédicteur et un simulateur d'actions [Berthoz, 2009]. Le cerveau a pour fonction d'anticiper les événements futurs de l'environnement et simuler le mouvement adéquat pour répondre à un besoin. La simulation est apparentée à une imagination du mouvement. C'est-à-dire, simuler mentalement le mouvement du corps dans un espace computationnel dans le cerveau. Par la suite, appliquer les mouvements simulés dans l'environnement réel et vérifier si les résultats correspondent a ceux de la simulation, s'ils sont différents alors le cerveau ajuste le mouvement.

Le physiologiste, indique que ce mécanisme est indispensable pour aller vite, par exemple dans les situations de danger pour échapper à un prédateur, capturer une proie ou tous simplement attraper un objet en mouvement, le cerveau n'a pas le temps de prendre toutes les informations sensorielles, de les traiter pour produire une action, il est important que le cerveau simule en interne les possibilités des actions avant de la produire ou choisir une action, car dans de nombreux cas, on n'a pas la possibilité de tester plusieurs actions. Dans l'exemple à attraper un objet en mouvement dans l'espace qui suit une trajectoire, le sujet ne peux pas lancer sa main pour attraper l'objet là où il est, il faut toujours aller là où l'objet sera, au moment de l'attraper. Donc, la fonction de simuler et prédire, est une fonction fondamentale, elle se traduit par le fait que, en même temps que nous planifions une action - d'attraper un objet - en même temps que le cerveau planifie le mouvement, il sélectionne les informations sensorielles pertinentes ou importantes pour le mouvement.

Dans un second exemple nous allons essayer de mettre en évidence ces fonctions et explorer les phases par lesquelles le cerveau transite durant une action rapide. Prenons l'exemple d'un

gardien de but qui saute pour attraper une balle, dans la phase ascensionnelle, le cerveau utilise *la proprioception* (de la position des différentes parties du corps) et *la vison* afin de mesurer le mouvement et doser l'énergie nécessaire pour produire le mouvement, ensuite s'enchaîne la phase de la prise de balle qui est une phase extraordinairement rapide pendant laquelle la vision ne sert à rien. Puis s'enchaîne la phase de chute où le cerveau utilise les capteurs vestibulaires pour mesurer l'inertie du corps en chute, puis au moment de toucher le sol, il va falloir prédire les informations pour préparer la chute. Autrement dit, le cerveau à chaque phase du mouvement et en fonction du contexte, le cerveau va présélectionner certains capteurs sensoriels qui sont importants. Mais le cerveau ne se limite pas à sélectionner les capteurs importants : il prédit l'état dans lequel ils devront être si le mouvement est accompli comme il doit l'être.

À chaque instant du mouvement, le cerveau aura prévu l'état dans lequel certains de ces capteurs devraient être (les capteurs sélectionnés). Sa fonction est donc non seulement d'aller collecter des informations sensorielles sur l'environnement (ce qui serait trop compliqué et long à traiter) mais simplement de comparer ces informations. Le cerveau a pour rôle de comparer les informations données par tous les capteurs avec les résultats prédits. Il simulera mentalement le mouvement qu'il a l'intention de faire ; il va prédire l'état de certains capteurs et il va comparer l'état de ces capteurs avec ce qu'il a prédit.

L'auteur indique que cette technique de simulation est une solution trouvée par le cerveau afin de simplifier la complexité de l'environnement, pour que le cerveau puisse préparer l'acte et en projeter les conséquences. Ce principe est nommé la « simplexité[10] »

Dans un autre volume [Berthoz, 2013] l'auteur traite d'un autre phénomène qui appelle la « Vicariance » c'est la capacité potentielle de sélectionner une stratégie parmi un ensemble offert ou disponible sous condition (défaillance, temporalité, singularité, posture réflexive, etc), en robotique cette capacité peut se traduire par exemple : les actions possibles (pousser par exemple) sur une boîte de conserve sont potentiellement les mêmes sur une boite d'allumette, etc.

L'idée d'appliquer la simplixité au mouvement du robot est rapidement arrivée, Jean-Paul Laumond discute dans [Laumond, 2014] des possibilités d'application de cette théorie en robotique. Encore dans un autre volume [Berthoz and Debru, 2015], un chapitre écrit par un roboticien est consacré à la prédiction en robotique [Ghallab, 2015], il discute les avantages évidents de la prédiction tel que l'anticipation est nécessaire à l'autonomie, l'anticipation permet la réalisation de tâches composées, etc. Malheureusement à ce jour il n'existe que des tentatives théoriques et timides, concernant l'implémentions de cette théorie en robotique mobile. Dans le cadre de notre travail, nous proposons dans le chapitre suivant une architecture basée sur les principes de la simplexité et de la vicariance pour les appliquer à la NAMO.

[10]On trouve différentes définitions dans la littérature de l'auteur : C'est la capacité de simplifier sans perte d'informations. C'est la capacité de pouvoir prédire par simulation les actions possibles.

3

Modélisation de l'architecture VICA pour la NAMO

Dans ce chapitre, nous présentons la contribution principale de cette thèse, une architecture robotique hiérarchique permettant à un système robotique autonome pour la navigation parmi les obstacles amovibles. Dans [Levitt, 1990] Revitte défini un système robotique autonome comme étant un système qui répond à ces trois questions : (a) Où suis-je et quelle est la nature des obstacles autour de moi ? (b) Comment rejoindre mon but à partir de la position initiale ? (c) Comment exécuter le mouvement ? Bien évidemment, la complexité des réponses à ces questions dépend directement de la complexité de l'environnement. Dans un environnement structuré et contrôlé, il est relativement facile de répondre à ces questions[1], cependant dans un environnement complexe et non structuré les réponses sont beaucoup moins évidentes.

Les processus mis en œuvre dans les techniques de navigation conventionnelles, considèrent généralement les obstacles comme infranchissables, ces techniques proposent habituellement comme solution **l'évitement des obstacles**. Les solutions apportées par ces méthodes peuvent dans certains cas montrer des faiblesses dans un environnement congestionné. Considérons par exemple le cas illustré dans la Figure 3.1, dans cette configuration, les méthodes de navigation conventionnelles vont proposer comme solution de contourner les obstacles comme illustré dans la Figure 3.1. Dans d'autres situations (e.g. Figure 3.2) où l'objectif est parfois obstrué (impossible d'accès) rendent ces méthodes de navigations inefficaces voire même impossibles à utiliser dans de tels environnements congestionnés.

Les techniques de navigation avec évitement d'obstacles ont déjà fait leurs preuves dans des environnements peu congestionnés, on trouve même des applications en robotique industrielle et de services (c.f. Chapitre 1). Dans le cas des environnements congestionnés, la NAMO offre

[1]Par exemple un environnement industriel offre un cadre très structuré, de faible variabilité dont tous les événements sont planifiés, tandis qu'un environnement humain présentent beaucoup moins d'éléments de structure et il est très variable (mon bureau n'est pas agencé comme votre bureau, etc.) un tel environnement est clairement dynamique et donc beaucoup moins prévisible.

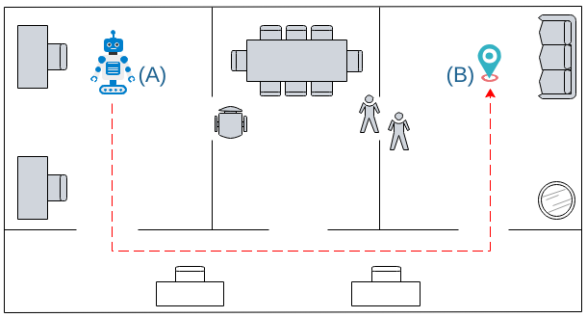


FIGURE 3.1 : Déplacement du robot du point A au point B avec évitement d'obstacles. On remarque qu'il existe un chemin plus court, mais obstrué par des obstacles.

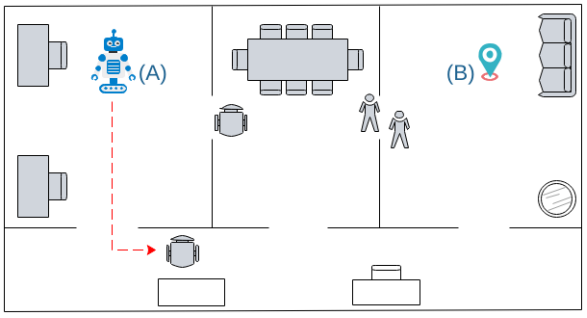


FIGURE 3.2 : Déplacement du robot impossible du point A au point B, en raison des obstacles qui bloquent les passages possibles.

une solution prometteuse, mais à notre connaissance, il n'existe pas encore d'applications en dehors des expérimentations faites en laboratoires. Contrairement aux méthodes de navigation classiques, la NAMO permet de doter les robots avec des capacités à modifier l'état de l'environnement en déplaçant les obstacles (tel que c'est illustré dans la Figure 3.3) dans le but d'optimiser les trajets de navigation, voire même de trouver des solutions pour des configurations qui sont impossibles à résoudre avec des méthodes de navigation traditionnelles.

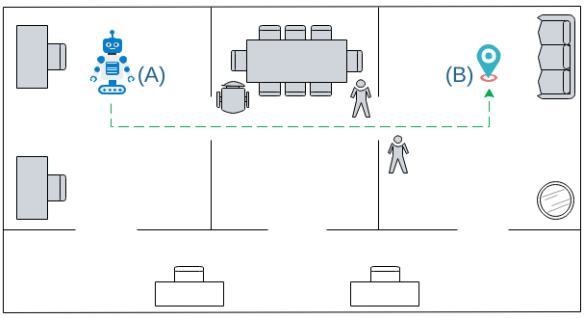


FIGURE 3.3 : Un robot équipé d'une méthode de navigation NAMO est capable de bouger la chaise, et demander aux personnes de lui céder le passage afin d'optimiser son parcours.

Il est essentiel pour la robotique en milieu domiciliaire de proposer des solutions de navigation capables de gérer des obstacles, sans cela les robots deviennent inefficaces. Dans ce contexte, un grand nombre d'approches ont déjà été expérimentées (c.f Chapitre 2). Cependant, aucune d'entre elles n'offre de solutions génériques capables de résoudre la plupart des configurations rencontrées dans des environnements réels. La difficulté dans ces situations, c'est que le robot doit trouver par lui-même les trajectoires efficaces même si elles sont obstruées, mais dont il est capable de proposer une solution pour **déplacer** ces obstacles en toute **sécurité**. Donc la navigation parmi les obstacles amovibles (NAMO) implique deux objectifs : (1) Trouver le meilleur compromis entre déplacer des obstacles ou les éviter, car parfois, il est plus simple de contourner un obstacle au lieu de le déplacer comme par exemples des obstacles lourds, volumineux, etc. (2) L'aspect de sécurité implique que le robot doit gérer les obstacles différemment selon leurs types. Le robot peut se permettre de pousser la chaise (si elle n'est pas occupée)

mais pas les personnes, car il suffit de leur demander de céder le passage, tel que c'est illustré dans l'exemple de la Figure 3.3.

On peut donc remarquer que la NAMO exige une compréhension de l'environnement pour agir en toute sécurité. La compréhension implique des compétences dans divers domaines tel que l'analyse de la scène, reconnaissance des obstacles, affordance, navigation, etc. Pour cela, nous proposons dans la suite de ce chapitre une approche basée sur une architecture robotique/cognitive capable d'atteindre ces deux objectifs.

Les faibles avancées en algorithmique qui traitent de la NAMO ont poussé les roboticiens à changer de point de vue. Certaines équipes s'inspirent des sciences cognitives pour aborder les problèmes de la NAMO en espérant comprendre et implémenter des mécanismes utilisés par le vivant. Depuis quelques années, des chercheurs tentent de mettre en évidence les intérêts que peuvent apporter les concepts de la simplexité et de la vicariance à la robotique. On peut citer par exemple les travaux du physiologiste Alain Berthoz [Berthoz and Petit, 2014] [Laumond, 2014] [Chatila, 2014] (c.f. Chapitre 2 Section 2.3) qui tentent de fédérer les roboticiens pour traiter les problèmes complexes tel que la NAMO en utilisant des techniques utilisées par le vivant [Berthoz and Petit, 2014]. Notre travail de recherche s'inscrit dans cette optique. Dans la suite de ce chapitre, nous décrivons une architecture robotique hiérarchique pour la navigation, dont le niveau décisionnel est couplé avec une architecture cognitive. Plus précisément, nous proposons dans nos travaux une architecture cognitive basée sur les principes de la simplexité et de la vicariance évoqués au chapitre 2, pour l'appliquer à la NAMO.

Dans le but de concevoir un tel système, qui permet aux robots d'avoir un comportement proche de celui d'un humain, en terme de navigation et d'interaction avec les différents acteurs de l'environnement, il est important de s'intéresser aux travaux en physiologie pour reproduire le comportement attendu. On peut citer par exemple Olivier Trullier et Alain Berthoz [Trullier et al., 1997] qui ont montré que les systèmes biologiques utilisent une représentation mentale de l'environnement (Cognitive Maps) pour planifier efficacement la navigation. Dans un système robotique, cela se traduit par l'utilisation de deux systèmes distincts, l'un pour la représentation (une architecture cognitive) et l'autre pour l'exécution du mouvement (architecture robotique).

Les auteurs préconisent de doter les robots d'une capacité de représentation d'un environnement dynamique, plutôt que sur des représentations statiques comme nous avons vu dans les architectures robotiques classiques décrites dans la seconde partie du chapitre 2. La modélisation d'un environnement à elle seule ne permet pas d'avoir qu'une abstraction de l'environnement, le choix des actions quant à lui doit faire appel à différents algorithmes selon les objectifs visés. Il est apparent qu'un tel système doit être composé de différents niveaux : modélisation, abstraction, planification, action, etc.

La planification a pour objectif de déterminer les actions à effectuer pour atteindre des objectifs. L'activité de planifier se base alors sur une modélisation du système de façon à contrôler les actions du robot pour atteindre ses objectifs. Dans le cas où le système est composé d'un ensemble d'obstacles de nature différente, ou même dans les cas où l'itinéraire vers l'objectif est obstrué, donc initialement hors d'atteinte, il est essentiel de fournir au robot une modélisation

du système permettant de prédire les états futurs de l'environnement ainsi que les conséquences de chaque action réalisée par le robot.

3.1 Outils et méthodes

Un robot est doté de capacités de navigation en milieu congestionné avec des obstacles de nature différente, doit être équipé pour les détecter, les pousser si nécessaire, communiquer avec eux, etc. Dans la suite de cette section, nous allons décrire le robot utilisé dans le cadre de cette recherche et les différents éléments matériels et logiciels qui le constitue. Il est essentiel aussi de déterminer plus formellement les conditions de l'environnement, la nature des obstacles et la nature de la commande de l'utilisateur, ces contraintes permettent non seulement de délimiter un périmètre de travail, mais aussi servir de métriques pour comparer les résultats obtenus lors des expérimentations. Pour cela, nous allons aussi délimiter l'environnement (type d'obstacles, leur nature, nature du sol, etc.) auquel notre robot est destiné.

3.1.1 Description de notre robot

L'espace des configurations du robot noté $\mathcal{C}$ est l'espace de la représentation paramétrique des postures du robot, c'est-à-dire l'ensemble des positions accessibles par le robot. Les coordonnées permettant de décrire la position du robot $(q_1, q_2, \ldots, q_n)$ sont appelées coordonnées généralisées, elles sont inférieures ou égale à la dimension de $\mathcal{C}$.

Un robot mobile est décrit comme une machine avec une base mobile. Dans ce paragraphe, nous fixons quelques principes que doit implémenter un robot avec une base mobile à roues qui se déplace dans un repère quelconque noté $\mathcal{R} = (O, \vec{x}, \vec{y}, \vec{z})$. Le robot est considéré comme un repère mobile noté $\mathcal{R}' = (O', \vec{x'}, \vec{y'}, \vec{z'})$ voir Figure 3.6. Le point origine mobile O' est un point remarquable de la base mobile, il correspond généralement au centre de l'axe des roues motrices de la base mobile.

Les deux robots mis en œuvre dans le cadre de nos travaux sont illustrés dans les figures 3.4 et 3.5. Les implémentations et simulations présentées dans ce travail sont développées essentiellement sous ROS, certaines situations sont illustrées à travers des simulations qui sont réalisées sous Gazebo et Rviz.

Le robot est équipé de capteurs/actionneurs suivants :

- Caméra RGB, caméra de profondeur (Real Sense RS300)
- Deux servomoteurs / odomètres (Dynamixel)
- LIDAR (RPLIDAR 360)
- Capteur de pression (cellule de charge)
- Gyroscope, accéléromètre et magnétomètre (Capteur AHRS 9 axes)
- Haut-parleur (Buzzer)

FIGURE 3.4 : Robot utilisé lors des premières simulations, nous l'avons entièrement conçu. Cette plateforme a été bondonnée au profit d'une *Turtlebot3*.

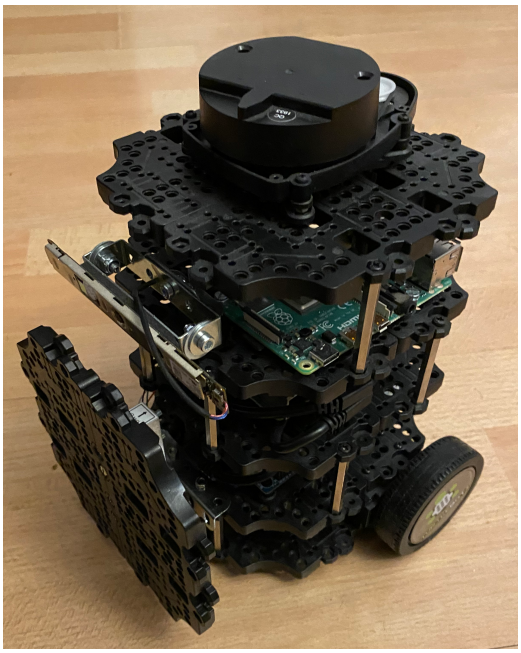


FIGURE 3.5 : Le robot Turtlebot 3 Burger sur lequel nous avons ajouté une caméra intel real sens SR300 et un capteur de force pour pousser les objets.

Le robot que nous mettons en œuvre ici permet deux actions possibles :

- **Déplacer les obstacles par poussée :** La seul façon de déplacer un obstacle pour notre robot et de le pousser dans une direction d avec une force $\vec{F}$.
- **Interaction :** Le robot peut interagir avec des humains et/ou robots par un signal sonore/envoie de commandes, pour leur indiquer de s'éloigner dans une direction donnée.

La base mobile est équipée de deux roues non-holonomes pour former une configuration unicycle. Les deux roues motrices sont disposées sur l'axe transversal, pour assurer l'équilibre, deux roues folles supplémentaires sont disposées sur l'axe longitudinal.

Dans le cas d'une roue classique (non-holonome) le déplacement de la roue par rapport au sol s'effectue dans l'axe perpendiculaire à l'axe de rotation de la roue, contrairement aux roues holonomes où le déplacement est possible sur deux axes. Donc un robot équipé avec des roues non-holonomes ne peut pas accéder instantanément à certaines positions, par exemple dans le cas de translation dur des positions parallèles à l'axe de la roue.

On appelle la situation d'un robot noté ξ la position et l'orientation de l'origine du robot mobile O', x et y représente l'abscisse et l'ordonnée de ce point dans R, θ est l'angle $(\vec{x}, \vec{x'})$ dans l'espace tridimensionnel noté $\mathcal{M}$.

$$\xi = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix}$$

Dans le cas d'un robot mobile avec une base sans articulations (base mobile monobloc), il est facile de connaître la position de tous les points par rapport à l'origine O'. Ainsi, on peut définir la configuration du robot mobile par un vecteur :

$$q = \begin{pmatrix} q1 \\ q_2 \\ \vdots \\ q_n \end{pmatrix}$$

Le centre instantané de rotation passe forcément par l'axe des deux roues (d et g). Le rayon de la courbure de trajectoire du robot est noté ρ, qui représente la distance entre le CIR et le point O' (voir figure 3.6). Soit la distance entre les roues notée L et la vitesse de trajectoire de la base mobile notée ω, on peut alors déduire la vitesse des roues v_d et v_g comme suite :

$$v_d = -r\dot{\varphi}_d = (\rho + L)\omega$$
$$v_d = r\dot{\varphi}_g = (\rho - L)\omega$$

On peut déterminer alors ρ et ω comme suite :

$$\rho = L\frac{\dot{\varphi}_d - \dot{\varphi}_g}{\dot{\varphi}_d + \dot{\varphi}_g}$$
$$\omega = -\frac{\rho(\dot{\varphi}_d + \dot{\varphi}_g)}{2L}$$

On peut constater que le CIR est situé exactement sur l'axe des deux roues. Ce qui procure à ce type de robot des propriétés de mouvement très intéressantes, car si $\dot{\varphi}_d = -\dot{\varphi}_g$ alors le robot avance tout droit, dans le cas où $\dot{\varphi}_d = \dot{\varphi}_g$ alors le robot tourne sur lui-même. Avec ces deux actions (*avancer en ligne droite* et *rotation*) on peut décomposer le mouvement d'un déplacement complexe en une combinaison de ces deux actions. Ce qui procure un avantage certain pour ce type de configuration.

Perception

La perception recouvre le volet d'acquisition d'informations sur l'environnement du robot à travers des capteurs qui vont mesurer des grandeurs physiques avec une certaine précision. Les mesures données par les capteurs dépendent de la sensibilité et de leur position. L'information brute fournie par les capteurs n'est pas suffisante à elle seule pour extraire l'information utile permettant au robot d'effectuer sa tâche. Pour cela, la perception intègre aussi les fonctionnalités de filtrage d'information, de détection, de segmentation, de suivi, d'identification et d'interprétation.

Le robot décrit ici, est équipé de différents capteurs lui permettant de percevoir son environnement. Des capteurs proprioceptifs : deux odomètres qui équipent chaque roue motrice et un accéléromètre. Des capteurs extéroceptifs : un capteur Intel Real Sens (qui regroupe une caméra RGB et une caméra de profondeur).

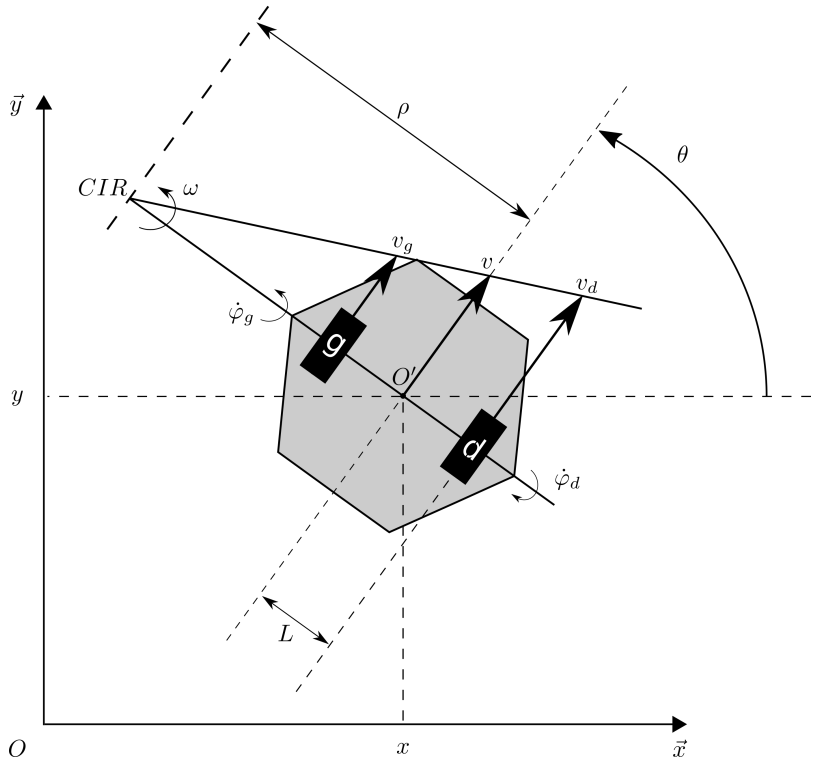


FIGURE 3.6 : Centre instantané de rotation

Les capteurs sont classés en deux catégories. Les capteurs *proprioceptifs* qui donnent des informations sur l'état interne du robot (odomètres, gyroscopes, accéléromètre, etc.), extéroceptifs qui donnent des informations sur l'état du monde extérieur (caméra, télémètres ultrasons, LIDAR, GPS, etc.).

L'utilisation de capteurs engendre des erreurs de mesures dues soit à la précision des capteurs ou à l'environnement. Le filtrage de données consiste à traiter les données issues des capteurs afin de détecter des erreurs de mesures. En robotique, il est souvent utile d'utiliser des données issues de plusieurs capteurs afin de réduire les erreurs, cette technique connue sous le nom de *la fusion de données*, elle a pour but d'estimer l'état du système à partir des observations, traditionnellement en robotique, on fait plus souvent appel à la technique tel que le filtrage de Kalman [Katzfuss et al., 2016], le filtrage à particule [Thrun, 2002].

- **Perception par télémétrie :** Les capteurs par télémétrie mesurent des distances entre le capteur et les surfaces solides. Les capteurs les plus répondus utilisent généralement des lasers ou des ultrasons, ils mesurent le temps aller/retour d'une onde (Laser, ultrason, infrarouge, etc.) et déduit la distance. À partir de ces mesures, il est possible d'établir un nuage de points dans un repère attaché à la position du capteur (voir Figure 3.32).

  Parmi les capteurs télémètres populaires, on trouve le capteur de télédétection par laser ou LIDAR[2] (On peut voir ce LIDAR intégré aux deux robot Figure 3.4 et 3.5 présenté précédemment) ils offrent de bonnes performances en terme de précision en milieu intérieur en raison de la faible luminosité et à courte distance avec un prix raisonnable. Mais ils

[2]Acronyme anglaise pour : *light detection and ranging*

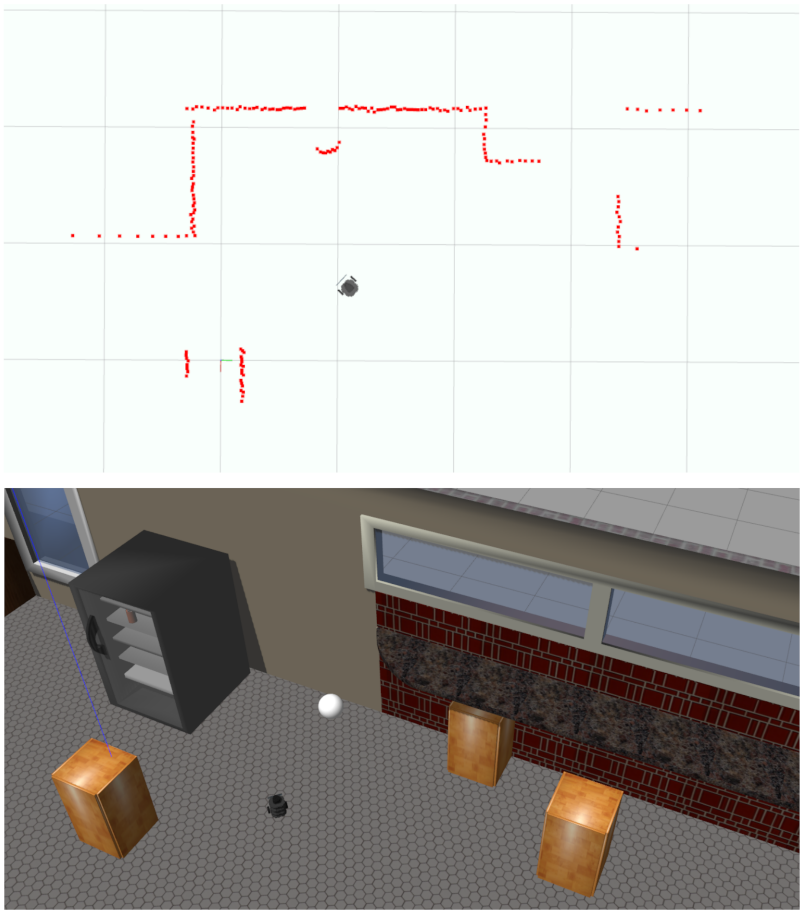


FIGURE 3.7 : En bas de l'image le robot Turtlebot 3 équipé d'un LIDAR dans un environnement domiciliaire simulé sous Gazebo. En haut, on voit, via RVIZ, le nuage de points généré par le LIDAR.

restent sensibles à certaines surfaces des obstacles (e.g. matériaux transparents) et les inclinaisons qui peuvent réfléchir la lumière dans une autre direction, dans ce cas, ils fournissent des données incohérentes avec la réalité.

- **Perception par vison directe :** Les caméras sont aussi très utilisées en robotique mobile. Parmi les plus répondues, on trouve les caméras RGB qui opèrent dans le domaine du spectre visible. Elles permettent de fournir des images riches en informations, mais pas directement exploitables, de plus, elles ne couvrent généralement qu'une petite partie de l'environnement immédiat du robot. Il n'est pas facile d'utiliser les caméras RGB pour obtenir des données métriques, elles sont surtout utilisées pour obtenir des informations sur la nature des obstacles en utilisant des techniques de reconnaissance de formes comme on peut le voir sur la Figure 3.8.

Un autre type de caméra connu sous le nom de *capteur de profondeur* qui utilise les infrarouges pour obtenir des informations de distance sous forme d'un nuage de points en 3D. Ce type de capteurs offre des possibilités diverses en terme d'analyse de scène (en utilisant des algorithmes sur les nuages de points), il permet par exemple d'estimer la taille d'un objet, sa distance par rapport au capteur, son orientation, la surface de

FIGURE 3.8 : Exemple de reconnaissance de formes en utilisant une caméra RGB. Cette illustration est réalisée grâce à la caméra intégrée au capteur Kinect Xbox, implémenté sur notre robot illustré à la Figure 3.4

contact au sol, etc. Ces informations sont importantes dans le cas où le robot est amené à manipuler les obstacles, comme dans le cas de la NAMO.

On trouve dans le commerce quelques capteurs qui englobent ces deux types de caméras (RGB, et capteur de profondeur) dans un seul dispositif (voir Figure 3.9), on peut voir par exemple un exemple de rendu dans la Figure 3.10. Ces capteurs sont utilisés initialement dans les consoles de jeux pour interagir directement avec l'utilisateur en captant ses mouvements.

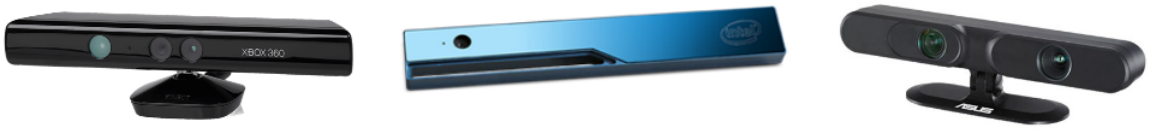


FIGURE 3.9 : Capteurs RGB et capteur de profondeur en un seul dispositif. (a) Kinect Xbox, intégrée à notre robot illustré à la Figure 3.4. (b) Intel RealSens (utilisée sur notre robot) intégrée à notre robot illustré à la Figure 3.5. Les performances de ce capteur en terme de qualité d'image et de rapidité dépasse les deux autres types illustrés ici. (c) Azus Xtion PRO, un autre capteur du même type d'une marque concourante, également très populaire en robotique.

- **Localisation absolue :** Le robot utilise deux capteurs odomètres (un sur chaque roue motrice) pour estimer le déplacement de la base mobile. L'odometrie est une technique permet de donner la posture relative de la base mobile par rapport à une position initiale. L'odométrie est souvent utilisée en robotique en raison de la simplicité de mise en œuvre et son faible coût. Cependant, elle présente aussi des mesures particulièrement imparfaites, car le calcul de la posture de la base mobile se fait en supposant que les conditions sont parfaites. C'est-à-dire que le glissement des roues est nul et les paramètres géométriques du robot sont connus : longueur de l'entraxe, diamètre des roues, l'alignement des roues

FIGURE 3.10 : Exemple du rendu en RGB et Nuage de point obtenu à l'aide du capteur Kinect. (a) Environnement du robot. (b) Camera RGB. (c) Nuage de point.

est parfait, etc. Le calcul permettant de déterminer la posture de la base mobile à l'instant t se fait simplement en intégrant les dérivées des vitesses comme suit :

$$x(t) = \int_0^t \dot{x}(\tau)d\tau$$
$$y(t) = \int_0^t \dot{y}(\tau)d\tau$$
$$\theta(t) = \int_0^t \dot{\theta}(\tau)d\tau$$

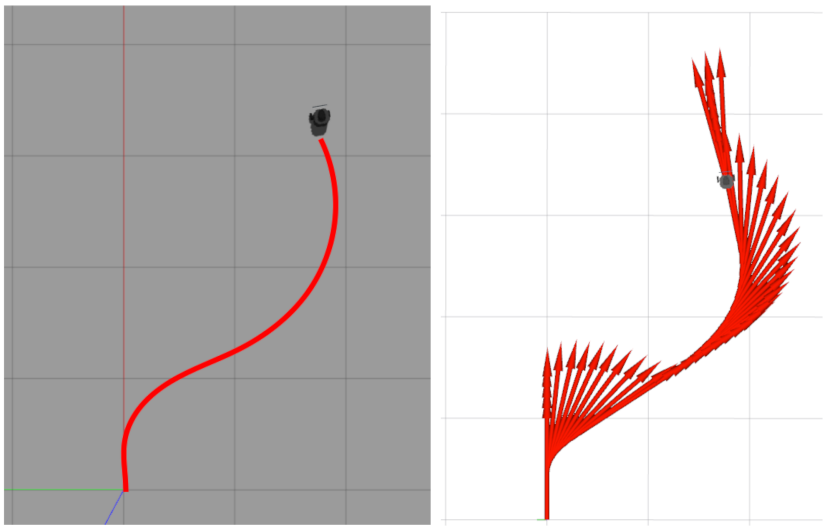


FIGURE 3.11 : À gauche la trajectoire du robot *TurtleBot 3* sous le simulateur *Gazebo*, et à droite la visualisation sous *RViz* des vecteurs vitesse orientation des positions prise par le robot à chaque instant de son déplacement.

- **Bouger les obstacles par poussée :** Le robot est équipé d'une plaque lui permettant

de pousser les obstacle. La plaque est monté sur un capteur de force lui permettant de mesurer la force de pousser (voir Figure 3.12).

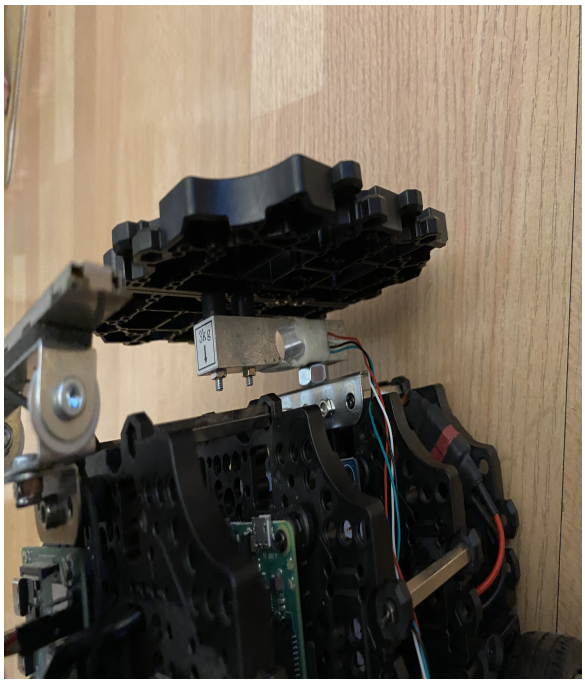


FIGURE 3.12 : Plaque permettant au robot de pousser les obstacles avec son capteur de force.

3.2 Description de l'environnement

La NAMO vise essentiellement à la navigation en milieu domiciliaire structuré. Dans le cadre de cette étude, nous nous limitons à un environnement avec les caractéristiques suivantes :

Le sol

Nous considérons que le sol est uniforme sur l'ensemble de l'environnement, c'est-à-dire un coefficient de frottement avec une variance minimale. Nous considérons aussi que le sol est plat et ne présente pas des différences de niveaux tel que les escaliers, les rampes, etc.

Les obstacles

Nous avons identifié quatre types d'obstacles, catégorisés comme suit (voir Figure 3.13) : (I) Les obstacles statiques, qui sont des objets qui occupent une position fixe dans l'environnement, ils sont divisés en deux catégories : (1) Les obstacles **fixes**, que le robot n'as pas le droit de bouger, parce que soit leur position est déterminée par des règles sociales (exemple : on ne doit pas bouger une table pour passer, c'est valable aussi pour les meubles, les objets de décoration, etc.), et/ou parce qu'ils sont fragiles, par exemple un instrument de musique qui traîne par terre, un objet en verre, etc. (2) Les obstacles **amovibles**, que le mobile a le droit de bouger tel que les chaises, les jouets, etc. (II) Les obstacles dynamiques, sont des entités qui se déplacent dans l'environnement, elles sont divisées en deux catégories : (3) Les obstacles

interactifs, sont les obstacles amovibles avec qui le mobile peut entrer en contact, ils sont généralement limités à deux entités : les humains et d'autres mobiles qui partagent le même environnement. Du point de vue du mobile, ces obstacles peuvent lui céder le passage s'il en fait la demande. (4) Les obstacles **non-interactifs** : sont les obstacles avec qui le mobile ne peux pas communiquer, ils sont généralement limités aux animaux de compagnie qu'on trouve dans les milieux domiciliaires.

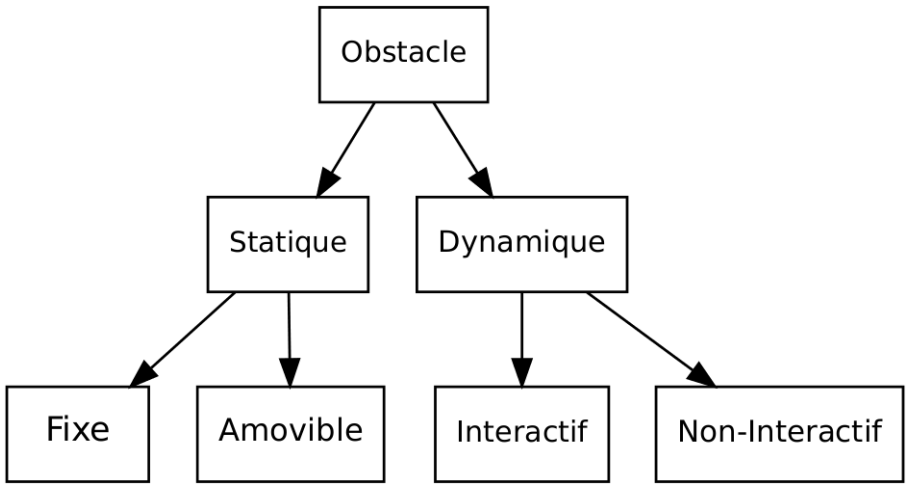


FIGURE 3.13 : Les types d'obstacles considérés dans le cadre cette étude.

On propose ici quelques exemples :

- **Des obstacles fixes :** le robot ne peut pas bouger ces obstacles, il doit obligatoirement éviter les collisions et tout contact avec ce type d'objets par exemple : tables, télé, mobiliers, objets de décoration tels que les vases les plantes de décoration, etc.
- **Des obstacles amovibles :** le robot peut bouger ces obstacles par simple poussée, par exemple les chaises, des jouets, meubles avec roulettes prévues à cet effet, etc.
- **Des humains et/ou d'autres robots :** le robot doit obligatoirement éviter tout contact ou collision, en revanche, il peut interagir avec eux pour demander de libérer le passager.
- **Des animaux de compagnie :** obstacles au comportement imprévisible. Le robot doit obligatoirement éviter tout contact ou collision avec ce type d'obstacles, de plus le robot est incapable de communiquer avec eux.

Le but à atteindre

Nous avons fait le choix que la position exacte du but à atteindre n'est pas connue à l'avance, seulement une direction approximative est indiquée au robot ainsi que le l'objet à trouver[3].

[3]Nous avons choisi cette façon de déterminer l'objectif du robot dans la perspective de faire un robot réellement utilisable en environnement domicilier. L'une des perspectives envisageable est de montrer du doigt une direction au robot et de lui indiquer l'objet à trouver. L'autre perspective et de pouvoir appeler le robot pour qu'il vienne rejoindre l'utilisateur, pour cela, il suffit que le robot détermine la direction approximative d'où vient le son.

- **Assigner un but :** Le robot reçoit les instructions (le but à atteindre) sous forme d'une direction θ et un objet à trouver. La direction indiquée au robot peut être approximative dans les limites suivantes :

$$\theta = 2 * \arccos\left(\frac{r}{d}\right)$$

d : Distance de l'objet par rapport à la position initiale du robot.
r : Portée du capteur.

- **Objectif à trouver :** Nous utilisons la caméra RGB et le système de reconnaissance d'objets pour déterminer si l'objet recherché se trouve la liste des objets détectés par le capteur RGB.

3.3 Analyse de la problématique

Cette section présente notre architecture robotique pour la NAMO baptisée VICA (**VI**carious **C**ognitive **A**rchitecture). Elle a pour but de permettre à un mobile de se déplacer en toute sécurité dans un environnement domiciliaire classique avec les caractéristiques présentées précédemment. La réalisation d'un tel robot capable de se déplacer et de trouver un chemin optimal dans un tel environnement soulève différentes questions, à savoir :

- **Quelle est la nature de l'obstacle ?** Afin de déterminer si le robot peut déplacer et comment déplacer un obstacle en toute sécurité (c'est-à-dire déterminer le type d'action à entreprendre et comment le faire.), il doit connaître la nature de l'obstacle (fixe, amovible, interactif ou non-interactif) avec une bonne certitude.

  Pour illustrer cela, un obstacle fixe doit être évité contrairement à un obstacle amovible où il faut déterminer la meilleure façon de le bouger (déterminer la trajectoire dans laquelle il faut le pousser). Un objet amovible avec lequel le robot peut interagir nécessitera seulement une demande du robot pour que ce dernier lui laisse le passage. Finalement, un objet amovible avec lequel le robot ne peut pas interagir, peut nécessiter que le robot attende que celui-ci se décale par lui-même ou bien l'éviter simplement.

  Dans certains cas, le type d'obstacle ne permet pas de déterminer avec certitude sa nature (amovible ou non) dans le cas par exemple qu'une chaise de bureau, elle est amovible si seulement si elle n'est pas occupée par une personne.

- **Est-il capable de bouger l'obstacle ?** La nature des obstacles détermine en partie le type d'action à entreprendre pour les éloigner du passage. Mais la méthode de déplacement à elle seule n'est pas suffisante, car la configuration de l'environnement rajoute une contrainte supplémentaire, on peut illustrer cette situation par exemple lors de la poussée d'un obstacle bloqué par un autre obstacle. On peut citer d'autres cas de figure, par exemple, il est plus facile de pousser une chaise avec des roulettes, comparé à une chaise sans, dans le cas où le sol est rugueux. Dans un autre exemple avec deux cartons l'un plein et l'autre vide, ici, il s'agit deux objets du même type, mais le mobile ne peux déplacer qu'un seul des deux.

- **Est-il intéressant de choisir de passer par là ?** Pour un robot mobile qui se déplace dans un milieu congestionné, décaler un obstacle pour passer ne doit pas être une fin en soi. Car l'action de décaler un objet peut être énergivore, chronophage et apporte des changements à l'environnement, ce qui nécessite une modification du modèle de l'environnement que le robot a précédemment calculé. Le robot doit disposer d'une métrique afin de déterminer si un déplacement d'obstacle peut apporter une valeur ajoutée. Parfois, il est plus simple de contourner des obstacles au lieu de se lancer dans la résolution d'une situation qui peut être assimilée à la résolution d'un jeu de taquin.

Les réponses aux questions posées ici impliquent un haut niveau d'abstraction. Nous pensons qu'une approche algorithmique simple (tel que les approches proposées dans le Chapitre 2) ne permettra pas de répondre à toutes ces questions en même temps. Or, on peut remarquer que la navigation dans un tel environnement pour un être humain est simple et naturelle. Donc, nous pensons qu'un robot qui évolue dans un environnement domiciliaire doit montrer un comportement proche[4] de celui des humains afin qu'il puisse se déplacer efficacement et en toute sécurité. Il est attendu qu'il puisse écarter les obstacles, éviter des collisions avec les objets fixes et les humains, trouver des trajectoires efficaces, interagir avec les personnes qui lui bloquent le passage, éviter les animaux de compagnie, anticiper le comportement des différents acteurs de l'environnement etc.

Nous avons montré précédemment (c.f. Chapitre 2) que la NAMO en environnement domiciliaire nécessite des algorithmes exponentiels, c'est-à-dire non-solvables en un temps raisonnable (polynomiale) [Moghaddam and Masehian, 2016]. Certains planificateurs (c.f. première partie du chapitre 2) permettent de résoudre les problèmes posés par la NAMO dans certains cas spécifiques (obstacles de forme géométrique uniquement, déplacement des obstacles dans la même direction, obstacles connus par avance, etc.). À cause de minima locaux ces algorithmes ne proposent pas de solutions génériques et ne sont que partiellement applicables en environnement domiciliaire.

Dans la seconde partie de l'état de l'art (chapitre 2 section 2.2) nous avons présenté des techniques qui consistent à aborder les problèmes des algorithmiques complexes d'un autre point de vue, elles proposent des systèmes appelés *architectures* (robotique/cognitives) qui permettent respectivement de décomposer les problèmes et d'extraire des représentations abstraites et simplifiées, pour finalement appliquer des algorithmes spécialisés et être capables de trouver des solutions en un temps raisonnable. Nous proposons donc d'aborder la NAMO à travers les architectures robotiques et cognitives, qui ont pour spécificité la flexibilité et la généralisation de problèmes. Nous pensons qu'il est raisonnable de proposer une architecture robotique couplée avec une architecture cognitive comme solution pour une telle problématique, car elle permettra de doter le système de capacités d'abstraction (donc simplification) et fait intervenir des compétences dans divers domaines, pour les appliquer selon la situation à différents moments.

Le choix d'appliquer les architectures cognitives pour la résolution de la NAMO s'impose, car comme nous l'avons vu dans l'état de l'art, les solutions algorithmiques pures offrent un cadre restreint pour aborder la NAMO en raison de la multiplicité des problèmes de nature différente. Pour résoudre le problème de la NAMO, il faut avoir plusieurs niveaux d'abstraction

[4]Nous entendons par 'proche du comportement humain, c'est la manière dont les humains se comportent pour déplacer, éviter et manipuler les obstacles qui gênent le passage.

pour saisir la complexité de l'environnement, et recourir à un système permettant d'appliquer des techniques selon la configuration dans laquelle le robot se trouve à un moment donné. Nous pensons qu'une solution algorithmique monolithique sera difficile à mettre en œuvre.

La planification et le contrôle d'un robot mobile consistent à la résolution de problèmes complexes. Car l'environnement congestionné implique un espace, des configurations et des actions possibles important. De plus, la perception et les connaissances des robots et de leur environnement sont limitées et elle peut évoluer dans le temps en raison de l'exploration et de l'expérience au fil du temps. Une telle complexité ne permet pas de pré-programmer l'ensemble des commandes potentielles à appliquer dans l'environnement, car une telle démarche obligerait une énumération de toutes les successions de situations possibles jusqu'à atteindre tous les objectifs. Même une solution plus élaborée permettant une modélisation statique de l'environnement ne permettrait pas de résoudre le problème d'une façon efficace, car l'environnement dont il est question est dynamique, il faut donc modéliser non seulement l'environnement tel qu'il est observé, mais aussi décrire le comportement futur de celui-ci, car il est en constante évolution et le comportement même du robot peut l'altérer avec sa capacité de déplacement et d'interaction avec les obstacles.

Les architectures hiérarchiques en robotique permettent de séparer la problématique de contrôle en plusieurs niveaux. Chaque niveau contrôle le niveau inférieur. Plus on monte dans l'architecture, plus les algorithmes travaillent sur des représentations abstraites.

3.4 Description générale de notre approche

Nous proposons donc une architecture robotique dotée de deux planificateurs. Un planificateur global qui permet au robot de planifier le chemin optimal pour atteindre son but, en faisant abstraction des obstacles. Un second planificateur, local, qui permet au robot gérer les obstacles sur le chemin. Une fonction d'évaluation permet de sélectionner le planificateur adéquat selon une fonction de coût.

Le planificateur global opère en environnement partiellement inconnu, il se base essentiellement sur les espaces libres pour construire un graphe. Un algorithme de calcul de chemin permet d'indiquer la direction à suivre pour atteindre l'objectif. Cet algorithme que nous avons développé et nommé H^*, appartient à une famille connue sous le nom *bug algorithms* (algorithmes inspirés d'insectes) [McGuire et al., 2019], le but étant de rapprocher le robot au plus près de l'objectif jusqu'à atteindre une situation dans laquelle il ne puisse plus avancer à cause des obstacles, ou lorsque le contournement d'obstacles demande un coût supplémentaire important. Dans ce cas, le système fera appel au planificateur local pour déterminer s'il y a une possibilité de décaler les obstacles et estimer le coût de cette action, finalement une décision de déplacer les obstacles ou de les contourner sera prise.

Le planificateur local permet la gestion des obstacles par poussée (le robot pousse les obstacles pour les écarter du passage). Le planificateur créé une représentation de l'environnement immédiat (proche) sous forme d'un système multi-agent, par la suite, il effectue des simulations sur cette représentation dans le but de déterminer les actions possibles et aussi déterminer un coût pour cette action avant même d'avoir agi. La représentation de l'environnement ne va pas

se limiter à la représentation des différents acteurs qui composent l'environnement immédiat, mais aussi des lois qui régissent l'environnement et ses acteurs. Donc cette représentation s'apparente à un moteur physique[5]. Lors de l'action, à chaque instant du mouvement, le système aura vérifié l'état dans lequel certains de ces capteurs devraient être afin de renforcer son apprentissage ou de remettre en question l'action qu'il est en train de réaliser.

Discussions : Pourquoi utiliser les systèmes multi-agent pour la représentation des obstacles La représentation des connaissances en IA classique (Ontologies, Catégories...) a montré ses limites, pour cela, nous souhaitons tenter une approche multi-agents pour l'organisation et la représentation de l'information. Le choix d'utiliser les systèmes multi-agents est lié au fait que les systèmes multi-agent (SMA) ont montré leurs efficacités dans la simulation de l'environnement. Habituellement, les SMA sont utilisés pour deux de leurs caractéristiques : La première est la résolution de problèmes de manière distribuée et la seconde est la simulation de phénomènes complexes. C'est pour cette seconde raison que nous avons choisi l'utilisation des SMA. Car l'environnement représente un système complexe du fait de son évolution perpétuelle au fil du temps, cette modification incessante est dû aux agents dynamiques qui évoluent dedans. Chaque agent dans l'environnement peut être statique comme par exemple un objet inerte, un autre type d'agent dynamique peut évoluer dans l'environnement et interagir avec d'autres agents par exemple un robot, un animal de compagnie ou une personne avec laquelle on peut communiquer, ce type d'agents modifie l'environnement (nous reviendrons plus en détail sur cet aspect dans le chapitre 5).

La fonction de coût

Le planificateur global dispose d'une représentation partielle de son environnement sous forme d'un graphe. Si l'objectif à atteindre est en-dehors des zones connues, alors l'algorithme suppose que l'objectif est plus loin de deux fois le rayon du capteur, du nœud le plus proche de l'objectif connu dans la direction de l'objectif. Le nœud le plus proche[6](distance optimale) de l'objectif est déterminé selon un algorithme décrit dans le Chapitre 3.

L'estimation du coût C_{global} du planificateur globale est calculée en fonction de la distance à parcourir dans les zones connues auquel on ajoute la distance à parcourir dans la zone inconnue ($2 \times rayon\ du\ capteur$).

$$C_{global} = \left(\sum S_i\right) + 2 \times r$$

$$C_{global} : Coût\ du\ planificateur\ globale$$

$$S_i : segments\ du\ graphe\ parcouru$$

$$r : rayon\ du\ capteur$$

[5]Ensemble logiciel permettant la résolution de problèmes de la mécanique classique par exemple les collisions, la chute des corps, les forces, la cinétique, etc.

[6]Le nœud le plus proche, représente le nœud le plus prometteur, susceptible d'aboutir à un chemin optimal et non pas le plus proche en terme de distance.

Le planificateur local s'occupe de la gestion des obstacles. Si aucun obstacle n'est détecté à proximité, il retourne un coût d'une valeur infinie. Dans le cas contraire, il effectue une simulation pour dépasser les obstacles et atteindre une position favorable pour continuer le déplacement. Pour effectuer les simulations, le système utilise un système multi-agents (décrit dans le Chapitre 5). Le coût de la gestion des obstacles prend en compte plusieurs paramètres : La taille de l'obstacle, sa surface en contact avec le sol, le coefficient de frottement[7] du sol et la distance de la poussée. Lors de la gestion des obstacles, plusieurs obstacles peuvent être déplacés, donc le coût C_{local} somme de l'ensemble de ces mouvements est calculé comme suit :

$$C_{local} = \sum_{i=1}^{N} \left(\sum_{j=1}^{M_i} \overrightarrow{F_j} \times \varphi i \times D_j \right)$$

$$c_{local} : Coût\ du\ planificateur\ local$$

$$D_j : Distance\ de\ poussé\ lors\ du\ mouvement\ j$$

$$N : Nombre\ d'obstacles$$

$$M_i : Nombre\ de\ mouvements\ en\ ligne\ droite\ pour\ un\ obstacle\ i$$

$$\varphi i : Coefficient\ de\ frottement\ de\ l'obstacle\ i$$

$$\overrightarrow{F_j} : Force\ de\ poussé\ du\ robot\ pour\ effectuer\ le\ déplacement\ D_j$$

Le choix du planificateur est sélectionné selon le coût le plus faible[8]. Lors des simulations, nous avons constaté un choix systématiquement porté sur le planificateur global, car sa valeur est toujours inférieure par rapport à C_{local}. Pour cela nous avons introduit un biais pour la compenser.

$$planner = min(C_{local}, C_{global} \times b)$$

$$planner : planificateur\ choisi$$

$$b : biais$$

3.4.1 Illustration d'un cas pratique

Pour mettre en évidence le fonctionnement de notre approche, nous allons présenter dans cette section une illustration avec un robot mobile qui évolue dans un environnement de bureau congestionné, avec des situations divers (espaces libres, espaces congestionnés face à des obstacles fixes, amovibles, etc.). L'objectif est de montrer le comportement attendu du robot à travers un scénario et montrer aussi l'enchaînement des processus tel qu'ils sont prévus.

[7]Le coefficient de frottement est calculé selon la taille et la surface de contact au sol de l'obstacle ainsi que le coefficient de frottement du sol.

[8]Les coût calculés ci-dessus sont des grandeurs adimensionnelles [9].

La Figure 3.14, illustre un scénario de fonctionnement attendu du robot, dans un environnement de bureau congestionné. Les passages possibles sont obstrués par des humains, des obstacles fixes ou amovibles. Dans cette illustration, le robot a pour objectif de rejoindre la position illustrée par la croix rouge (le but est de trouver le téléphone sur la table) dans le schéma, l'utilisateur indique au robot la direction dans laquelle se trouve son objectif (flèche verte dans la Figure 3.14). Le planificateur global détermine le chemin optimal pour atteindre son objectif (le fonctionnement du planificateur global sera décrit dans le chapitre 3). Le planificateur global conduit le robot à la situation (1) illustrée dans la Figure 3.15. Dans cette seconde illustration, le planificateur global est incapable de trouver un chemin en raison des obstacles, dans ce cas, il fait appel au planificateur local (le fonctionnement du planificateur local sera décrit dans le chapitre 5).

La Figure 3.15 illustre le robot dans une situation de blocage, les passages possibles sont obstrués par les personnes qui discutent à gauche et le mobilier de bureau à droite. Le planificateur global analyse la situation et calcul un coût de déplacement des chaises ou demander aux personnes de céder le passage, dans ce scénario, on considère que la solution retenue (situation qui présente un coût moindre) consiste à envoyer un message aux personnes afin de lui céder le passage.

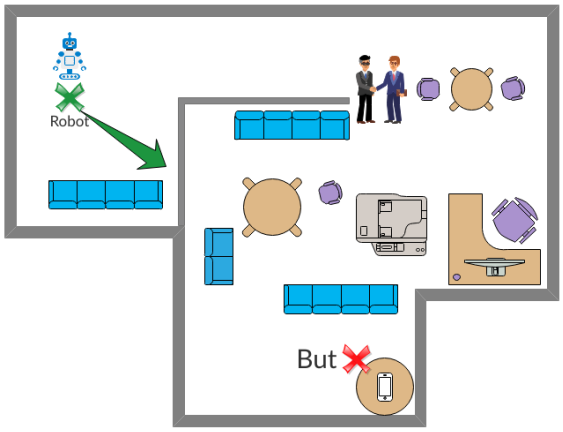


FIGURE 3.14 : Position initiale du robot, la flèche verte montre l'instruction de l'utilisateur qui montre la direction de l'objectif à atteindre. L'objectif à atteindre par le robot est le téléphone sur la table (but).

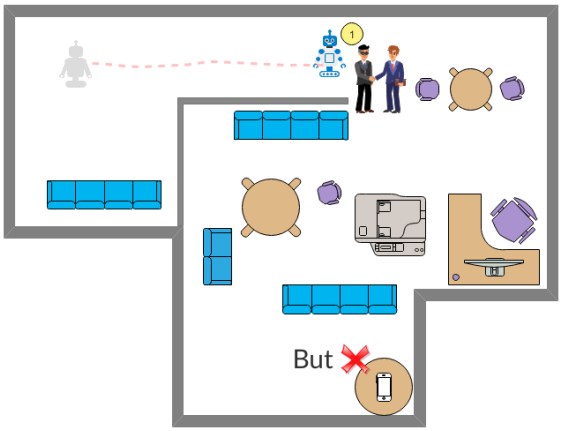


FIGURE 3.15 : Les passages possibles sont obstrués par les deux personnes qui discutent à gauche et par la table et les deux chaises à droite.

La Figure 3.16 montre la réaction des personnes face au message du robot. Le planificateur global conduit le robot dans une nouvelle situation de blocage face à des obstacles fixes et amovibles illustrée dans la Figure 3.17. L'appel au planificateur local est systématique dans le cas d'un blocage.

Le planificateur local fait le choix de déplacer la chaise (obstacle amovible) qui obstrue le passage, cette situation est montrée dans la Figure 3.18. Le planificateur global reprend la main et conduit le robot à son objectif 3.19.

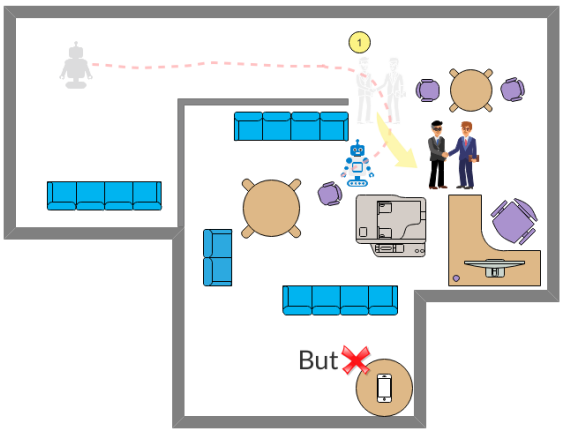


FIGURE 3.16 : Cette illustration montre la réaction des personnes au message du robot, qui cèdent le passage. Ce qui conduit le robot à nouvelle situation.

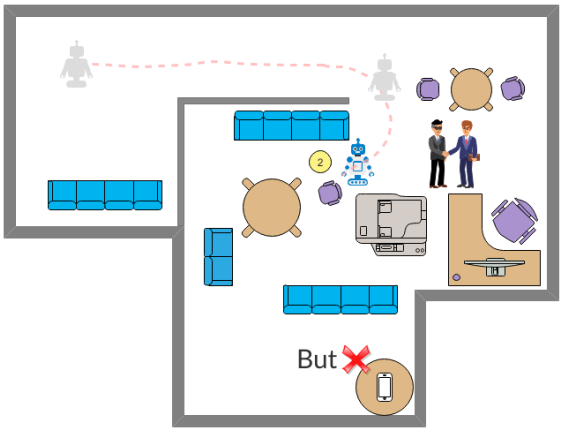


FIGURE 3.17 : La situation (2) montre le robot en situation de blocage, devant deux obstacles fixes (imprimante à droite et la table à gauche) et un obstacle amovible (la chaise).

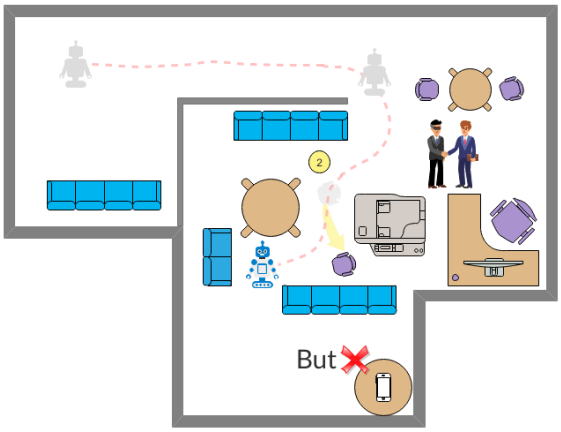


FIGURE 3.18 : Le robot pousse la chaise qui gêne le passage pour se frayer un chemin jusqu'à atteindre une position où le planificateur global reprend la main.

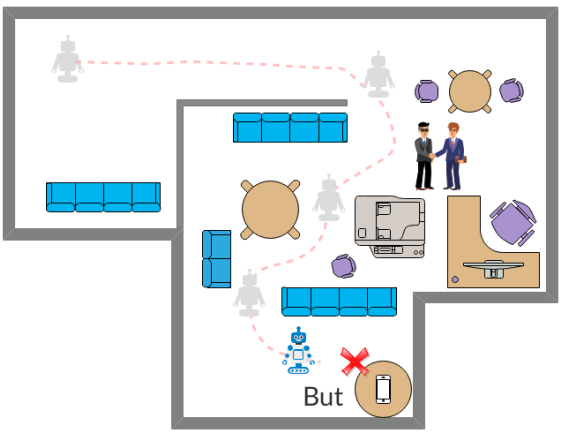


FIGURE 3.19 : Le planificateur global conduit le robot dans la direction de l'objectif, jusqu'à détecter l'objet recherché. Le robot a atteint son objectif et se met en pause.

Nous avons montré à travers un scénario idéal qui illustre l'enchaînement de deux actions principales, planification globale et planification locale, qui ont pour objectif respectivement la navigation du robot en zones dégagées et la gestion des obstacles en situation de blocage. La sélection et le basculement entre planificateurs se font sur la base d'une fonction de coût. La planification globale se base uniquement sur la direction indiquée par l'utilisateur pour décider de la direction de la navigation et continue l'exploration jusqu'à trouver l'objet. Pour permettre la mise en œuvre d'un tel système capable de reproduire cet enchaînement. Dans la section suivante, nous allons poser des contraintes sur l'environnement du robot et les conditions d'utilisation de l'architecture proposée.

3.5 Description de l'architecture VICA

Pour construire notre architecture robotique, nous avons retenu le modèle d'architecture hiérarchique, elle est composée de trois niveaux : *(1) Le niveau fonctionnel* permet de gérer les éléments matériels qui composent le robot, ce niveau permet l'interfaçage avec l'environnement à travers des capteurs et des actionneurs. *(2) Le niveau d'abstraction*, permet de construire des modèles de l'environnement (représentation du monde) à partir des données issues des capteurs du robot. Cette architecture propose deux types de représentation du monde : La première sous la forme d'un graphe et la seconde sous la forme d'un système multi-agent (SMA). *(3) Le niveau décisionnel* est composé de deux planificateurs, un réactif et l'autre cognitif. Le premier permet de planifier globalement la trajectoire à suivre dans les espaces dégagés (sans collision avec les obstacles). Il se repose sur la représentation du monde sous forme d'un graphe, pour calculer des bouts de trajectoires qui permettent d'atteindre l'objectif. Le second planificateur est dédié à la gestion des obstacles, il effectue des simulations sur la représentation SMA pour déterminer le meilleur enchaînement d'actions pour reconfigurer l'environnement (bouger les obstacles) et dégager un passage. Les composant de cette architecture sont illustrés dans la Figure 3.23.

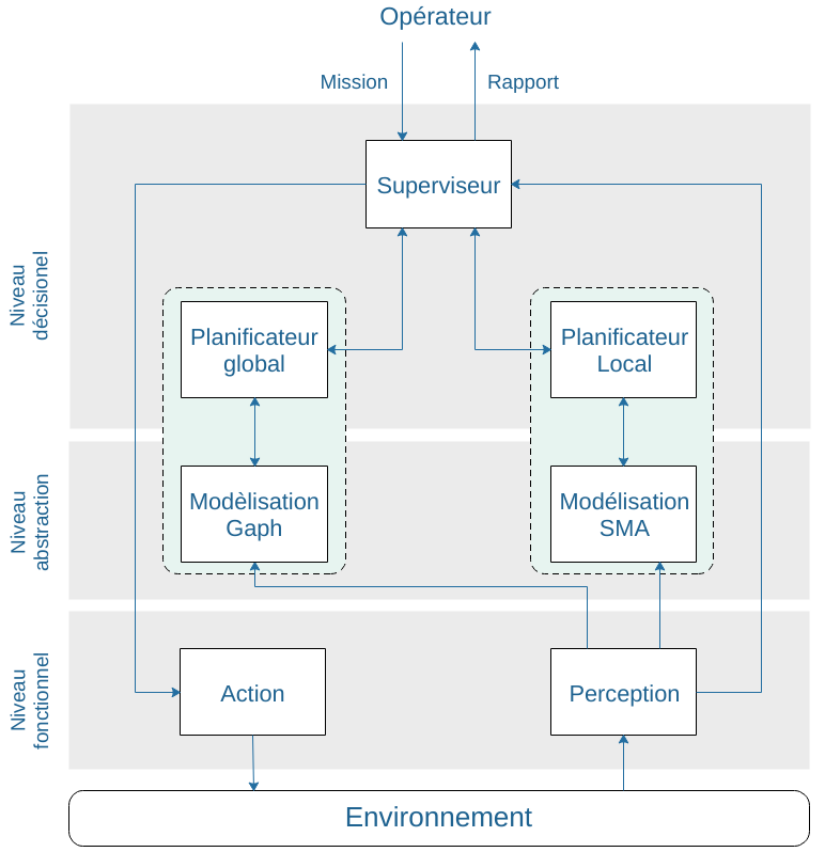


FIGURE 3.20 : Architecture VICA à trois niveaux.

- **Niveau fonctionnel :** c'est le siège des fonctions de bases du robot, il permet l'interface avec les composants physiques et les couches supérieures. Il assure la gestion des éléments sensori-moteurs. On trouve essentiellement deux modules :

(1) *Le module perception* offre des fonctionnalités de traitement de données reçues par les capteurs (traitement d'images, filtrage capteur de distance, données des capteurs de force, etc.). Chaque fonctionnalité est encapsulée dans un module qui fournit un ensemble de services accessibles par des requêtes.

(2) *Le module action* offre un ensemble de services permettant la gestion des actionneurs (moteurs), ces services sont accessibles par des requêtes. Les actions sont soumises à des boucles de contrôle, gérées par les niveaux supérieurs (superviseur).

- **Niveau d'abstraction :** Ce niveau utilise les données issues des capteurs (niveau fonctionnel) pour produire une représentation de l'environnement. Ce niveau est composé de deux modules. Ainsi, deux types de représentations de l'environnement sont réalisées :

  (1) *Représentation multi-agent de l'environnement :* Produit une représentation du monde sous forme d'un système multi-agent (SMA), où chaque objet (obstacle) de l'environnement est représenté par un agent. Cette modélisation à base d'agent permet de faire des simulations pour anticiper l'état futur du monde. Cette représentation est utilisée dans le cadre de la planification locale pour la gestion des obstacles.

  (2) *Représentation à base de graphe :* Produit une représentation du monde sous forme d'un graphe. Nous utilisons la division cellulaire pour construire ce graphe pondéré, chaque cellule est connectée aux cellules adjacentes, le coût d'un déplacement horizontal ou vertical est de 1 et le coût d'un déplacement diagonal est de $\sqrt{2}$. La taille d'une cellule est égale aux dimensions du robot. Ces deux modules reçoivent les données du niveau fonctionnel et mettent à jour les représentations continuellement.

- **Niveau décisionnel :** C'est le plus haut niveau de l'architecture hiérarchique, il intègre les capacités délibératives, il est composé principalement de trois modules :

  (1) *Le module planificateur globale :* Utilise la représentation graphe pour calculer et planifier un chemin optimal pour atteindre une position donnée.

  (2) *Le module planificateur local* utilise la représentation multi-agent pour déterminer un coût pour atteindre une position donnée. Ce planificateur utilise la simulation pour déterminer le mouvement et le coût de chaque action.

  (3) *Le module superviseur* La fonction principale de ce module est le pilotage et l'ordonnancement des actions.

Nous avons présenté jusqu'à présent les principaux modules qui composent l'architecture robotique proposée. Afin de mettre en évidence le fonctionnement de l'architecture, nous proposons un découpage par fonction (voir le diagramme d'activité Figure 3.22). Nous avons identifié deux niveaux de fonctionnement de cette architecture à savoir un niveau cognitif qui est la partie « *intelligente* » qui s'occupe de la planification et un niveau dite réactif, qui s'occupe de la perception et l'exécution des plans d'actions.

Nous appelons « niveau cognitif » les fonctionnalités nécessitant un processus calculatoire important qui peut être déterministe, nécessitant aussi un temps de délibération important et variable. La figure 3.21 montre le découpage fonctionnel, chaque bloc montre un une fonctionnalité. Des fonctionnalités peuvent être encapsulées dans d'autres. On peut dénombrer les fonctionnalités suivantes :

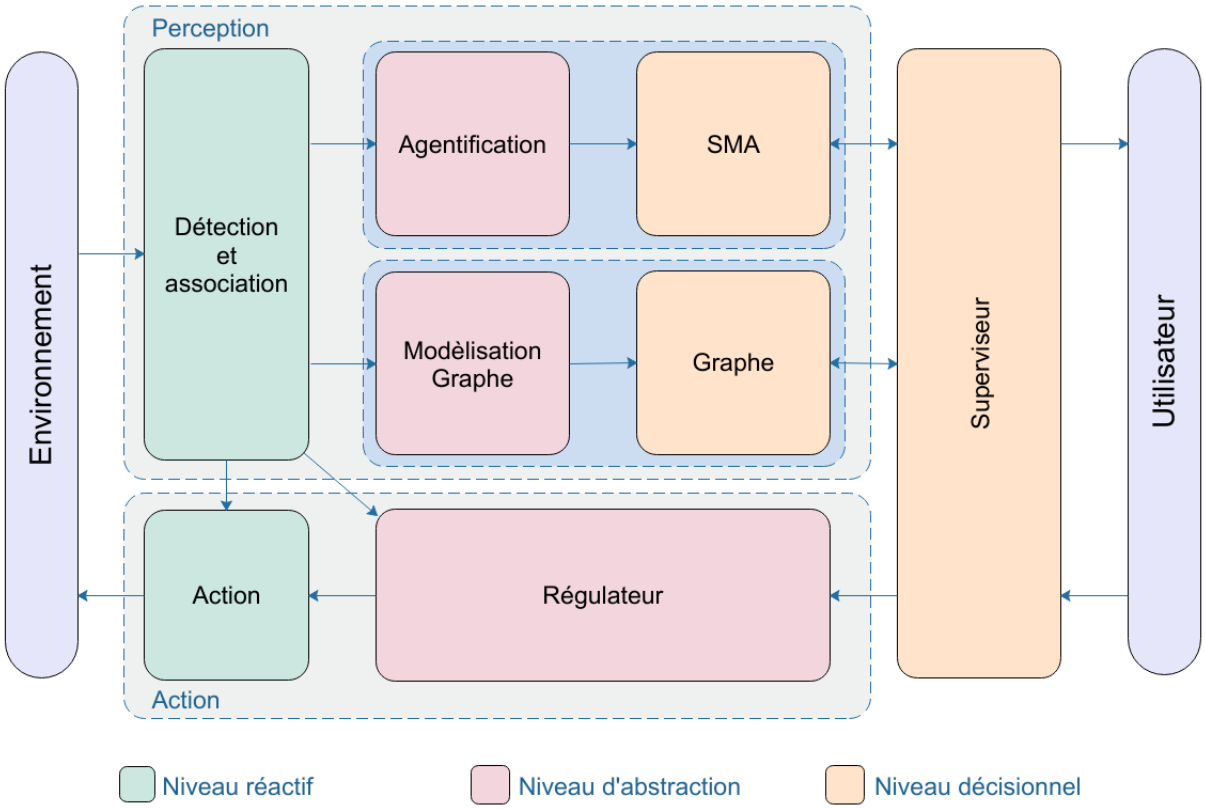


FIGURE 3.21 : Les principaux composants de l'architecture VICA : Le niveau cognitif regroupe le niveau d'abstraction et le niveau décisionnel.

- Interaction utilisateur
- Perception
- Déterminer les sous-objectifs
- Chois du planificateur
- La planification globale
  - Estimation du coût des trajectoires
  - Exécution mouvement
- La planification locale
  - Estimation mouvement par simulation
  - Exécution mouvement

3.5.1 Interaction utilisateur

L'utilisateur donne les instructions de position à rejoindre et obtient en fin de mission un rapport. Pour l'instant, l'utilisateur donne ses instructions au robot via le logiciel *RViz*, en

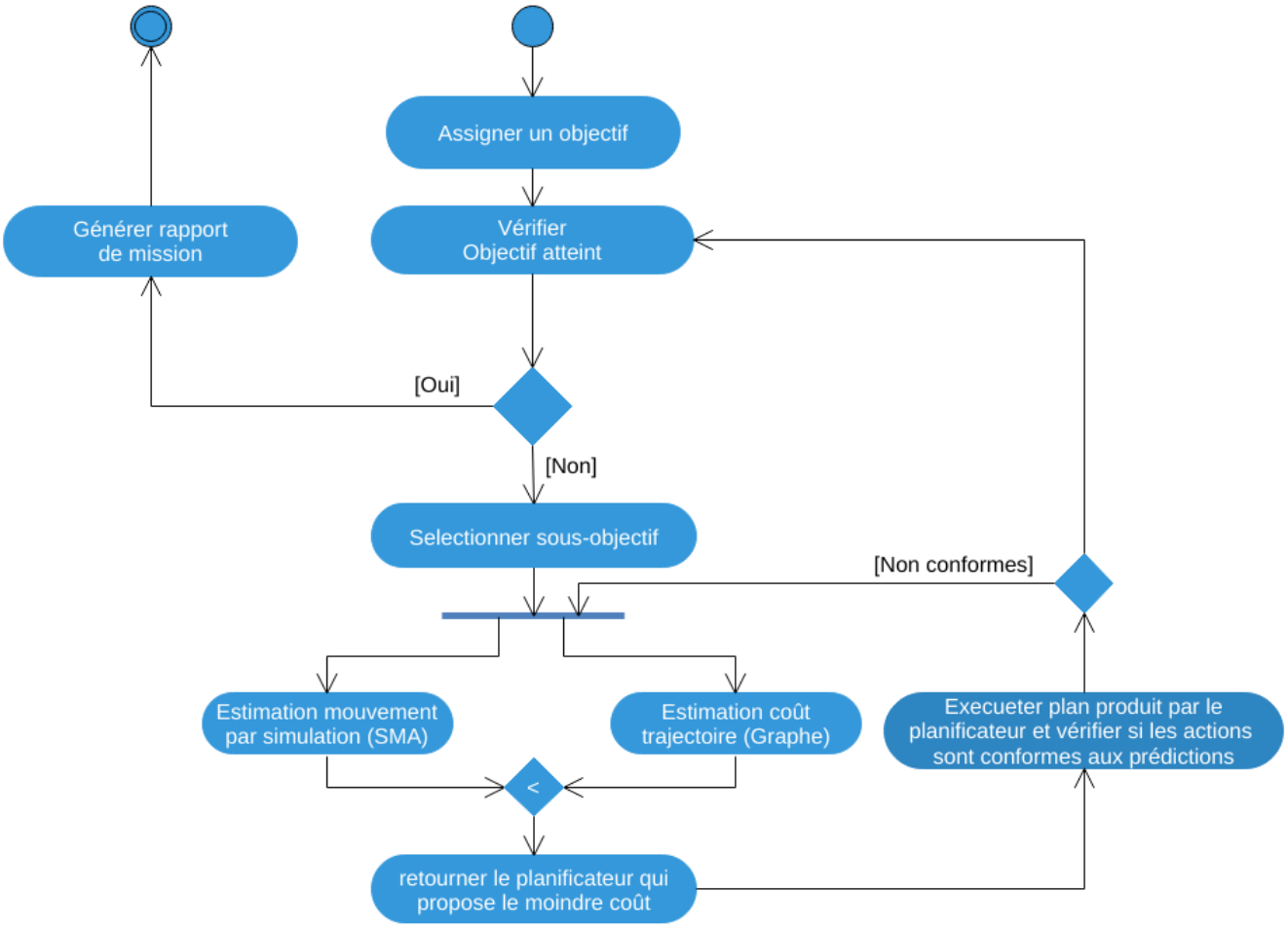


FIGURE 3.22 : Plan global de l'architecture VICA.

utilisant la fonction *2D Nav Goal* via le menu. Il est aussi possible d'envoyer message de type `geometry_msgs/PoseStamped` sur le topic `/move_base_simple/goal`

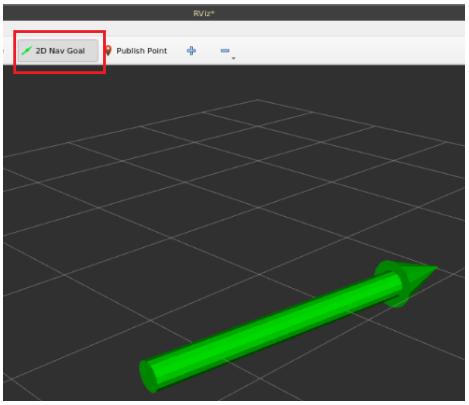


FIGURE 3.23 : Assigner à robot à but à atteindre via RViz.

Le rapport consiste en un ensemble de message que robot envoie tout au long de sa mission. Ces messages peuvent être consultés par l'utilisateur en lisant le *topic* suivant : `/vica/repport`.

3.5.2 Perception et représentation de l'information

La fonctionnalité de perception englobe l'acquisition de l'information et sa représentation. Le but principal de ce module est d'alimenter les niveaux supérieurs avec les nouvelles informations en provenance de l'environnement.

La « détection » est composée essentiellement d'algorithmes permettant de traiter les données des capteurs, pour chaque capteur un algorithme approprié est associé pour permettre d'extraire les informations nécessaires de la scène observée, par exemple le capteur RGB est associé avec un réseau de neurones de reconnaissance des objets, le capteur LIDAR est associé avec un algorithme de cartographie, etc.

Dans le système proposé on dispose de deux représentations, la première « Représentation par Graphe » , donne une représentation des espaces libres de l'environnement sous forme d'un graphe pondéré permettant ainsi le calcul de chemin, La Cartographie, correspond au processus permettant la détection et la création des nœuds du graphe ainsi que les pondérations des arêtes. La seconde « représentation SMA » consiste en une représentation sous forme d'un système multi-agent de l'environnement, où chaque obstacle est représenté sous forme d'un agent. La phase d'agentification correspond au processus qui permet la détection de nouveaux objets (non-détectés antérieurement) dans l'environnement sous forme d'agents.

Donc le module de détection sert à servir deux processus (voir Figure 3.24). Ces deux processus n'utilisent pas les mêmes capteurs. Le *processus 1* qui correspond au processus de la planification locale, utilise le capteur RGB et le Capteur de Profondeur. Le *processus 2* qui correspond au processus de la planification globale, utilise l'odométrie et le capteur Lidar. Plus concrètement le module détection illustré dans la Figure 3.24) implémente deux primitives[10] :

- `Perception.GetListObjects()`, utilise en entrée le capteur RGB et le capteur de profondeur, et en sortie, donne la liste des objets, leurs dimensions ainsi que leurs positions relatives au robot.
- `Perception.GetMap()`, utilise le capteur *LIDAR* pour cartographier les espaces libres.
- `Perception.GetValue(`*type*`)`, permet d'obtenir des données capteurs, par exemple le capteur de force (mesure la force de poussée du robot sur les obstacles), odometrie et le LIDAR.

L'objectif de ce module est de percevoir l'environnement à travers les différents capteurs du robot et analyser son contenu. Le robot que nous avons mis en œuvre est doté de multiples capteurs. Une caméra de type *Kinect / Intel Real Sens SR 300* composée d'une caméra RGB et d'une caméra de profondeur pour la détection, la reconnaissance et l'estimation de la distance du robot par rapport aux obstacles. Le robot embarque aussi d'autres capteurs de type LIDAR, odomètres, accéléromètre, etc. Qui ne participent pas à la détection des obstacles, mais fournissent des informations sur la position et la vélocité du robot.

[10]Ces primitives sont implémentées sous la forme de services ROS, le premier accessible via `/vica/perception/GetListObjects` et le second `/vica/perception/GetValue`

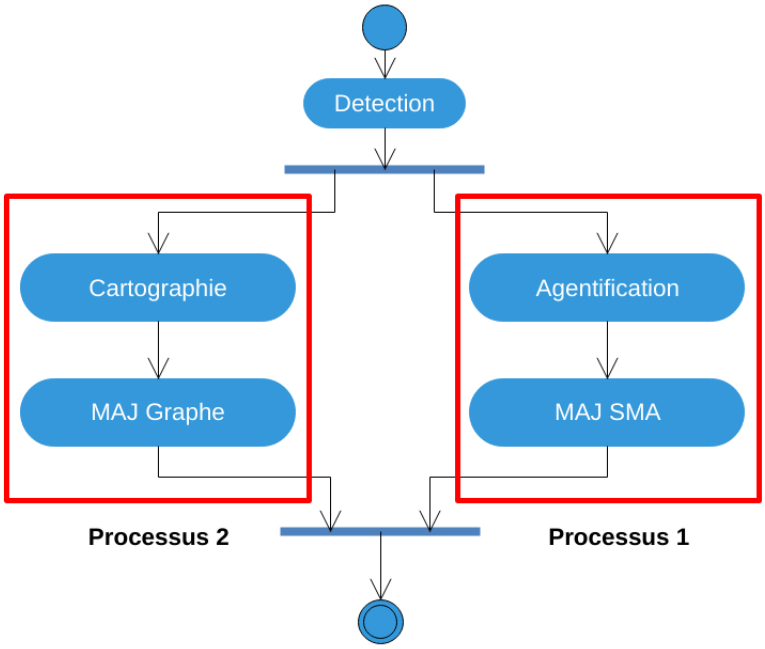


FIGURE 3.24 : Planification globale.

La primitive : `Perception.GetListObjects()`

L'objectif du module utilisant cette primitive est d'analyser la scène. Pour réaliser cette tâche, il a besoin de connaître la liste des différents objets qui entourent l'environnement immédiat du robot, leurs positions par rapport au robot, leur taille, etc. L'implémentation de cette primitive est organisée en deux parties comme illustré dans la Figure 3.25. La première partie, appelée ici *détection*, permet de détecter les différents objets via un capteurs *RGB*, et un *Capteur de profondeur infrarouge*. Le sous-module *Détection & Merge* analyse chaque objet afin de récolter le plus d'information selon le type de capteur. Le capteur *RGB* permet de déterminer le type d'objet avec un certain degré de confiance comme illustré dans la Figure 3.26. Le capteur de profondeurs permet quant à lui de déterminer la distance qui sépare les obstacles du robot ainsi que leurs dimensions.

La seconde partie de ce module (*Association*) permet d'ajouter des informations supplémentaires aux différents objets reconnus dans l'environnement. Les informations rajoutées sont issues de l'expérience précédemment acquise du robot lors de ses déplacements. Les informations rajoutées concernent essentiellement les éléments qui entrent en jeu pour déplacer les obstacles (la force nécessaire pour déplacer l'objet), d'autres informations concernant l'interaction avec les autres acteurs de l'environnement, etc.

Les objets reconnus dans l'environnement sont stockés dans une file de messages. Ces messages sont traités un à un pour essayer de trouver une correspondance dans la base de données afin d'associer les valeurs correspondantes aux attributs des objets détectés. Plus exactement, la base de données n'est rien d'autre qu'une table (c.f. Tableau 5.2) qui met en correspondance les noms et types (voir Figure 3.28) des objets détectés avec leurs coefficients de frottement[11].

[11]Le coefficient de frottement par défaut pour le tous les objets est initialisé à 0.4 cette valeur est modifiée lors des manipulations des obstacles par le robot, si un écart est constaté alors un biais de 0.02 est ajouté ou

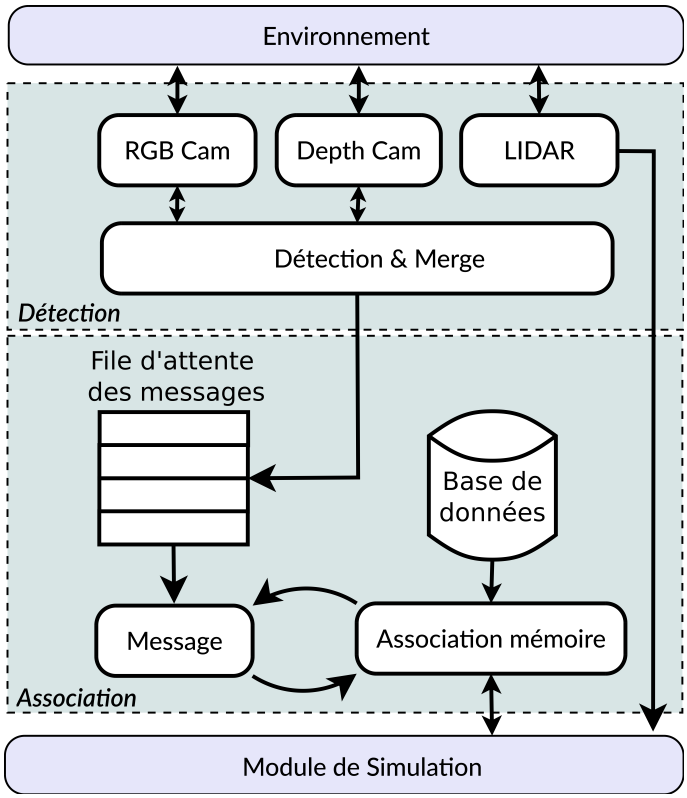


FIGURE 3.25 : Module détection et association.

La reconnaissance des objets est réalisée à l'aide de la librairie TensorFlow [Abadi et al., 2016], nous avons utilisé un modèle pré-entraîné (*ssd_mobilenet_v1_coco*) à l'aide du dataset *Microsoft COCO* [Lin et al., 2014]. Le dataset Microsoft COCO présente un avantage dans notre contexte, car il embarque un large ensemble de labels (80 labels, c.f. 5.2) qu'on trouve habituellement dans un environnement domiciliaire (chaises, tables, différents jouets, TV, etc.). La Figure 3.26 illustre la reconnaissance d'objets dans un environnement domiciliaire typique à l'aide du robot décrit au §3.1.1. La Figure 3.27 illustre *le module de détection* en utilisant le simulateur *Gazebo*.

Le modèle pré-entraîné utilisé, se base sur des réseaux de neurones convolutionnels (CNN), il utilise l'architecture *mobilenet*, associé avec la technique *SSD* (Single Shot Detector) permettant alors la reconnaissance de plusieurs types d'objets dans une seule image. Les *mobilenet* présente aussi l'avantage d'être rapides et efficace. Nous avons testé cinq différents modèles pré-entraînés à l'aide du dataset *Microsoft COCO*, nous avons constaté que les niveaux de performances sont sensiblement les mêmes, le choix de (*ssd_mobilenet_v1_coco*) est arbitraire.

Les dimensions des objets sont calculées grâce au nuage de points fourni par la caméra infrarouge. La détermination de la taille d'un objet est réalisée de la façon suivante : On superpose la zone reconnu sur l'image RGB sur le nuage de point généré par la caméra de profondeur comme illustré dans la Figure 3.29 pour extraire une zone plus petite de l'objet en question. Le nuage de point est donné sous forme d'une matrice avec des valeurs flottantes. Chaque valeur indique la distance qui éloigne ce point de la caméra, si un point n'est pas détecté

soustrait à la valeur précédente).

FIGURE 3.26 : Détection des objets : L'image RGB de gauche montre l'image tel qu'elle est reçu par le capteur RealSens RS300. L'image de droite montre la reconnaissance des objets encadrés en rouge avec le label correspondant et l'indice de confiance.

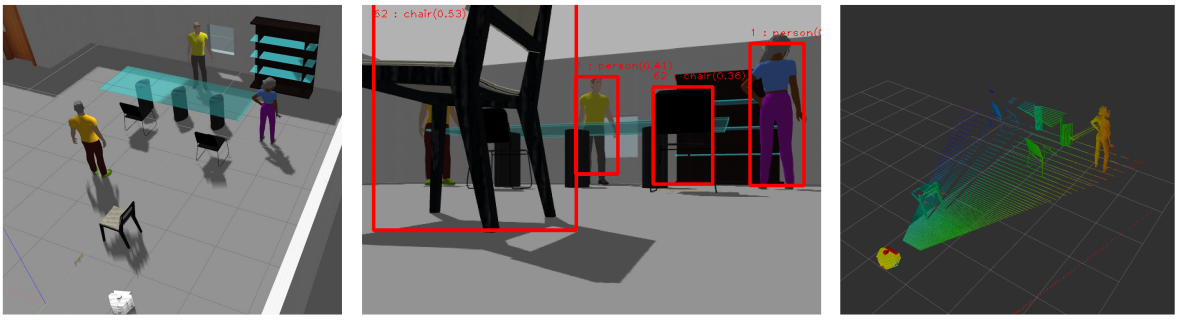


FIGURE 3.27 : L'image de gauche montre l'environnement virtuel **Gazebo** dans lequel le robot évolue. L'image du milieu montre la reconnaissance des obstacles à travers le capteur RGB. La dernière image à droite montre le rendu du capteur de profondeur dans **Rviz**.

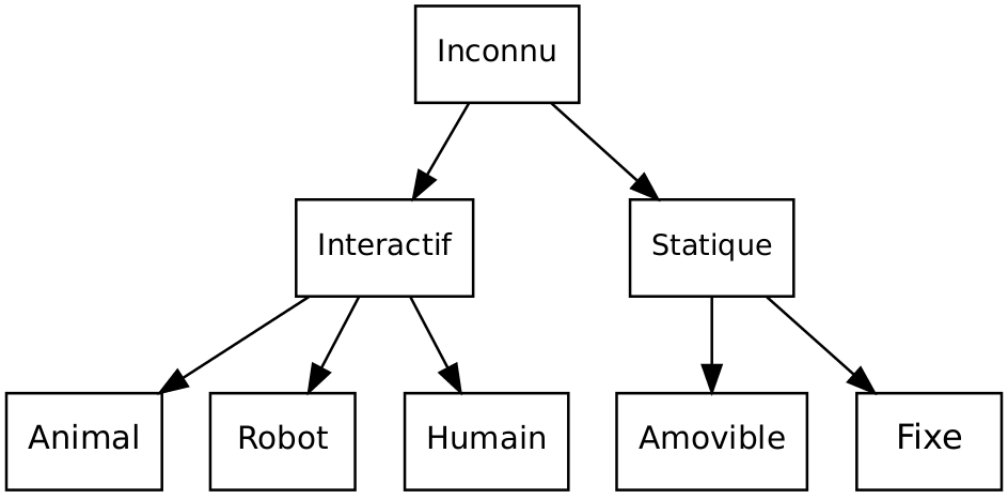


FIGURE 3.28 : Catégorisation des types d'obstacles.

par la caméra alors une valeur $infinie$ lui est associée.

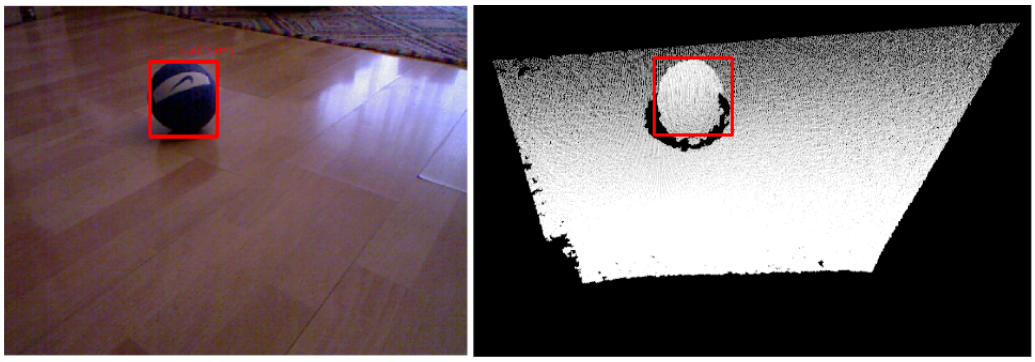


FIGURE 3.29 : À gauche l'image obtenue de la caméra RGB et le système qui reconnut un ballon de foot à 97% dans la zone en rouge. À droite, on superpose la zone reconnue (où se trouve le ballon).

Le calcul de la taille des objets se base donc sur cette matrice $3D$. On calcule 3 matrice $2D$ (M_x, M_y *et* M_z). Sur chaque face (matrice $2D$) on prend la valeur maximale dans cette direction. Le calcul se fait par rapport au point central, donc il faut faire le calcul à chaque fois dans les deux directions.

$$M_{x_{i,j}} = max_{k=0}^{n}(M_{k,i,j} - M_{0,i,j}) \ \cup \ M_{x_{i,j}} = max_{k=0}^{-}n(M_{k,i,j} - M_{0,i,j})$$

$$M_{y_{i,j}} = max_{k=0}^{n}(M_{i,k,j} - M_{i,0,j}) \ \cup \ M_{y_{i,j}} = max_{k=0}^{-}n(M_{i,k,j} - M_{i,0,j})$$

$$M_{z_{i,j}} = max_{k=0}^{n}(M_{i,j,k} - M_{i,j,0}) \ \cup \ M_{z_{i,j}} = max_{k=0}^{-}n(M_{i,j,k} - M_{i,j,0})$$

FIGURE 3.30 : Extraction d'une partie de la matrice de l'image principale. Ici l'image du ballon en l'axe Y.

Pour déterminer la dimension d'un objet, il suffit alors de déterminer la plus grande valeur sur chacun des axes. La figure 5.3 illustre un cas pratique, et donne les dimensions des objets dans le Tableau 5.1.

Ici, on illustre un cas concret et complet de la primitive `Perception.GetListObjects()`. Ce module fourni deux flux de message (*topic ROS*) des objets détectés avec leur taille, leur position par rapport au robot et leur type. L'exemple illustré dans la Figure 5.3 produit en sortie de l'analyse de la scène les données montrées dans le tableau 5.1 :

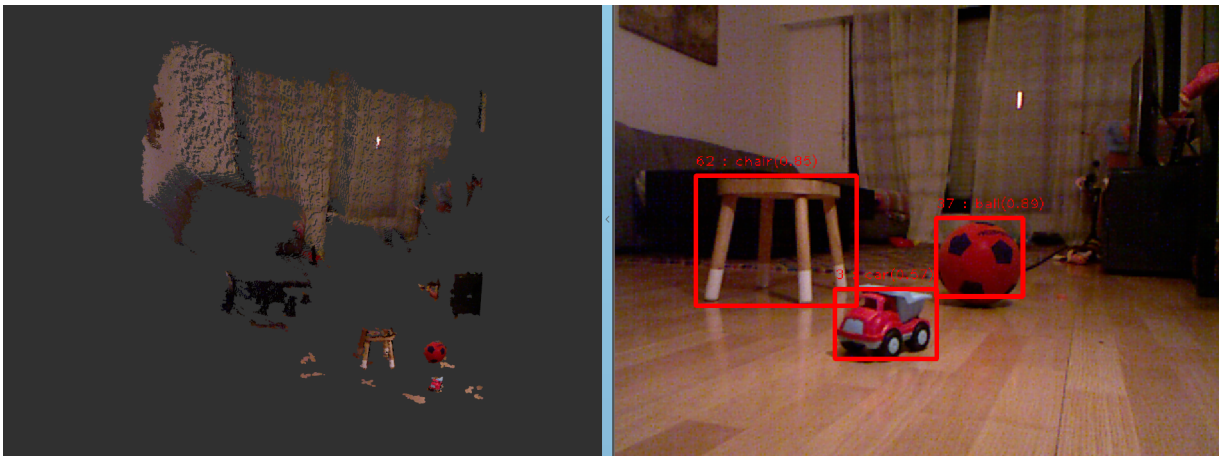


FIGURE 3.31 : Détection des objets à l'aide d'un capteur RGB (image de droite) et un capteur de profondeur (image à gauche). Les objets reconnus sont délimités par un rectangle rouge et dernier et transposés dans le nuage de données de profondeurs (image de gauche) afin d'extraire les dimensions de ces objets.

Id	Type	Prob. certitude	Cat.	Dist.	Orientation	Dim. X	Dim. Y	Dim. Z	Coef. frottement
62	Chair	0.82	Mouvable	1418.41	343	22.67	24.14	32.95	0.4
3	Car	0.63	Mouvable	87.72	356	122.78	52.29	77.10	0.4
37	Ball	0.9	Mouvable	1861.54	9	36.41	70.84	27.75	0.4

Tableau 3.1 : **Id** : identifiant de l'objet dans la base de données, **Type** : nom de l'objet, **cat.** : catégorie de l'objet (fixe, amovible ou interactif), **Dist.** : Distance de l'objet par rapport au capteur, **Orientation** : Orientation en degrés du centre de l'objet par rapport au capteur, **Dim. X** : Largeur en mm, **Dim. Y** : Profondeur en mm, **Dim. Z** : Largeur en mm.

La primitive : `Perception.GetMap()`

Cette primitive fournie au module du planificateur global un scanne en $2D$, sous forme d'un tableau à 1 dimension. Chaque valeur (nombre flottant) du tableau représente la distance qui sépare le point détecté du LIDAR après avoir éliminé le bruit en utilisant simplement le module *laser_filters*[12] disponible dans ROS. Le nombre de valeur représente la résolution du scanner (elle est paramétrable et dépend aussi du type du LIDAR) la première valeur du tableau représente la distance détectée à 0 degré et la dernière à 2π.

[12] *laser_filters* : https://wiki.ros.org/laser_filters#generic_laser_filter_node

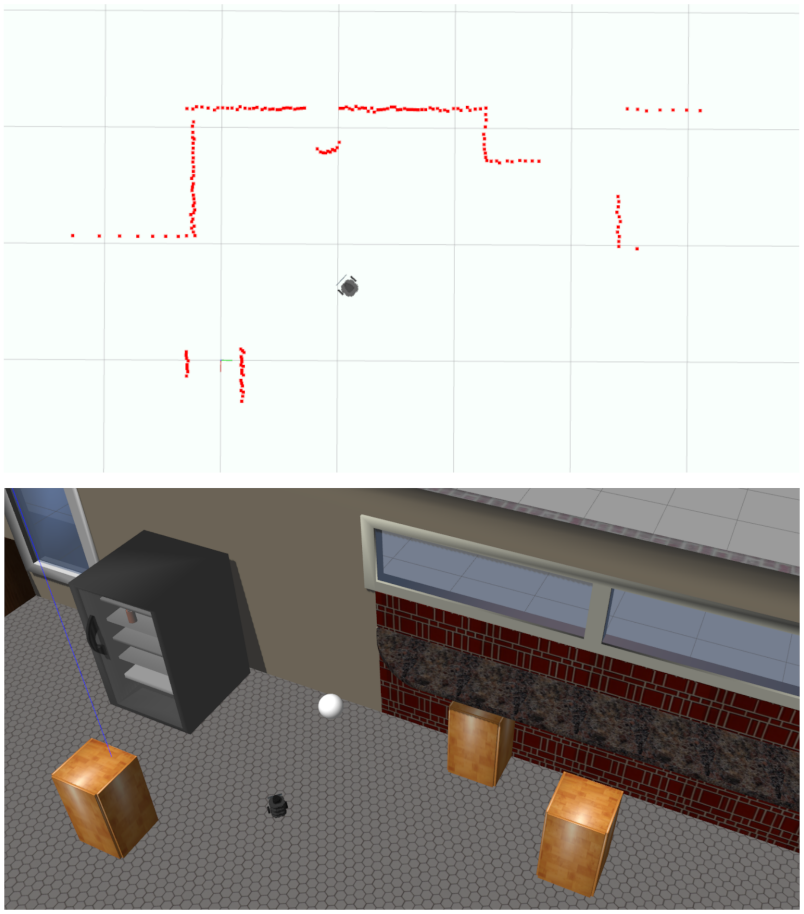


FIGURE 3.32 : En bas de l'image le robot Turtlebot 3 équipé d'un LIDAR dans un environnement domiciliaire simulé sous Gazebo. En haut, on voit, via RVIZ, le nuage de points généré par le LIDAR.

3.5.3 Déterminer les sous-objectifs

Pour déterminer les sous-objectifs à atteindre, nous avons utilisé dans un premier temps une technique basée sur les espaces libres. Cette technique s'est avérée inutilisable dans certaines configurations. Pour cela, nous avons donc expérimenté une seconde technique plus concluante basée sur un algorithme inspirée d'insectes (utilise le même algorithme que le planificateur global). Dans la suite de cette section, nous allons décrire les deux techniques expérimentées ainsi que leurs avantages et inconvénients.

Déterminer les sous-objectifs sur la base des espaces libres

L'approche que nous avons proposée repose fortement sur le but assigné au robot. Comme nous l'avons indiqué au début de ce chapitre, le robot reçoit comme instruction un cap à suivre et un objet à trouver[13]. La localisation du robot est relative à son but, donc il peut déterminer approximativement s'il s'éloigne ou se rapproche de son objectif. Donc le robot suppose que l'objet est le plus proche possible (deux fois plus loin que le rayon optimal dans lequel il peut

[13]On utilise le capteur RGB pour identifier le *but*.

reconnaître les objets[14]).

La Figure 4.15 illustre de cette idée. Le robot dispose d'une direction de son objectif, à partir de cette direction, il propose d'atteindre un nouveau point appelé *sous-objectif* (illustré par des points jaunes dans la Figure 4.15). Ce point est placé dans la direction exacte de l'objectif à une distance égale à deux fois le rayon du détecteur[15] (ce cas est illustré dans la figure (a)). Ce point est placé dans cette direction si est seulement si l'espace dans cette direction est libre[16]. Dans le cas où l'espace devant n'est pas libre alors on place le sous-objectif dans l'espace libre qui se rapproche le plus de la direction de l'objectif[17] (ce cas est illustré dans la figure (b)). Une fois le prochain sous objectif fixé, le robot fait appel aux planificateurs (global / local) pour atteindre ce sous-objectif[18].

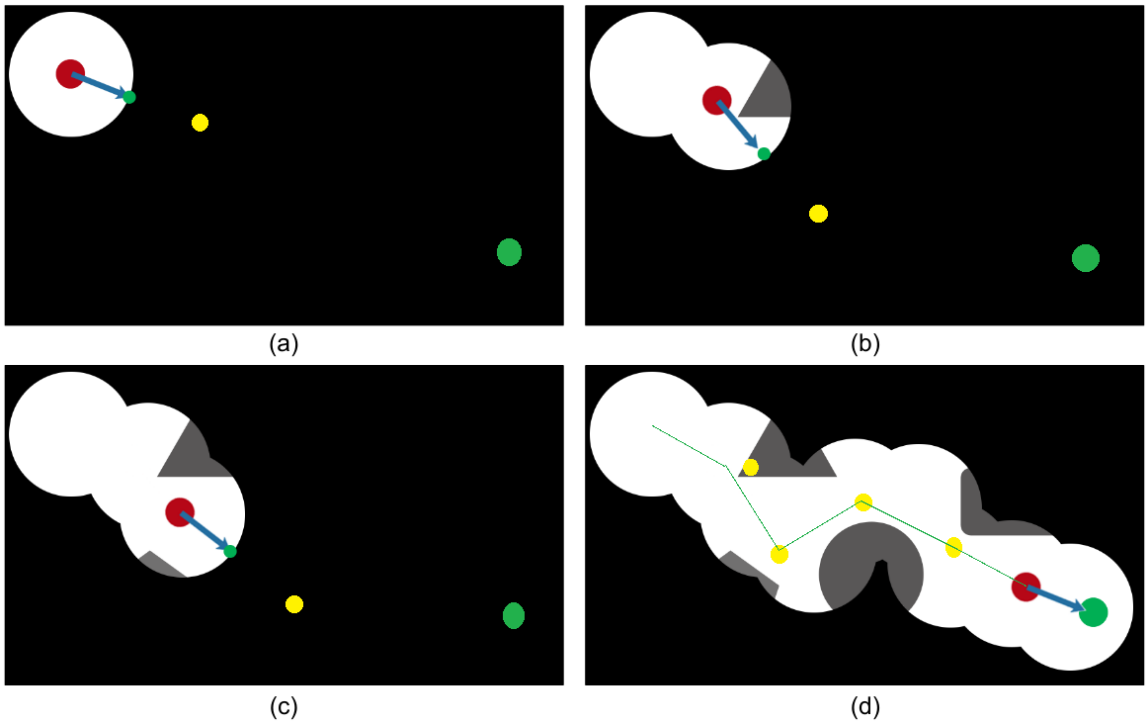


FIGURE 3.33 : Les différentes Figures montrent le robot en point rouge, et les sous-objectifs fixés avec un petit point jaune. (a) Montre la commande de l'utilisateur qui indique une direction. (b) et (c) montre l'avancement du robot l'adaptation de la trajectoire, avec de nouveaux sous-objectifs. (d) Le robot atteint l'objectif, matérialisé par un grand point vert et la trajectoire du robot montrée par les lignes vertes.

[14]Sur notre robot la distance optimale à laquelle il peut reconnaître correctement les objets et d'environ 150 cm, pour nos expérimentations nous avons choisi la distance de 300 cm.

[15]Attention : le rayon prit en compte est celui du capteur RGB pour reconnaître les objets et non pas celui du LIDAR.

[16]Cette fois on utilise le capteur LIDAR pour reconnaître les espaces libres.

[17]Cette contrainte évite au robot de placer les objectifs derrière les murs, c'est le cas le plus observé lors des expérimentations.

[18]Dans la Figure 4.15, on suppose que la portée du LIDAR est égale à la portée du capteur RGB permettant de reconnaître les objets. Pour réaliser cela sur le robot, il suffit de neutraliser (remplacer par la valeur $infini$) toutes les valeurs supérieures à r sur les données reçues du LIDAR.

Pour chercher les espaces libres, l'idée consiste à se baser sur les données du LIDAR pour cartographier l'environnement immédiat du robot, comme dans la plupart des implémentations des méthodes SLAM. L'environnement immédiat du robot consiste donc en un plan 2D comme illustré dans la Figure 3.34 (b).

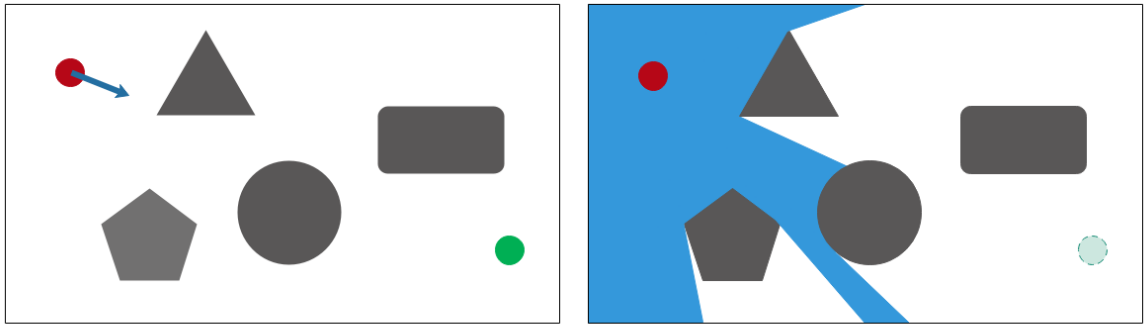


FIGURE 3.34 : Le robot est illustré avec un point rouge, et l'objectif à atteindre en vert. La figure (a) montre l'environnement complet dans lequel le robot évolue et les instructions de l'utilisateur avec la flèche bleu, (b) montre le balayage du LIDAR, en bleu, on trouve l'espace libre C_{free} visible à partir de la position actuelle.

À partir de la position initiale, on cherche les espaces libres, illustrés en violet dans la figure 3.35. La recherche des espaces libres se fait la manière suivante : En balaye l'ensemble des données renvoyées par le LIDAR [19], à chaque fois qu'un cisaillement dans les données, ou des valeurs inf sont repérées, on inscrit les angles $\theta 1$ et $\theta 2$ ainsi que les distances $d1$ et $d2$. Si un obstacle masque un autre obstacle alors $\theta 1$ et $\theta 2$ sont égaux. Si des valeurs inf sont détectées alors $\theta 1$ et $\theta 2$ sont différents. La figure 3.35 illustre les espaces libres avec des lignes en couleur violettes. L'Algorithme 3.5.3 montre le procédé utilisé pour extraire les passages possibles à partir des données de capteurs de type LIDAR.

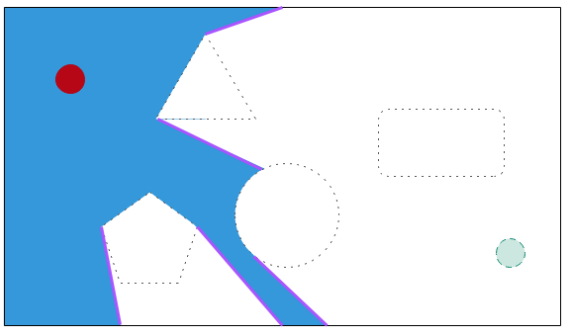


FIGURE 3.35 : Le robot, matérialisé ici en rouge utilise un capteur de type LIDAR, la zone bleue montre son rayonnement. On peut déduire que les zones marquées en violet sont des potentiels passages.

[19]Le LIDAR renvoie une liste de nombres flottants, chaque nombre correspond à une distance de l'obstacle, la position du nombre dans la liste indique l'angle à lequel cette distance est mesurée, si la distance est plus grande que la portée du LIDAR alors la valeur inf est inscrite.

L'Algorithme 3.5.3 montre le procédé utilisé pour extraire les passages possibles à partir des données de capteurs de type LIDAR. On commence par détecter les objets on éliminant les valeurs inf dans les données du LIDAR comme décrit dans l'Algorithme 3.5.3 cette algorithme retourne une liste d'angles où se trouvent les obstacles. Par la suite on inverse cette liste pour obtenir la liste des angles comme illustré dans l'Algorithme 3.5.3.

La position du robot et les chemins possibles sont stockés dans un graphe. Chaque nœud stock aussi les passages accessibles depuis ce point dans un arbre de segments[20]. La Figure 3.36 montre un exemple dans lequel un robot évolue en milieu congestionné. La Figure3.37 montre le graphe généré, dans le graphe chaque nœud stock l'heuristique qui indique la distance euclidienne de l'objectif.

La méthode initiale consiste à choisir comme prochain sous-objectif le point le plus proche de l'objectif final. Cette méthode s'est avérée inutilisable dans le cas où les planificateurs n'arrivent pas à s'effrayer un passage dans la direction suggérée. Le cas le plus souvent constaté est lorsqu'un mur se dresse devant l'objectif sans moyen de le contourner dans cette direction, alors le système suggère un point plus loin dans une direction opposée et au prochain coup, on revient au point initial qui pose problème. Pour résoudre ce problème une des méthode est de construire un graphe permettant ainsi de noter les positions déjà visitées et d'appliquer un algorithme de recherche de chemin. Pour cela, on procède comme suite : Les points déterminés par les coordonnées polaires (θ, d) (relatives à la position du robot) illustrés en bleu dans la figure 3.38 (a), sont accessibles directement par le robot (le robot peut les rejoindre en ligne droite avec une distance notée d), alors on suppose que à partir de ces points il existe un chemin en ligne droite (heuristique notée h) qui mène directement à la position finale (illustré dans la Figure 3.38 (a) avec des lignes en pointillés bleu.

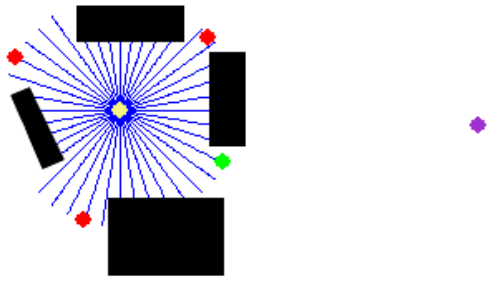


FIGURE 3.36 : La position actuelle du robot est montrée avec le point vert. Le point jaune montre l'ancienne position du robot à partir de laquelle il a trouvé les points de passage en rouge. Le point violet montre l'objectif à atteindre.

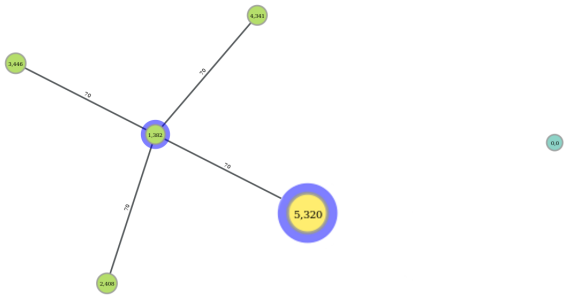


FIGURE 3.37 : Les nœuds du graphe montrent les positions accessibles par le robot, les nœuds entourés d'un cercle violet sont déjà visités, le nœud jaune et vert indiquent respectivement la position actuelle du robot et la position finale.

On choisi alors le point qui nous rapproche le plus possible de la position finale (Figure 3.38 (b)). Ce choix est orienté à l'aide de l'algorithme $A*$. Lors du déplacement du robot à chaque

[20]Un arbre de segments permet de stocker des intervalles ou des segments. Il permet de savoir rapidement ($O(\log_2(n))$) si un certain point appartient à un segment.

Algorithme 4 DectectObjets($data$)

Require: $data$
Ensure: $list\_angles$
  $id \leftarrow 1$
  $list\_angles \leftarrow \emptyset$
  $min\_elem \leftarrow min(data)$ ▷ Si y a pas d'obstacles alors on retour un seul élément.
  if $data[min\_elem] = \inf$ **then** ▷ Chaque objet est identifié avec un Id.
    $ADD\_ELEMENT(id, list\_angles, 0, 2\pi)$
    return $list\_angles$
  end if
  $beg\_elem \leftarrow 0$
  $end\_elem \leftarrow sizeof(data)$
  while $data[min\_elem] \neq \inf$ **do**
    $min\_elem \leftarrow min(data)$
    $left\_elem \leftarrow min\_elem - 1$
    $right\_elem \leftarrow min\_elem + 1$
    $id \leftarrow id + 1$
    while $True$ **do** ▷ Vérifier qu'on est toujours dans la page de données
      if $left\_elem < beg\_elem$ **then**
        break
      end if ▷ Détecter cisaillement dans les données, indique un espace libre ou qu'il y a potentiellement un objet qui cache partiellement un autre.
      if $\Delta(data[left\_elem], data[left\_elem + 1]) > \epsilon$ **then**
        break
      end if
      $left\_elem \leftarrow left\_elem - 1$
      ▷ Faire la même chose du côté droit
      if $right\_elem > end\_elem$ **then**
        break
      end if
      if $\Delta(data[right\_elem], data[right\_elem + 1]) > \epsilon$ **then**
        break
      end if
      $right\_elem \leftarrow right\_elem + 1$
      ▷ Ajouter à la liste l'angle de l'objet détecté
      $ADD\_ELEMENT(id, list\_angles, left\_elem, right\_elem)$
      ▷ Supprimer l'objet détecté
      for $i = left\_elem \rightarrow right\_elem$ **do**
        $data[i] = \inf$
      end for
    end while
  end while
  return $list\_angles$

découverte d'un nouveau point (Figure 3.39) le graphe est enrichi et un nouveau chemin est recalculé.

Algorithme 5 DectectPassages($data$)

Require: $data$
Ensure: $list$
  $list \leftarrow \emptyset$
  $i \leftarrow 0$
  for chaque $d \in data$ **do**
    $i \leftarrow i + 1$
    if $\Delta(d, data[i]) > \epsilon$ **then**
      ADD_ELEMENT($list, d, i$)
    end if
  end for
  return $list$

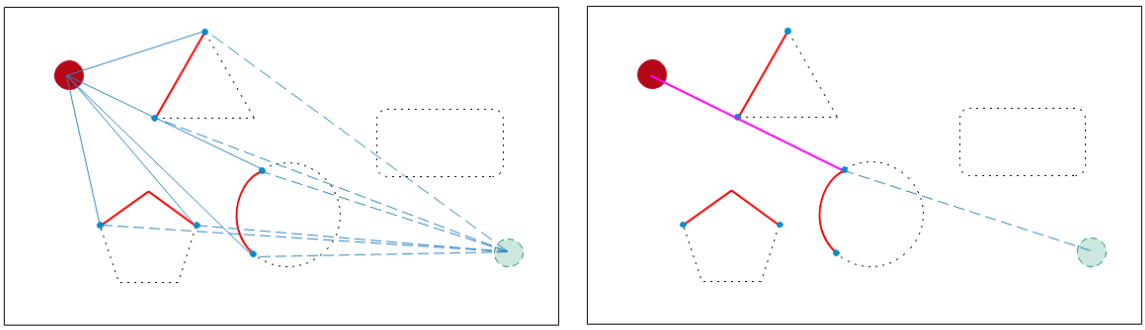


FIGURE 3.38 : description

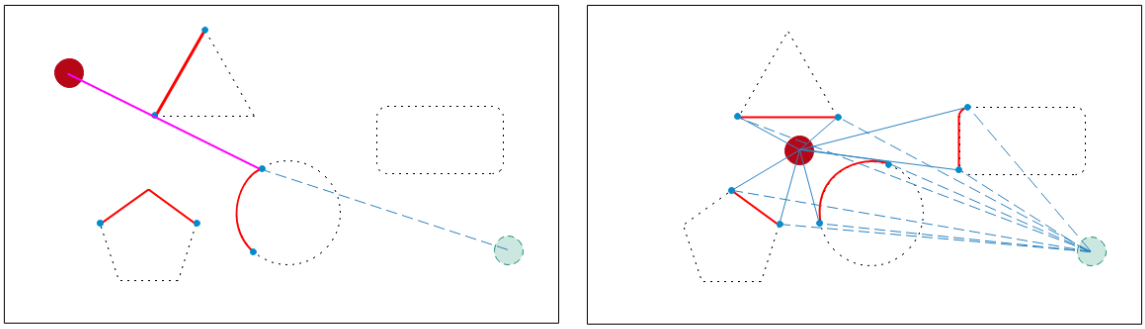


FIGURE 3.39 : (a) Le choix du chemin est effectué à l'aide de l'algorithme A^*. (b) Découverte de nouveaux points, alors le chemin est recalculé systématiquement.

Ici encore la difficulté réside lors de la construction du graphe, il est difficile de déterminer si des points déjà considérés comme accessible depuis un positon le sont encore lors d'un déplacement. Le graphe qui résulte est difficilement exploitable, car généralement, il admet uniquement un seul chemin, même si le robot visite une seconde fois la même zone. Pour conclure, cet algorithme n'est pas utilisé pour sélectionner un les sous-objectifs.

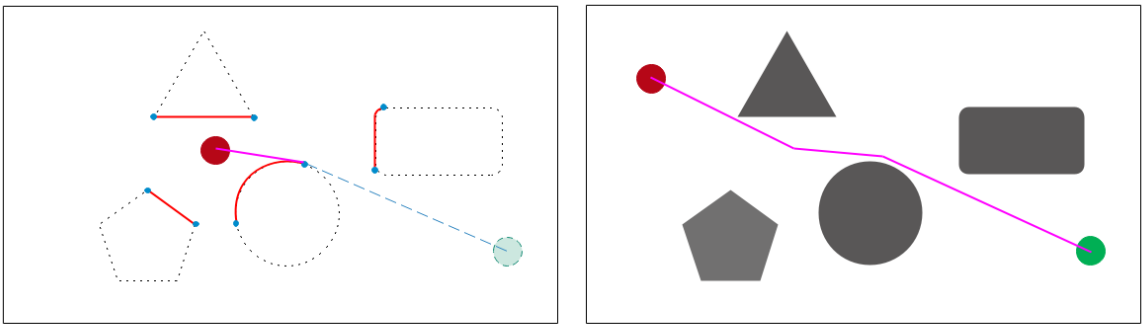


FIGURE 3.40 : (a) recalcule du nouveau chemin. (b) On constate que le chemin global est le chemin le plus court entre la position de début et la position finale.

Nous avons conçu un logiciel[21] il permet à l'utilisateur d'indiquer la position de début et la position de l'objectif à atteindre, permet aussi de dessiner à la main les obstacles et finalement, il permet de lancer d'autres algorithme de même type (RRT et Roadmap) sur la même carte pour comparer les algorithmes entre eux. La Figure 3.41 montre ce le parcours proposé par l'algorithme dans une situation favorable. La Figure 3.42 montre le graphe généré.

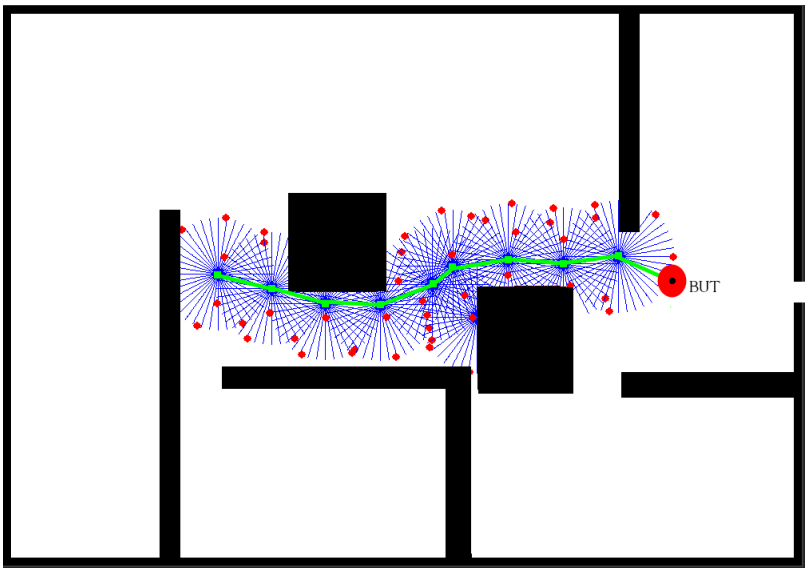


FIGURE 3.41 : Simulation dans une situation favorable. Les traits bleus représentent le scan du LIDAR. Les points rouges représentent les positions accessibles, dans le cas d'un grand espace libre, on ajoute un point à chaque $\pi/6$.

Déterminer les sous-objectifs sur la base de l'algorithme H*

Pour déterminer les objectifs d'une façon efficace, nous proposons d'utiliser la même technique basée sur l'utilisation du LIDAR. En revanche, au lieu de créer un graphe à basé sur les espaces libres, nous utilisons la technique de la division cellulaire pour la cartographie, puis le chemin

[21]logiciel disponible ici https://github.com/hdd-robot/nav_forward.git

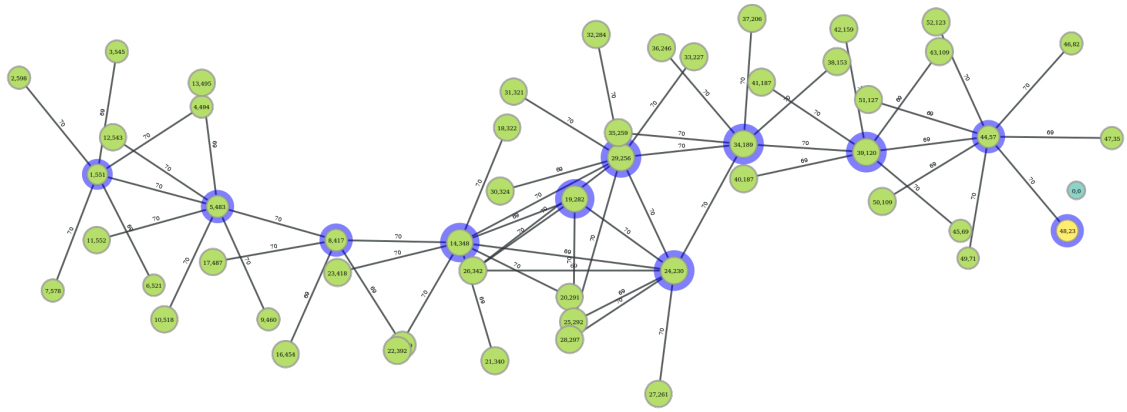


FIGURE 3.42 : Le graphe généré par l'algorithme.

est déterminé grâce à l'algorithme H^*. Cet Algorithme que nous avons développé (décrit au Chapitre 4) est à mi-chemin entre les *bug algorithms* (algorithmes inspirés d'insectes) et les algorithmes de recherche de chemin heuristique. Tout comme dans la première technique décrite ci-dessus, il permet d'aller tout droit, et contourner les obstacles avec l'avantage d'utiliser des heuristique pour optimiser les trajectoires et aussi garder en mémoire les espaces déjà visités.

La Figure 3.43 illustre le fonctionnement de la technique qui permet de déterminer le prochain sous-objectif. Soit le robot, dont la position est illustrée avec un point rouge au centre de la figure, a patrie de cette position, il cartographie son environnement immédiat grâce au LIDAR, ce qui lui permet d'étendre sa connaissance de l'environnement. La zone illustrée en jaune est la partie connue de l'environnement, le robot utilise l'algorithme H^* pour déterminer le point le plus proche connu permettant d'accéder au but. Le sous-objectif (illustré avec un gros point vert) est déterminé à l'intersection entre une ligne droite tracée du robot vers l'objectif et deux fois rayon de la capacité du capteur[22].

Une fois le sous-objectif déterminé, un des deux planificateurs conduit le robot vers cet objectif. Considérons la Figure 3.44, Le robot utilise un planificateur global qui le conduit à sa position actuelle à partir d'ici le planificateur global, ne peut plus avancer, alors ce planificateur calcul une nouvelle position lui permettant d'accéder à cette position et donne une estimation du coût de cette manœuvre (point vert hachuré en haut). Le planificateur local propose aussi un coup pour atteindre le sous-objectif. Il existe deux scénarios possibles : Le premier illustré dans la Figure 3.44, dans ce cas, le planificateur local ne peut pas dégager un passage (donc retourne un coût de déplacement *infini*), donc le planificateur global a un meilleur coût, donc il conduit le robot à la position calculée, et une nouvelle itération commence pour rejoindre le sous-objectif.

Dans Figure 3.45 on suppose que le planificateur local est capable de déplacer l'obstacle devant lui, donc il retourne un meilleur coût. Dans ce cas, il déplace l'obstacle pour accéder au sous-objectif. À partir de là, un nouvel sous-objectif est calculé avec le même processus. Le choix des planificateurs est orchestré par le module de supervision.

[22]Ici encore, on utilise le rayon d'utilisation du capteur RGB pour identifier les objets.

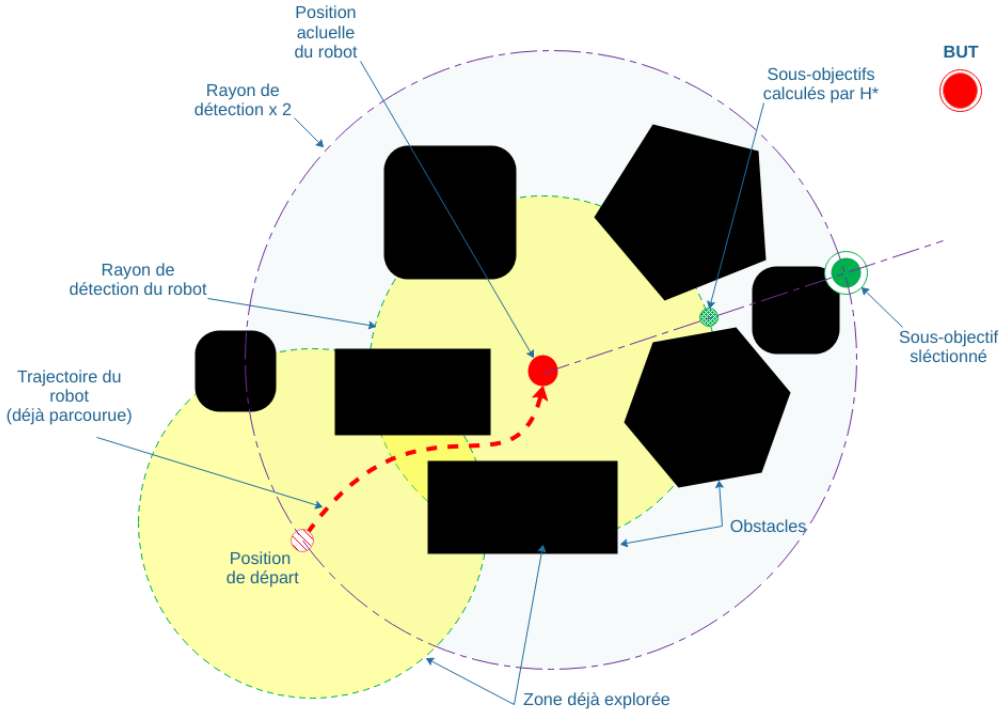


FIGURE 3.43 : Déterminer les sous-objectifs.

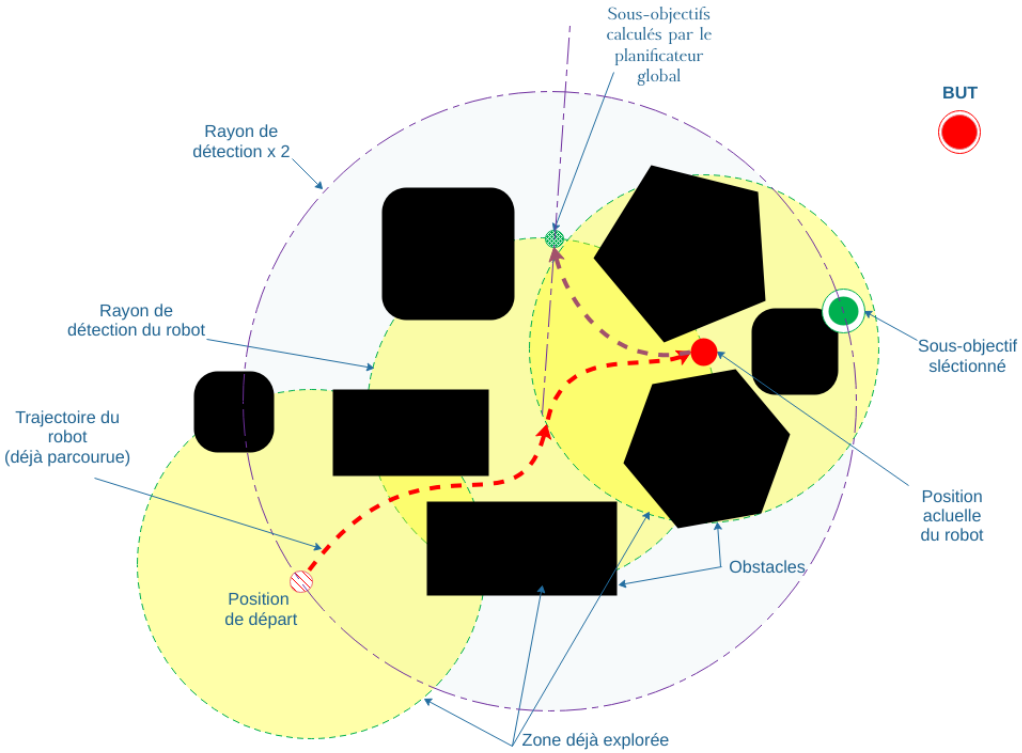


FIGURE 3.44 : Déterminer les sous-objectifs (suite).

La technique décrite ci-dessus permet la navigation dans un environnement complètement inconnu. Il est bien évidemment possible d'utiliser d'autres techniques pour un environne-

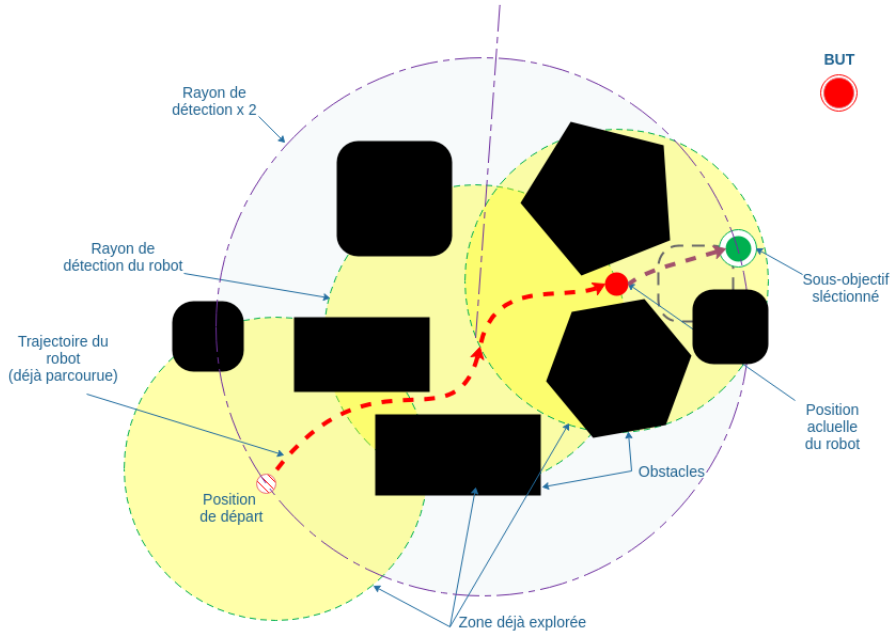


FIGURE 3.45 : Déterminer les sous-objectifs (suite).

ment partiellement connu, qui peuvent être plus ou moins efficaces, on peut proposer par exemple d'utiliser la méthode SLAM (Simultaneous Aocalization And Mapping) permettant au préalable de cartographier l'environnement afin d'avoir une carte sur laquelle on fixe les sous-objectifs. Mais cette méthode nécessite un coût supplémentaire, car il faut au préalable cartographier l'environnement, mais devient inutilisable, si certaine position sont inaccessibles ce qui aura pour conséquence une cartographie incomplète, de plus cette méthode est connue pour ne pas fonctionner dans des environnements avec des obstacles dynamiques et des obstacles de petite taille. Une autre solution consiste à utiliser l'algorithme D^* ou D^*Lite [Koenig and Likhachev, 2002][23]. À l'instar de notre algorithme, D^* donne un chemin global en se basant sur une connaissance partielle de l'environnement, mais cet algorithme doit connaître la position exacte du but, ce qui n'est pas le cas ici.

3.5.4 Planification

Le superviseur permet de calculer le meilleur compromis permettant d'atteindre l'objectif fixé par l'utilisateur (cf. chapitre 3). Son objectif principal, est la sélection d'un planificateur optimal pour une situation donnée et fixer les sous-objectifs. Les détails de fonctionnement des planificateurs global et local sont donnés respectivement dans les chapitres 4 et 5.

Planification global

Le planificateur global admet deux primitives :

- `PlanificateurGlobal.SimulationCout(`*pos*`))` permet de donner une estimation d'un coût pour atteindre la position relative (*pos*) avec l'algorithme H^*.

[23] D^*Lite est une variante de D^*.

- `PlanificateurGlobal.GetSteps(`*pos*`))` donne en retour un chemin pour atteindre la position *pos* (si possible). Le chemin est défini comme liste de positions (nœuds) à suivre dans un ordre donné.

Planification local

Le planificateur local admet lui aussi deux primitives :

- `PlanificateurLocal.SimulationCout(`*pos*`))` permet de donner une estimation d'un coût pour atteindre la position relative (*pos*) en utilisant des *simulations*.
- `PlanificateurLocal.GetSteps(`*pos*`))` donne en retour un chemin pour atteindre la position *pos*, si possible. Le chemin est défini comme liste de mouvements à effectuer dans un ordre. Pour chaque mouvement un poids est associé déterminant l'estimation que le système fait de la résistance des objets à pousser.

Pour implémenter le système de simulation, nous utilisons un AMS, le robot observe l'environnement et représente chaque entité de l'environnement sous forme d'un agent. Étant donné que nous avons pris comme référence un environnement domiciliaire, pour des raisons de simplicité, nous avons répertorié dans une base de données un nombre d'obstacles limité (au total 80 objets qu'on trouve généralement dans un tel environnement, c.f. Tableau 5.2) et nous avons manuellement classé ces objets en trois catégories (fixes, amovibles, interactifs et non-interactifs)[24]. Un objet fixe, ne peut être bougé et occupe une position fixe dans l'environnement (e.g. table, cabinet, etc.). Un objet amovible peut être déplacé par le robot en le poussant (e.g. chaise, ballon, etc.). Un objet interactif est un objet que le robot peut déplacer en lui envoyant un message (e.g. humain, robot aspirateur, etc.). Les lois de l'environnement décrivent comment les agents (les obstacles représentés) ce comportement face aux actions du robot. Un objet fixe, ne réagit pas aux actions du robot et continue a occuper la même position. Un objet amovible se déplace dans le sens opposé en accord avec le coefficient de frottement. Un objet interactif décrit la façon dans l'objet va se déplacer en accord avec le message reçu.

Au final, un tel SMA peut être considéré comme un moteur physique (comme ceux implémentés dans les jeux vidéo, des simulateur de la physique, etc.). Pour expliquer plus globalement le rôle de SMA, nous allons donner un exemple d'un robot qui observe une scène composée d'un ballon, d'une chaise et d'un robot aspirateur. Dans ce cas, la représentation de ces trois éléments va engendrer trois agents dans le SMA avec leurs positions et lois respectives. Le système fera des simulations en bougons les obstacles (en simulation) jusqu'à trouver le meilleur coût lui permettant d'accéder à l'objectif avec un minimum d'actions. Dans ce cas, ce planificateur retourne le plan d'actions à réaliser dans l'environnement réel. Un processus d'asservissement et intégré à ce système, il consiste à vérifier après chaque action réalisée dans le monde réel si elle correspond aux résultats de la simulation.

[24]La base de données des objets décrit le nom de l'objet (pour permettre d'identifier l'objet reconnu dans l'environnement à travers le module de perception décrit précédemment, leurs statuts (fixe, amovible, interactifs ou non-interactifs), un coefficient de frottement pour les objets amovibles, et finalement le type et la forme de message pour les objets dynamiques.

3.5.5 Choix du planificateur

Le choix du planificateur est assuré par le module de supervision, la primitive suivante lui permet de comparer le résultat de chaque planificateur.

- `Superviseur.ChoixPlanificateur(`*objectif*`)` en fonction de son objectif (généralement une position à atteindre) permet de déterminer quelle approche utiliser, avancement en évitement d'obstacle ou gestion des obstacles. Le fonctionnent est décrit dans l'Algorithme 6.

Algorithme 6 Superviseur.ChoixPlanificateur

Require: (θ, λ) avec θ = Direction et λ Objet à trouver
Ensure: Rapport /* positions à parcourir */
  $poids \leftarrow \infty$
  $rapport \leftarrow \emptyset$
  while $poids > 0$ **do**
    $pos \leftarrow CalculerProchainePosition(\theta, \lambda)$
    $p1 \leftarrow PlanificateurGlobal.SimulationCoût(pos)$
    $p2 \leftarrow PlanificateurLocal.SimulationCoût(pos)$
    if $p1 < p2$ **then**
      $tmp \leftarrow PlanificateurGlobal.Simulation(pos)$
      $poids = p1$
    else
      $tmpRapport \leftarrow PlanificateurLocal.Simulation(pos)$
      $poids = p2$
    end if
    $rapport \leftarrow rapport + pos$
  end while

3.5.6 Exécution du mouvement

Le module d'exécution de mouvement admet deux primitives :

- `Action.Goto(`θ`,`ρ`,plan)`, permet de rejoindre une position relative (ici avec des coordonnées polaires)en suivant le plan d'action donné en paramètre. Lors de l'avancement, les données des capteurs sont constamment retourné[25], permettant ainsi l'asservissement. Cette primitive utilise décompose le plan d'action en une soute de mouvement, puis chaque mouvement est réalisé grâce à la primitive `Action.Forward(`θ`,`d`,`*threshold*`)`.

- `Action.Forward(`θ`,`d`,`*threshold*`)`, permet de faire avancer le robot dans la direction θ pour une distance d, en utilisant une boucle de contrôle qui prend en entrée les données issues d'un capteur de force (c.f. Algorithme 7). Le paramètre *threshold* indique le seuil pour lequel l'action est annulée. Plus concrètement ce service permet au robot de pousser un obstacle d'une distance d dans une direction θ. Ce service utilise les données issues d'un capteur de force pour contrôler son mouvement. Pour pousser un objet en toute

[25] On utilise ici ROS Action dans l'implantation https://wiki.ros.org/actionlib

sécurité, un seuil (*threshold*) est utilisé pour permettre d'abandonner l'action dans le cas où il est dépassé. Pour le déplacement dans les espaces libres *threshold* est positionné à sa valeur minimum.

Algorithme 7 Action.Forward

Require: (θ,d,*threshold*)
Ensure: Rapport
  $final\_pos \leftarrow calculer\_position\_relative(\theta, d)$ /* position finale */
  $r\_pos \leftarrow get\_robot\_pos()$ /* position actuelle du robot*/
  $rapport \leftarrow \emptyset$
  while $r_pos \neq final\_pos$ **do**
    $r_pos \leftarrow get\_robot\_pos()$ /* position actuelle du robot*/
    $rapport \leftarrow \emptyset$
    $effort \leftarrow Action.Goto(\theta, \lambda)$
    if $effort < threshold$ **then**
      $rapport \leftarrow rapport + pos$
    else
      return $rapport \leftarrow seuildépassé$
    end if
    return *rapport*
  end while

3.6 Expérimentations

L'architecture VICA permet au robot de produire un plan et de l'adapter, si nécessaire, au cours de son exécution, pour atteindre un objectif. Le plan peut impliquer de : (1) se déplacer dans les espaces libres, (2) déplacer des obstacles par poussée, et (3) de demander aux obstacles interactifs de céder le passage. Une première version de VICA a été implémentée sur un robot virtuel sous *gazebo*. Il a été testé dans un environnement de bureau encombré comprenant différents types obstacles et différents scénarios.

Pour comparé notre architecture avec d'autres planificateurs, nous avons élaboré un environnement de test sous *Gazebo* (voir Figure 3.46). On commence par générer aléatoirement dans un environnement[26] clos de $400cm \times 400cm$ un certain nombre d'obstacle de petite taille (entre $5cm^3$ et $20cm^3$ uniquement) (on commence par 5 obstacles puis en augment le nombre d'obstacle progressivement après chaque simulation : 10 ,20 jusqu'à atteindre 70 obstacles). Le robot démarre toujours à la même position et l'objet à trouvé est le jouet (voiture) qui se trouve de l'autre côté. Pour chaque configuration de l'environnement on lance trois planificateurs un planificateur basé sur D^* *Lite*[27] et un autre planificateur basé sur RRT disponible ici[28].

[26]Pour cela nous avons élaboré un *plugin Gazebo* permettant de déposé des objets aléatoirement.

[27]Nous n'avons pas développé ce planificateur, nous utilisons un planificateur disponible https://github.com/palmieri/srl_dstar_lite. Il s'agit un package ROS basé sur *ROS move$_b$ase*.

[28]Là aussi, nous n'avons pas développé ce planificateur, nous utilisons un planificateur disponible ici https://github.com/srl-freiburg/srl_global_planner

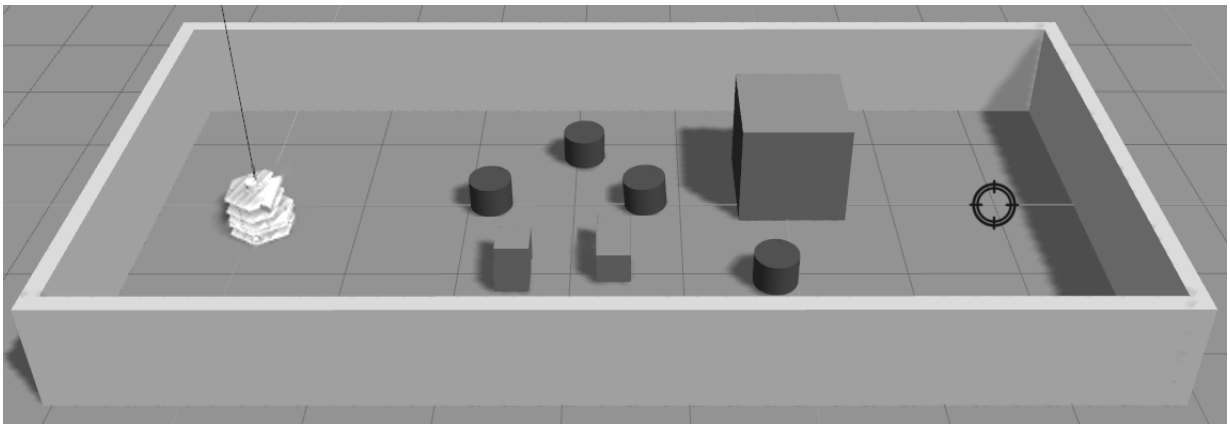


FIGURE 3.46 : Déterminer les sous-objectifs (suite).

Le tableau 3.6 compare notre modèle avec deux autres méthodes de planification *RRT* et D^* *Lite* bien connues. Notre planificateur est capable de déplacer des obstacles, le nombre de mouvements est indiqué dans la colonne (Mouvements). Le temps d'exécution du mouvement est indiqué en secondes dans la colonne (Time). Les colonnes (distances) indiquent la longueur parcourue en centimètres. La distance entre la position de départ et le but est fixe (350 cm) pour toutes les expériences. Notre méthode est capable de faire mieux que D^* *Lite*, car elle est capable de déplacer des obstacles et de trouver un meilleur chemin au lieu d'éviter les obstacles. Les expériences présentées ici n'impliquent que deux types d'obstacles (fixes et mobiles), les obstacles interactifs et dynamiques (robots, humains, etc.) ne sont pas testés.

	VICA			RRT	D* Lite
Nb Obs.	Nbr Mouv.	Temp. (sec)	Dist. Parc. (cm)	Dist.	Dist.
5	0	112	370	418	365
10	2	118	388	436	387
20	4	152	385	444	380
30	7	142	358	465	485
40	9	145	390	487	498
50	10	190	378	395	395
60	12	180	397	409	404
70	13	145	395	415	411

Tableau 3.2 : Résultats de simulations.

Nous avons aussi élaboré des simulations dans un environnement réel (voir Figure 3.47). Là aussi, nous n'avons pas pu tester l'interaction avec les objets interactifs, en raison de la petite taille du robot (la caméra du robot est proche du sole).

3.7 Implémentation

La Figure 3.21 montrée précédemment (vue globale de l'architecture VICA), chaque module (Détection et association, action, Agentification, SMA, etc.) est implémenté sous forme d'un

FIGURE 3.47 : Simulation avec un robot réel

metapackage ROS[29]. Donc une fonctionnalité décrite dans cette figure peut se traduire à un par plusieurs packages. Les détails de l'implémentation peuvent être trouvés sur https://github.com/hdd-robot/vica.git. Cette architecture est majoritairement dévelopée sous ROS.

ROS (Robot Operating System) [Quigley et al., 2009], il ne s'agit pas réellement d'un système d'exploitation mais plutôt d'un framework flexible pour l'écriture de programmes pour les robots. Il propose un cadre de travail pour que chaque tâche soit programmée sous forme d'un processus Unix en utilisant le langage *C++* ou *Python*, ROS offre un système de communication inter-processus évolué et simple d'utilisation, mais à la base il utilise les ressources du système d'exploitation pour réaliser ces tâches (communication ICP, processus Unix, couche réseau, etc.). Donc ROS s'agit plus d'un méta-système d'exploitation. L'avantage de l'utilisation de ROS se trouve dans la grande collection de bibliothèques spécialisées en robotique qui simplifient la tâche du programmeur. En plus de cela, il offres aussi un ensemble d'outils de visualisation tel que *R-VIZ*[Kam et al., 2015] et *rqt*. Il offre aussi une excellent compatibilité avec des environnement de simulation tel que *Gazebo* [Koenig and Howard, 2004], *V-Rep* [Rohmer et al., 2013] ou-bien *MoveIt*. Il existe, à l'heure actuelle, environ une centaine de plateformes robotique de recherche, d'éducation et même en industrie qui supporte nativement ROS, on peut citer par exemple *PR2* [Cousins, 2010], *NAO*, *Turtlebot* [Guizzo and Ackerman, 2017], etc.

[29]package virtuel qui ne contient pas de code source, il fait simplement référence à un ou plusieurs packages. Ils ont pour objectif de regrouper les packages qui assurent une fonctionnalité de haut niveau. Un package peut participer à la composition de plusieurs métapackages.

La communication entre les modules se fait essentiellement par messages, nous utilisons les systèmes de communication fourni par ROS à savoir : *ROS Topic*[30], *ROS Service*[31] et *ROS Action*[32]. Pour chaque module nous avons créé des messages personnalisés.

Il existe de nombreuses autres alternatives à ROS, on trouve par exemple MRPT (Mobile Robot Programming Toolkit) qui est une suite de bibliothèques C++ pour la robotique, MRDS (Microsoft Robotics Developer Studio) qui permet d'offre une alternative pour les développeurs MS Windows car la plupart des plateformes robotiques utilisent exclusivement les systèmes de type Unix. CARMEN (Carnegie Mellon Robot Navigation Toolkit) une collection de bibliothèques spécialisées dans la navigation pour robots. Parmi ces plateformes, ROS semble être le système le mieux documenté, car il bénéficie d'une communauté très active, actuellement une nouvelle version, *ROS2* est en train d'être réalisée pour combler certaines lacunes, comme par exemple le temps réel, tolérance aux fautes, etc.

Notre choix s'est porté sur l'utilisation de ROS en raison de la disponibilité d'un très grand nombre de bibliothèques et l'abondance de documentation. Les simulations réalisées dans le cadre de notre travail utilisent la version *ROS Kinetic*[33] installée sur une distribution *Ubuntu 16.04*.

La simulation via des outils tels que **Gazbo** permet de valider les modèles avant de les tester en grandeur nature. La simulation permet aussi de réellement simplifier le travail, car elle offre la possibilité de réduire les perturbations engendrées par un environnement réel. Malheureusement, dans le cas de la reconnaissance d'objets à travers les capteurs RGB en utilisant les graphes de réseaux de neurones pré-entraînés ne fonctionnent pas aussi bien dans un simulateur, utilisation d'un environnement réel offre de meilleurs performances. Les travaux concernant cette problématique ne sont pas très nombreux, d'autres articles tels que [Borrego et al., 2018] suggères qu'il est plus intéressant de créer une base d'apprentissage (*dataset*) dans un environnement virtuelle puis de l'appliquer à un environnement réel. Malheureusement, ce type de graphe pré-entraîne n'existe pas à l'heure actuelle, la création d'un tel *dataset* sort de notre domaine de recherche.

Pour l'implémentation du système multi-agent (SMA) nous avons utilisé le Un framework *SMA situés* (décrit au Chapitre 5) doté d'un environnement configurable et dynamique, que nous avons développé en C++ pour des raisons de compatibilité avec ROS. Conforme au standard *FIPA* et doté d'une interface graphique (minimale) permettant la visualisation de la position des agents dans l'environnement. *gAgent*[34] est un projet toujours en cours de développement, il a été développé expressément pour être utilisé dans VICA.

Les simulations que nous avons présentées sont réalisées grâce à l'outil de simulation Gazebo. Un simulateur d'environnement 3D, cinématique, dynamique et multi-robot et intègre un moteur physique ainsi, il permet de simuler les interactions entres les objets (collisions, chute,

[30] Communication par messages dans un canal à sens unique

[31] Envoie un message et attends une réponse.

[32] Envoie un message et reçoit plusieurs message concernant le déroulement de l'action jusqu'à la fin de l'action, avec la possibilité de réagir en cours de l'action.

[33] https://wiki.ros.org/kinetic

[34] https://github.com/hdd-robot/gAgent

etc.). *Gazebo* intègre une importante bibliothèque d'objets pré-configurés qu'on peut disposer dans l'environnement, il intègre aussi une importante collection de robots pour effectuer des simulations et permet d'intégrer de nouveaux robots en fournissant une description en fichier URDF. *Gazebo* compatible avec ROS ce qui justifie en partie l'utilisation de cet outil dans ce travail. Le principal avantage d'utiliser un tel simulateur au lieu d'un robot réel, c'est de permettre de comparer nos résultats avec d'autres méthodes existantes dans les mêmes conditions, ainsi avoir un comparatif sans biais.

En plus des simulations, nous avons souhaité proposer une application en conditions réelles. Pour nos expérimentations en environnement réel, nous avons commencé par développer notre propre plateforme robotique (voir Figure 3.4) au début de notre travail de recherche. Par la suite, nous avons passé à une plateforme robotique commerciale *Turtlebot3 Burger* (vioir Figure 3.5), que nous avons adapté pour les besoins de ce travail. Cette plateforme a déjà été utilisée dans de nombreux travaux de recherche.

Développer notre propre plateforme robotique n'a pas un réel intérêt scientifique, mais nous a permis d'apprendre sur l'interaction du robot avec des objets amovibles[35] et de se forger notre propre expérience. Mais le fait de passer sur une plateforme commerciale à permis un gain de temps considérable en terme de développement et maintenance des logiciels embarqués, mais il a fallu apporter de nombreuses modifications, par exemple ajouté une caméra (RGB, Capteur de profondeur) *Intel RealSens SR300*[36] et un capteur de pression (cellule de charge) pour mesurer la force de pousser des objets, car ces deux capteurs ne sont pas fournis sur cette plateforme.

Les simulations présentées sont réalisées sur un ordinateur de bureau avec un processeur *Intel i5* 2400 Mhz, *8 Go* de RAM, une carte graphique *NVIDIA Quadro P600*. La distribution installée est *Ubuntu 16.04* avec le noyau *Linux 4.15*.

Lors des simulations réelles, les logiciels de traitement de haut niveau ne sont pas embarqués sur le robot. Les capteurs du robot envoient leurs informations par WiFi à la station de travail, pour les traitements, et cette dernière revoit les informations pour les actionneurs, très peu de traitement sont réalisés sur le robot. Cela est possible grâce à l'utilisation de ROS Client/Serveur, deux instances *ROS client* sont embarquées sur le robot, et ROS serveur est installé sur la machine de bureau. Lors de l'utilisation de la simulation, le simulateur *Gazebo* se substitue au robot race à l'utilisation de *ROS control*, cette manipulation est transparente pour les nœuds de traitement.

3.8 Conclusion

Dans ce chapitre, nous avons décrit VICA, une architecture robotique appliquée à un robot mobile fonctionnant dans un environnement congestionné (NAMO). Nous avons produit un

[35]Cet aspect n'est pas très bien décrit dans la littérature, en dehors des robots manipulateurs.

[36]L'ajout de la caméra *Intel RealSens SR300*, à révélé une incompatibilité des ports (USB2 / USB3), il a fallu aussi rajouter un nouveau nano-ordinateur *Raspberry Pi 4* doté de ports USB3 pour gérer exclusivement la caméra.

modèle avec une représentation multi-agents de l'environnement, où le robot est capable d'informer et de comprendre son évolution, tout en agissant et en modifiant son comportement de manière appropriée.

L'architecture robotique que nous avons mis en oeuvre sous le nom de VICA (**VI**carious **C**ognitive **A**rchitecture) est à mis-chemin entre une architecture robotique et une architecture cognitive. Architecture robotique, parce que elle intègre les divers aspects permettant la mise en place d'un robot autonome. Architecture cognitive, car le niveau décisionnel intègre un haut niveau d'abstraction, inspiré du fonctionnement du cerveau proposé par Alain Berthoz.

Au début de nos travaux de recherche sur la NAMO, nous avons été inspiré par les travaux d'Alain Berthoz sur la vicariance et simplixité [Berthoz and Petit, 2014] [Berthoz, 2013]. Le mot *Vicariance* fait référence à un système capable de s'adapter à diverses situations, autrement dit, le fait qu'un système peut être utilisé pour d'autres fonctions pour lesquelles il est prévu. La *Simplixité* c'est la capacité qu'a un système de représenter une situation complexe d'une façon plus simple afin d'appliquer des processus de résolution plus génériques. C'est effectivement le but recherché dans notre travail pour optimiser la navigation en milieu congestionné.

VICA modélise des comportements de navigation de type humain face à des obstacles. Souvent, le cerveau humain, lorsqu'il cherche des solutions, choisis des solutions qui nécessitent le moins d'effort. Par exemple, si l'on souhaite passer de l'autre côté soit en faisant le tour de la table, soit en déplaçant la chaise qui gêne le passage, la seconde solution est souvent choisie. Le planificateur proposé ici, engendre un comportement que nous pensons proches du comportement humain. Il propose de choisir le chemin le plus court, même s'il implique de bouger des obstacles. Après avoir échoué à déplacer des obstacles, il envisage d'autres actions.

L'hypothèse selon laquelle l'environnement pourrait offrir au robot les connaissances acquises grâce aux interactions, a été confirmée par les résultats qui montrent l'efficacité du modèle. VICA propose un comportement intelligent du robot pour la navigation, en évitant un maximum d'obstacles tout en assurant de parcourir une distance minimale.

Cette architecture est ensuite validée par des résultats. Mais il reste à confirmer son interaction avec des objets plus complexes (êtres humains, animaux de compagnie, d'autres robots, etc.) et des configurations de scènes complexes pour vérifier que le robot est capable d'évoluer dans un environnement véritablement complexe et naturel. Des travaux supplémentaires sur la représentation des connaissances sont nécessaires pour être compatibles avec différents types d'entités évoluant dans l'environnement. Le module de perception multimodale sera complété pour l'extraction de toutes les possibilités d'action sur un objet (Prospects). Une amélioration envisageable est de doter VICA d'un module d'apprentissage avec une connaissance progressivement dynamique assurerait la meilleure configuration pour l'objectif dans n'importe quel environnement.

4
Planification globale

Dans le chapitre précédent, nous avons décrit la structure générale de l'architecture robotique permettant la navigation parmi les obstacles amovibles (VICA). Nous avons identifié trois grandes fonctionnalités que doit assurer ce système. À savoir (1) *la planification globale*, qui permet de trouver la stratégie de navigation, (2) *la planification locale pour la gestion des obstacles* en se basant sur des simulations et finalement (3) le *choix des actions* et le retour d'expérience après les actions pour assurer la sécurité lors du mouvement.

Dans ce chapitre, nous allons présenter en détail notre planificateur global. Ainsi, nous allons présenter une vue d'ensemble de la solution recherchée, puis faire un état de l'art des solutions existantes et finir par proposer une solution efficace à notre problème et présenter les résultats obtenus.

Le planificateur global que nous proposons ici a pour objectif de déterminer un chemin qui permet de rapprocher le robot le plus possible de son objectif en **évitant les obstacles**. Si des obstacles se trouvent sur son passage alors le système confie la tâche au planificateur local, qui permettra de résoudre la situation (s'il y a une solution). Sinon le planificateur global reprendra la main pour trouver un autre chemin.

Dans la suite de ce chapitre, nous allons présenter quelques méthodes utilisées dans le cadre de la planification globale. Par la suite, nous allons proposer une solution permettant d'atteindre cet objectif et les comparer avec les solutions existantes et finalement nous allons discuter des avantages et des inconvénients.

4.1 La localisation

La localisation consiste à déterminer la position et l'orientation du robot[1] par rapport à des repères fixes (appelé aussi amers) dans son environnement. Pour les robots, les manipulateurs avec une base fixe, cette problématique ne se pose pas, car il est facile de connaître la position de chaque articulation du robot par rapport à la base fixe. Au contraire pour les robots mobiles dans un environnement ouvert, il n'y a pas de liens fixes avec l'environnement ce qui rend la tâche difficile. En extérieur, ce problème se pose moins en raison de l'utilisation des GPS, par contre en environnement intérieur le robot doit faire appel à des repères fixes dans l'environnement pour se situer.

On peut définir la localisation comme une estimation de la posture du robot dans son environnement par rapport à des repères fixes. La posture courante du robot à l'instant t est localement évaluée en fonction de la posture précédente $(t-1)$. Donc la localisation en robotique consiste à estimer la position du robot dans son environnement par rapport à un repère (généralement la position initiale du robot), en utilisant les données historiques de ses observations, l'historique des commandes et les connaissances de l'environnement, la localisation se fait alors de façon itérative, c'est-à-dire la posture courante du robot est localement évaluée en fonction de la posture précédente. Le problème de la localisation en robotique peut être vu comme l'estimation de la probabilité de distribution.

soit :

- x_t : position du robot à instant t (*vecteur d'état*).
- u_k : L'application de u_t à l'instant $t-1$, c-à-d l'action du robot qui mène le robot de l'état x_{t-1} à l'état x_k(*vecteur de contrôle*) ?.
- m_i : position de l'amer i.
- z_t : L'observation à l'instant t.
- $X_{0:t}$: l'ensemble des vecteurs d'état (x_0,x_1,...,x_t) de t_0 à t.
- $U_{0:t}$: l'ensemble des vecteurs de commande (u_0,u_1,...,u_t) de t_0 à t.
- $Z_{0:t}$: l'ensemble des observations (z_0,z_1,...,z_t) de t_0 à t.
- m : la carte de l'environnement avec l'ensemble des obstacles.

Alors, on peut définir la localisation à l'instant t comme suite :

$$P(x_t|Z_{0:t}, U_{0:t}, m)$$

Dans cet état, l'estimation de la position du robot à l'instant t prend en compte l'ensemble des observations précédentes de t_0 à t. Ce qui permet d'obtenir une estimation des positions

[1]Le couple position/orientation, noté (x, y, θ), est souvent désigné par un seul mot **pose** dans la littérature, nous allons adopter cette convention sans la suite de ce rapport.

précédentes, mais augmente la complexité des calculs. Pour simplifier les calculs, on utilise uniquement les données à l'instant précèdent soit $t-1$:

$$P(x_t|z_{t-1}, u_{t-1}, m)$$

Cette simplification permet de gagner en temps de calcul, mais rend impossible toute correction à posteriori en cas de dérive. Donc il est important de stocker les différentes observations de l'environnement au long du déplacement du robot. Ces informations constituent une véritable **carte de l'environnement**, car à tout moment, ces informations peuvent servir à reconstituer son état antérieur, ou être utilisées pour construire une stratégie de navigation.

4.1.1 Cartographie

La cartographie consiste à créer une représentation de l'environnement à partir des données des capteurs. Cette représentation (carte) est construite à partir des informations connues *a priori*, reçus via ses capteurs. Elle servira essentiellement à estimer la pose du robot dans l'environnement et à établir des stratégies de navigation.

Cette représentation est souvent incomplète et elle n'est pas figée dans le temps, elle est susceptible d'être modifiée ou améliorée au fur à mesure du déplacement du robot. Cette représentation est stockée sous forme de cartes métriques et/ou des cartes typologiques.

- Cartes métriques ou géométrique : Cette classe de modèle permet de représenter l'environnement par sa géométrie avec un référentiel fixe (généralement la position initiale du robot). Cette représentation permettra de calculer les chemins et les trajectoires en se basons sur des cartes métriques. Pour ce faire, l'espace géométrique est transposé dans l'espace des configurations du robot. Un robot même avec une géométrie complexe est représenté par un point dans l'espace des configurations [Siegwart et al., 2011]. Il existe dans la littérature une très grande variété de représentation métrique que l'on peut regrouper en deux familles : Les méthodes basées sur une grille et les méthodes basées primitive.

  Les méthodes basées sur une grille, découpent l'espace 2D en un millage structuré (grille), chaque élément du millage est appelé cellule et représente une portion de l'espace [Elfes, 1989a] [Elfes, 1991] [Elfes, 1989b], initialement cette technique suppose que l'environnement est statique en raison de la difficulté liée à la mise à jour de la grille qui implique une lourde capacité de calcul. Depuis, cette technique, est étendu pour être utilisée par exemple dans : SLAM (Simultaneous Localisation and Mapping) pour mettre à jour la carte à chaque nouvelle position du robot [Levinson and Thrun, 2010] [Steux and El Hamzaoui, 2010], MOT (Moving Object Tracking) qui représente les objets avec des cellules auxquelles on associe un vecteur vitesse [Coué et al., 2006], d'autres travaux utilise ces deux techniques simultanément SLAMMOT (Simultaneous Localisation and Mapping Moving Object Tracking) [Gate, 2009]. Dans certains travaux, on représente même la hauteur des cellules dans une grille dite $2,5\ D$, pour chaque cellule, on associe une valeur qui représente sa hauteur [Himmelsbach et al., 2008], enfin d'autres approche divise l'environnement en un ensemble de cubes dans une grille dite $3\ D$ [Himmelsbach et al., 2008].

Les méthodes basées primitive utilisent les primitives géométriques pour représenter l'environnement. Dans ce cas, on représente uniquement des objets souhaités généralement sur une plan $2D$ qui forme un plan horizontal vue de dessus de l'environnement, on représente alors chaque objet (obstacle, amers, robot, etc.) sous forme de coordonnées auxquelles est associé des données permettant de décrire l'objet et aussi pour permettre de propager de façon cohérente les primitives dans le temps. En terme d'application, on trouve « Featured SLAM »[Montemerlo et al., 2002] qui utilise cette technique pour représenter l'environnement sous forme d'un vecteur d'état qui contient la position des amers.

Il existe aussi des approches qui utilisent ces deux techniques simultanément, cette approche est dite *hybride*. Par exemple dans [Himmelsbach et al., 2008] les auteurs utilisent une grille $2,5\ D$ pour représenter l'environnement, par la suite, ils ajoutent des primitives géométriques pour la représentation des amers.

- Cartes topologiques : La localisation topologique [Paletta et al., 2001] [Kröse et al., 2001] [Sim and Dudek, 1999] consiste à trouver les nœuds dans le graphe qui représente l'environnement à partir de données issus des capteurs du robot [Remazeilles, 2004]. Donc il n'est pas question de trouver une position, mais de trouver une situation dans un graphe [Siegwart et al., 2011]. L'avantage de la localisation topologique, c'est qu'elle est peu sensible aux erreurs de mesures contrairement à la localisation métrique. La précision de localisation topologique dépend de la précision de la description de l'environnement.

  Il est possible de calculer les chemins et les trajectoires en se basons sur des cartes topologiques. Dans ce cas, l'itinéraire planifié est constitué d'une suite de situations à atteindre. Celles-ci peuvent être exprimées dans un repère associé à la scène ou dans le repère capteur. L'itinéraire est calculé en utilisant également les méthodes issues de la théorie des graphes. En fonction du degré de précision avec lequel est décrit l'environnement, il est possible que le chemin planifié ne tienne pas compte de l'ensemble des obstacles.

Pour chaque type de carte utilisée pour la modélisation de l'environnement, on trouve une méthode de localisation qui est associée. La localisation métrique consiste à calculer la position du robot à partir de la carte métrique. On utilise généralement des données issus des capteurs proprioceptifs et extéroceptifs pour calculer la position du robot dans un repère local ou globale. Il est à noter que l'utilisation unique de capteurs proprioceptifs est possible, mais engendre des problèmes de dérive qui peuvent apparaître généralement liés au phénomène de glissement des roues, d'erreurs de modélisation du robot etc[Cobzas and Zhang, 2001] [Wolf et al., 2002]. Donc, pour améliorer cette localisation, il est nécessaire de coupler les données issues des capteurs proprioceptifs et extéroceptifs par exemple avec des techniques tels que l'odométrie visuelle [Nistér et al., 2004] [Comport et al., 2005] [Cheng et al., 2006].

L'environnement peut être représenté d'une façon continue ou discrète. Dans le premier cas, le calcul de chemin et de trajectoires se fait généralement en utilisant des graphes de visibilités ou des diagrammes de Voronoi [Tessellations, 2000]. Dans le second cas, la planification est calculée à l'aide d'algorithmes utilisés en théorie des graphes tel que l'algorithme de Dijkstra [Barbehenn, 1998] ou $A*$ [Duchoň et al., 2014].

La modélisation de l'environnement peut être incomplète et le robot l'enrichit au fur et à mesure de la navigation. Dans ce cas, le robot ne dispose pas de la position des obstacles. Plusieurs solutions sont proposées, nous pouvons citer par exemple [Lamiraux et al., 2004] qui propose de considérer la trajectoire comme une bande élastique déformable. Cependant une replanification globale peut être requise pour une modification majeure de l'environnement.

4.1.2 Navigation

La *localisation* permet au robot de répondre à la question « *Où suis-je ?* », donc elle permet de donner une indication sur la position du robot dans son modèle d'environnement. La *navigation* quant à elle, elle permet au robot de répondre à la question « *Comment aller de la position A et rejoindre le but B ?* », et aussi répondre d'autres questions telles que : « *Comment reconnaître B ?* » et « *Comment éviter les obstacles ?* », etc. Donc la navigation est un processus calculatoire permettant au robot, à partir du modèle de l'environnement, de déterminer un *itinéraire*[2] d'une position donnée pour rejoindre une position finale (but). Pour cela, il existe différentes stratégies de navigation [Trullier and Meyer, 1997] [Trullier et al., 1997]. On peut distinguer les grandes catégories : (1) les stratégies sans modèles internes et (2) les stratégies avec modèles internes. La première catégorie est considérée comme étant naïve, car elle n'implique pas une sauvegarde des situations précédemment rencontrées, mais ces méthodes sont plus rapides et moins gourmandes en mémoire. Plus généralement la seconde catégorie est plus utilisée quand les moyens de calcul et de stockage le permettent.

La navigation dans un environnement inconnu impose au robot de se localiser dans un environnement inconnu, or pour se localiser, il faut au préalable une cartographier l'environnement et inversement. La localisation et cartographie simultanées connue sous l'acronyme SLAM[3] [Stachniss et al.,] permet à un robot mobile de naviguer à partir d'une position inconnue dans un environnement inconnu en construisant simultanément la représentation de l'environnement et la posture du robot.

L'une des façons de classer ces stratégies de navigation selon [Trullier and Meyer, 1997] [Trullier et al., 1997], on peut distinguer cinq catégories, de la plus simple à la plus complexe :

1. **Approche d'un objet par vison direct** Sans modèle interne et avec une stratégie locale[4], elle présente la capacité basique de diriger le robot vers un objet visible. Généralement, cette technique est réalisée à l'aide d'une fonction mathématique simple, par exemple une descente ou remontée du gradient. L'une des premières implémentations de cette célèbre technique [Braitenberg, 1986] (véhicules de Braitenberg) présente un simple robot équipé de deux capteurs de lumière qui sont reliés directement aux commandes des moteurs, le but à atteindre est une source de lumière visible depuis la position du robot. Le robot est simplement une plateforme différentielle, elle est constituée de deux roues et la commande des vitesses de rotation des roues est déterminée par l'intensité lumineuse détectée par les capteurs, alors le robot se dirige naturellement vers la source lumineuse. Cette technique est difficilement utilisable pour des applications réelles.

[2]Un itinéraire peut consister en un chemin, une trajectoire ou une suite de situations à atteindre dans l'espace des configurations.

[3]Simultaneous Localisation and Mapping

[4]Fonctionne uniquement dans les zones de l'environnement où le but est visible.

2. **Guidage vers un objet par vison direct** Aussi sans modèle interne et avec une stratégie locale, elle présente la capacité de guider le robot vers un but visible, mais en utilisant des points de repères appelés *amers*[5]. L'objectif n'est pas forcément un objet, mais une position relative par rapports aux amers. Cette technique est inspirée des comportements des abeilles [Cartwright and Collett, 1987]. Plusieurs implémentations de robots ont utilisé cette technique [Gaussier et al., 2000] [Gourichon et al., 2002] [Lambrinos et al., 2000]. Cette technique est également réalisée à l'aide de fonctions mathématiques simples tel que la descente du gradient. Cette technique est difficilement utilisable pour des applications réelles.

3. **Action associée à un lieu** Cette technique décrite dans [Remazeilles and Chaumette, 2007] permet de réaliser une navigation globale, c'est-à-dire atteindre un but qui n'est pas visible depuis la position du robot. Elle requiert un modèle interne de l'environnement qui consiste à diviser l'environnement en un ensemble de lieux dans lesquelles les perceptions restent similaires et puis associer une action chaque lieu. L'exécute d'une action à un lieu donnée permettra le déplacement au lieu suivant. L'enchaînement des actions forme au final un chemin (voir Figure 4.1).

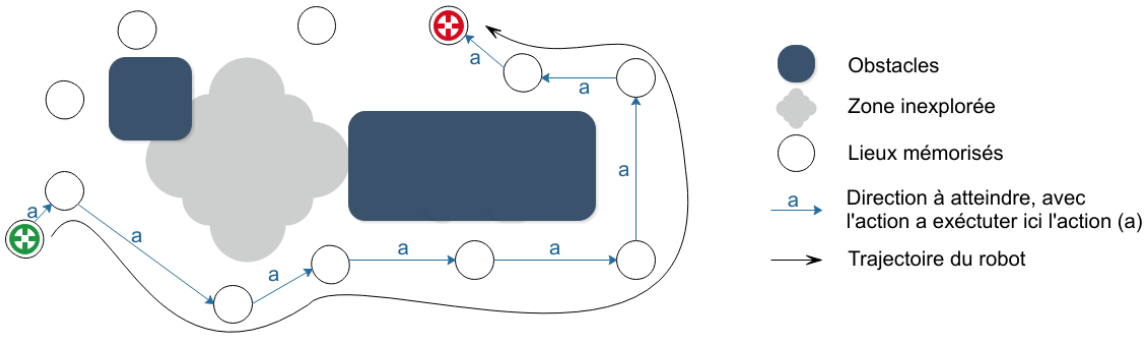


FIGURE 4.1 : Action associée à un lieu : Chaque lieu encode une action (a) qui permettra au robot de rejoindre une nouvelle position lui le mènera (a long terme) de rejoindre son but. Dans cet exemple, le point de départ est matérialisé par une cible verte et le point d'arriver en rouge. Le robot enchaîne les actions, un seul chemin unique lui permettra d'atteindre son objectif, le raccourci de gauche n'est pas utilisable.

4. **Navigation topologique** Cette technique permet à un robot de mémoriser un ensemble de lieux et les possibilités de passer de l'un à l'autre, indépendamment de tout but. Pour rejoindre un but, il faut alors une étape de planification qui permet de rechercher, parmi tous les chemins possibles, le chemin rejoignant le but. En pratique, la représentation est réalisée sous forme d'un graphe sur lequel on applique des algorithmes de recherche de chemin entre deux lieux. Cette technique ne permet que la planification de chemins entre deux lieux connus et en suivant des chemins connus uniquement.

5. **Navigation métrique** Cette technique permet à un robot de planifier des chemins au sein de zones inexplorées de son environnement en utilisant une carte métrique. Elle mémorise pour cela les positions métriques relatives des différents lieux, en plus des liens

[5] Amer : *landmark* ou *beacon* en anglais.

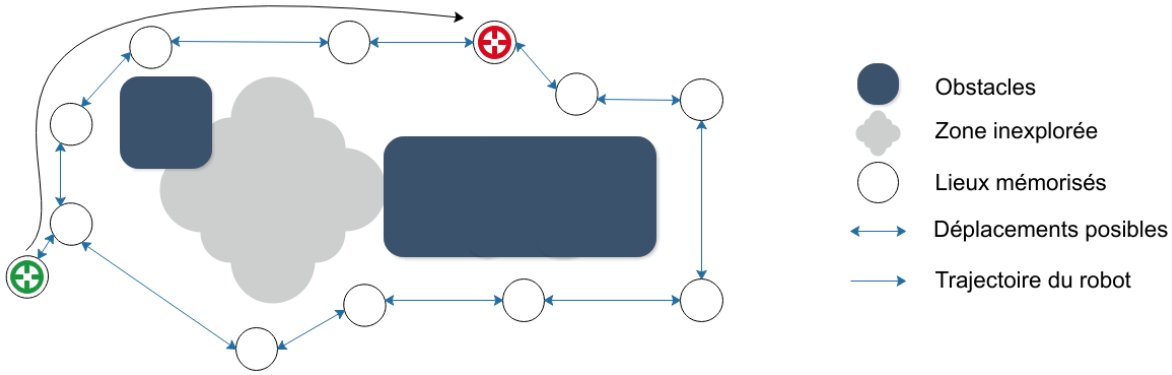


FIGURE 4.2 : Navigation topologique : Permet de mémoriser un ensemble de lieux et les interconnexions entre ces lieux, au final cette représentation forme un graphe. Pour rejoindre un but, on utilise des techniques de la théorie des graphes pour trouver un chemin entre deux lieux. Ici le robot cherche le plus court chemin entre la cible de départ verte et la cible rouge, deux chemins possibles, un à droite et l'autre à gauche, le chemin de gauche est sélectionné selon des critères de distance.

qui existent entres les lieux. Ces positions relatives permettent, par composition de vecteurs, de calculer une trajectoire allant d'un lieu à un autre, même si la possibilité du déplacement n'est pas représentée sur la carte sous forme d'un lien.

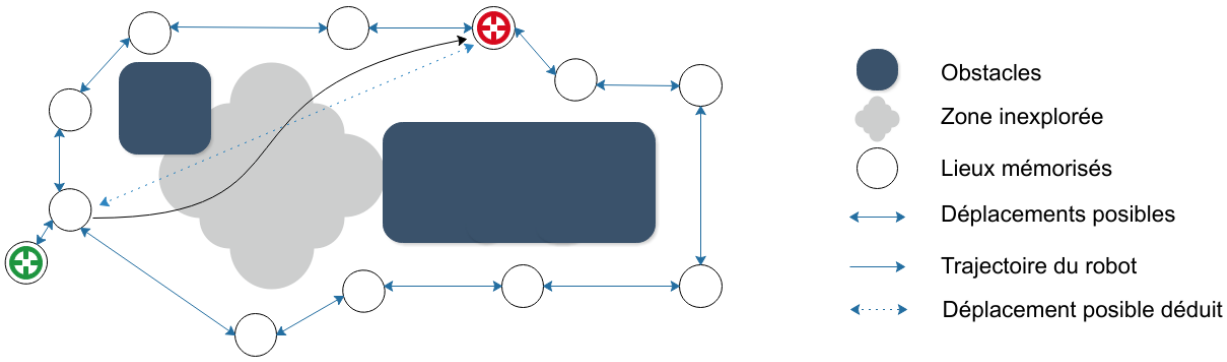


FIGURE 4.3 : Navigation métrique : Permet de calculer le plus court chemin entre deux lieux et même de planifier des raccourcis entre deux lieux dans une zone inexplorée.

Les trois premières méthodes de navigation présentées précédemment utilisent des actions réflexes pour déterminer la trajectoire du robot, elles sont souvent regroupées sous le terme *navigation réactive*. La différence entre ces méthodes se situe essentiellement au niveau des percepts utilisés pour déclencher les actions.

La navigation topologique et la navigation métrique sont des techniques utilisées pour le déplacement d'un robot est elle répondent parfaitement à la question posée précédemment « *Comment aller de la position A et rejoindre le but B ?* ». Elles donnent la capacité au robot de mémoriser dans son modèle interne les relations spatiales entre les différents lieux. Ces relations indiquent la possibilité de se déplacer d'un lieu à un autre, mais ne sont plus associées à un but particulier. Ainsi, le modèle interne est un **graphe** qui permet de calculer différents

chemins entre deux lieux arbitraires. Ces méthodes, elles permettent au robot de planifier des chemins au sein de zones déjà explorées ou inexplorées de son environnement.

Le processus permettant de trouver un chemin en utilisant un modèle est appelé planification. L'objectif de la planification consiste à trouver un chemin ou une trajectoire dans l'espace des configurations libre permettant d'atteindre la configuration finale (but) [Choset et al., 2005].

Les actions nécessaires à la navigation, sont réalisées à travers des commandes. Il est important pour le robot de connaître l'état des capteurs après exécution des commandes de mouvement afin d'appliquer d'éventuelles corrections et minimiser les erreurs. Les retours peuvent être de deux types : retour d'état ou retour de sortie [Levine, 2018]. La commande par retour d'état est une technique permettant de réduire au maximum l'erreur entre l'état courant et un état de référence correspondant à l'état final désiré. La commande par retour de sortie est une technique permettant de réduire au maximum l'erreur entre l'état courant mesuré par les capteurs du robot et un état de référence.

La navigation d'un robot d'un point initial pour rejoindre un but, est un processus itératif composé d'une suite d'actions citées ci-dessus (localisation, navigation, planification, etc.), il peut être nécessaire de prendre des décisions à différents stades du processus. Une fois que la trajectoire est déterminée, il faut **contrôler** le robot afin de l'amener à exécuter le mouvement tel qu'il a été prévu. Du fait de l'incertitude qui caractérise aussi bien l'environnement, la nature des obstacles, etc. Il est important d'avoir un retour du déroulement des mouvements par rapport au plan initialement prévu. L'exécution du mouvement se fait à travers des commandes envoyées aux actionneurs. Les capteurs embarqués sur robot permettent de donner une indication de la façon dont le mouvement se déroule, ainsi il est possible l'adapter à chaque instant.

4.1.3 L'environnement

L'environnement de l'utilisation du robot détermine les méthodes à utiliser en terme de cartographie, localisation, méthode de navigation, etc.

L'environnement domiciliaire est hostile pour un robot autonome, car il présente les caractéristiques suivantes :

- **Environnement congestionné** : densité importante d'obstacles, avec possibilité que certaines positions de l'environnement soient obstruées et ne sont pas accessibles pour le robot avec évitement.
- **Environnement dynamique** : la position des obstacles peut changer au cours du temps.
- **Environnement fréquenté par des humains ou des animaux**.
- **Environnement inconnu** : le robot ne dispose pas de carte préalablement enregistrée de son environnement.

La solution la plus efficace dans ce cas est de demander à l'humain de se décaler afin de dégager le passage, au contraire un animal ne pourra pas interpréter les demandes du robot ainsi,

il serait judicieux le contourner. 1) le but n'est pas accessible 2) éviter un obstacle demande de parcourir une distance importante pour le contourner. Donc le robot est amené à déplacer des obstacles, pour dégager un passage, si son but est obstrué par des obstacles amovibles. Donc il est impératif que le robot sache distinguer un obstacle qui capable de bouger d'un obstacle fixe. De plus, si un humain obstrue le passage, il est dangereux d'entrer en collision. La solution la plus efficace dans ce cas est de demander à l'humain de se décaler afin de dégager le passage, au contraire un animal ne pourra pas interpréter les demandes du robot ainsi, il serait judicieux le contourner.

Un robot qui puisse navigue dans un tel environnement, il est amené an définir des chemins et trajectoires efficaces et en toute sécurité pour atteindre son but. Donc avant même que le robot commence à se déplacer, il doit être capable de **se localiser** par rapport à repère fixe de son environnement. Lorsque ce dernier est inconnu, il doit être capable de **cartographier** et **identifier** les obstacles qui constituent son voisinage immédiat, afin d'établir une stratégie de navigation et disposer de points de repères supplémentaires pour améliorer sa localisation. Le robot doit reposer sur ses capteurs qui lui fournissent ces informations. Des capteurs *proprioceptifs* qui donnent des informations internes sur l'état interne du robot (odomètres, gyroscopes, etc.), extéroceptifs qui donnent des informations sur l'état du monde extérieur (caméra, LIDAR, GPS, etc.).

Pour qu'un robot puisse définir des chemins et trajectoires efficaces et en toute sécurité pour atteindre son but. Il lui faut cartographier préalablement au moins une partie de son environnement afin d'établir un chemin, par la suite la gestion des obstacles rencontrés durant le trajet sont des décisions locales. Donc la décision du mouvement à entreprendre s'effectue donc généralement en deux temps. 1) si le robot dispose d'une connaissance à priori sur son environnement, il peut alors planifier un mouvement complet lui permettant de rejoindre son but, sinon il faut faire des planifications partielles en attenant d'avancer et améliorer sa connaissance de son environnement. 2) Ensuite, au cours de l'exécution du mouvement, il devra être capable d'éviter les obstacles inattendus ou dont le comportement varie en adaptant son mouvement ou l'action à mener.

4.2 Les algorithmes de planification

La planification de mouvement est une étape importante de la navigation. Elle permet au robot de trouver une séquence de configurations valides qui lui permet de se déplacer d'un point A pour rejoindre un but B. Ce problème est connu en robotique sous le terme de *planification de mouvement* (*en anglais : Motion planning*. Historiquement, il a été décrit dans [Schwartz and Sharir, 1983] appelé aussi « problème du déménageur de piano ». Il stipule que si un mobile dans une position et une orientation initiale veut se déplacer vers une position et une orientation finale, il doit exister au moins un chemin valide entre ces deux positions. Un déplacement est dit valide, s'il est intégralement effectué sans collision. La planification de mouvement consiste donc à trouver pour un mobile une trajectoire valide. Une méthode de planification de mouvement doit : (1) soit générer un mouvement tel que l'objet peut rejoindre la position finale sans entrer en collision avec les obstacles de l'environnement, (2) soit conclure qu'un tel mouvement est impossible.

Un problème de planification de mouvement est spécifié par la configuration de l'environnement et la posture du robot. L'objectif et de trouver un chemin entre deux configurations du robot dans l'environnement (position initiale et position finale). La configuration de l'environnement est généralement connue sous le nom de *espace de travail* noté W, lui-même composé d'un ensemble *d'obstacles* rigides notés O_i. L'orientation des obstacles est définie par une transformation qui permet de passer de référence de l'espace F_W à un repère F_{O_i}.

Un robot est représenté comme un ensemble d'éléments rigides qui sont reliés par des liaisons cinématiques. L'ensemble de ces liaisons forment *la chaîne cinématique* du robot. Pour un robot mobile doté d'un bras manipulateur (pince, main, etc.) sa posture dans l'espace de travail W peut être défini grâce à un ensemble de paramètres nommés *degrés de liberté*, dans ce cas, la géométrie du robot appelée aussi *configuration* évolue dans un espace 3D ou 2D. Dans le cas d'un robot mobile équipé uniquement d'une base mobile comme dans notre situation, la posture du robot est fixe et occupe toujours le même volume dans l'espace de travail W. Ainsi l'espace des configurations noté C dépend uniquement des propriétés du robot (dimensions) et ne peut évoluer dans le temps. Nous avons fait le choix dans notre étude de nous limiter à un robot mobile sans articulations, pour des besoins de simplifications.

4.2.1 Méthodes de planification globale

Les algorithmes de planification de mouvement sont divisés en deux grandes classes : (1) les méthodes déterministes appelées aussi les méthodes exactes, qui permettent de retrouver le même chemin à chaque exécution pour une configuration donnée de l'environnement. (2) Les méthodes probabilistes appelées aussi méthodes par échantillonnage, quant à elles, peuvent trouver des chemins différents pour les mêmes conditions initiales, mais elles garantissent de trouver une solution si elle existe ou de déterminer qu'une solution est inexistante définitivement. Dans la section suivante, nous allons détailler quelques méthodes de planification de mouvement sur lesquelles nous nous basons, mais un état de l'art détaillé se trouve dans [LaValle, 2006].

Les roadmaps

Les solutions apportées par la première classe d'algorithme se basent sur la topologie de l'espace des configurations libres des obstacles, notée C_{free}, afin de construire un graphe. Donc elles consistent à réduire le problème à une simple recherche d'un chemin dans un graphe. La technique se déroule en deux étapes : La première consiste à construire un graphe conforme à l'espace des configurations. La seconde étape, fait appel aux algorithmes de recherche de chemins dans un graphe par exemple : $Dijkstra$, A^*, $Bellman$, etc. Quant à l'étape de construction des graphes elle s'appuie par exemple sur les méthodes suivantes :

- **Décomposition en cellules :** Elle consiste en une représentation discrète de l'environnement du robot. Le résultat d'une décomposition cellulaire donne en résultat un ensemble de polygones subdivisant l'espace des configurations libres C_{free}. À partir de cette représentation, on créé un graphe reliant les centres des polygones adjacents. Finalement, il suffit de déterminer les polygones qui contiennent le point de départ et le point d'arrivée pour déminer un chemin à l'aide d'un algorithme de parcours ($Dijkstra$, A^*, $Bellman$, BFS, DFS, etc). Pour obtenir un chemin sans collisions, il faut en définitive relier les cellules par des arêtes pour obtenir le graphe solution. On peut, par exemple,

utiliser le point milieu de la frontière deux cellules adjacentes, un exemple d'une méthode de décomposition cellulaire est présentée dans la Figure 4.4.

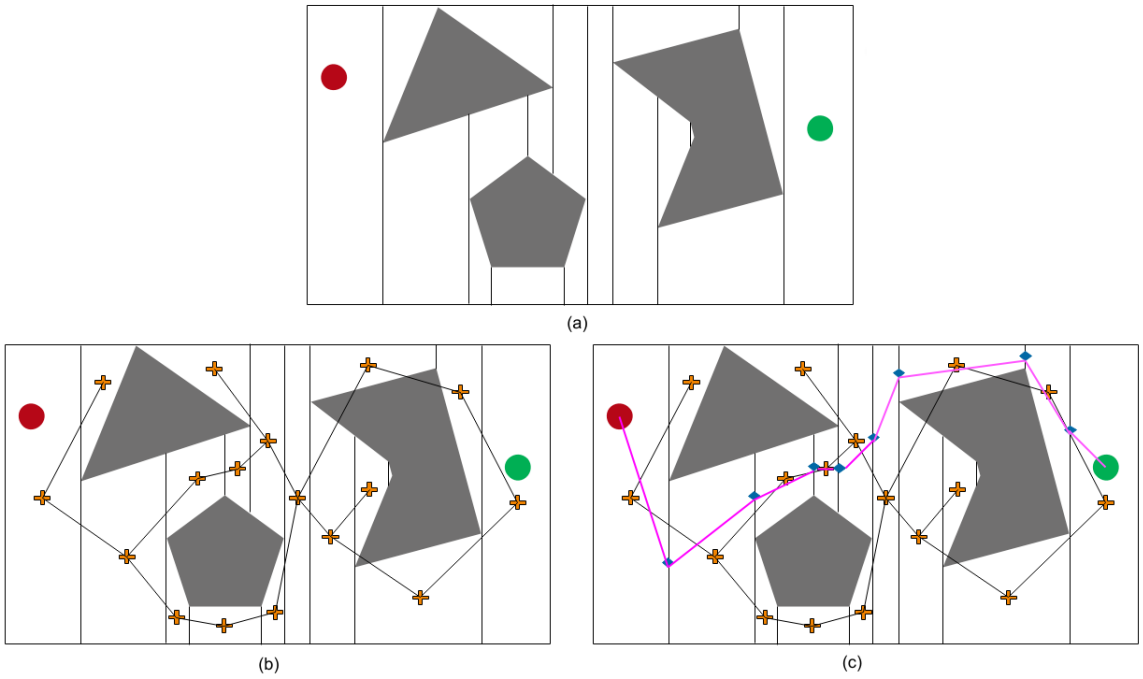


FIGURE 4.4 : La méthode de division cellulaire verticale consiste diviser verticalement l'espace C_{free} (a) en trapèzes et triangles pour chaque sommet des obstacles, puis étendre en lignes verticales de haut en bas (b). Par la suite placer un point au milieu de chaque frontière entre les cellules (c). Finalement relier les centres aux points frontières pour obtenir un chemin entre le point de départ rouge et le point d'arrivée vert.

Une autre technique de division cellulaire consiste à diviser l'espace des configuration C en un ensemble de cellules généralement carrées, comme illustré dans la figure 4.5. Dans cette technique, on considère que chaque cellule est reliée aux huit cellules voisines. L'ensemble des cellules peuvent être représentées sous forme d'un graphe, les cellules représentent les nœuds et les passages d'une cellule à une autre sont représentés par des arêtes pondérées comme suit : Les passages d'une cellule à une autre verticalement ou horizontalement est pondéré à 1, les passages diagonaux sont pondérés à $\sqrt{2}$. Si la cellule représente un obstacle alors les arrêtes vers cette cellules sont pondérées à l'infini. Comme précédemment une variété importante d'algorithmes tels que A^*, D^*, etc, peuvent être appliqués pour chercher un chemin optimal.

- **Graphe de visibilité :** Les graphes de visibilité décrit dans [Nilsson, 1969] ont été utilisés sur le premier robot mobile *Shakey*. L'idée consiste à relier le point de départ au point final par des lignes droites passant par les sommets des obstacles de formes polygonales. Le graphe est construit de la façon suivante : Il faut relier le point de départ aux sommets des obstacles qu'il peut voir directement (ligne droite). En effet, il est toujours possible relier le robot et le sommet d'un polygone avec une ligne qui ne croise aucun obstacle. Les graphes de visibilité présentent toutefois un inconvénient majeur pour la navigation avec évitement d'obstacles, car à chaque sommet, le robot entre en contact avec l'obstacle. Un exemple de graphe de visibilité est présenté dans la Figure 4.6. Il existe

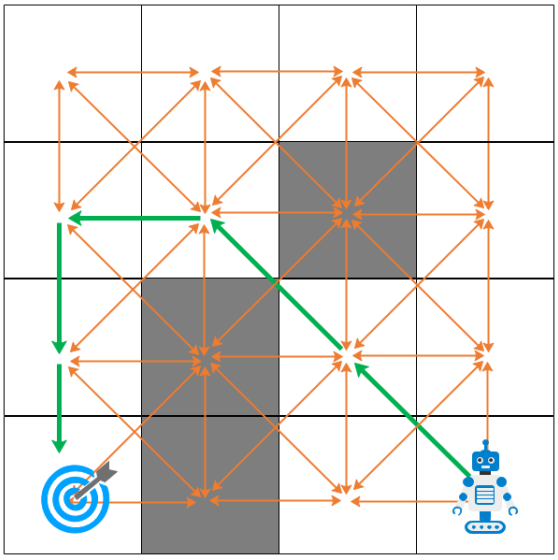


FIGURE 4.5 : Dans cette méthode, chaque cellule est reliée aux cellules adjacentes et les obstacles sont représentés avec des cellules grisées. Ici le robot a trouvé une trajectoire optimale représentée par les flèches vertes.

des solutions pour cette problématique qui consiste à tracer des lignes imaginaires autour des obstacles, la distance entre les obstacles et ces lignes doit être supérieure au rayon du robot, cette technique est connue sous le nom de la technique des bandes élastiques, elle est décrite dans [Quinlan and Khatib, 1993].

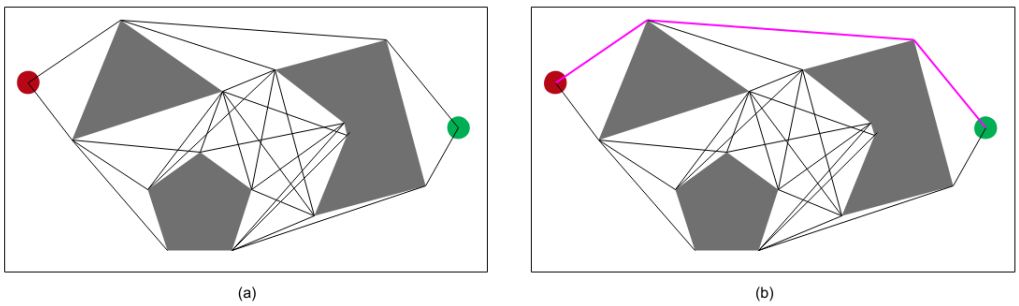


FIGURE 4.6 : À partir d'une représentation 2D du monde, (a) Pour chaque sommet d'un polygone on trace une arête vers les autres sommets visibles et les points de départ et d'arrivée. (b) à partir du graphe obtenu, on applique un algorithme de recherche de chemin.

- **Les diagrammes de Voronoï :** Contrairement aux graphes de visibilité, les diagrammes de Voronoï décrit dans [Takahashi and Schilling, 1989] évitent le contact avec les obstacles et permet au robot de naviguer avec sécurité (loin des bords des obstacles). La construction d'un tel diagramme se fait en traçant des lignes d'égales distances aux obstacles. Ensuite, le graphe est obtenu en raccordant le point de départ au point le plus proche du diagramme et de même pour le point final. La figure 4.7 illustre un exemple d'utilisation de cette méthode.

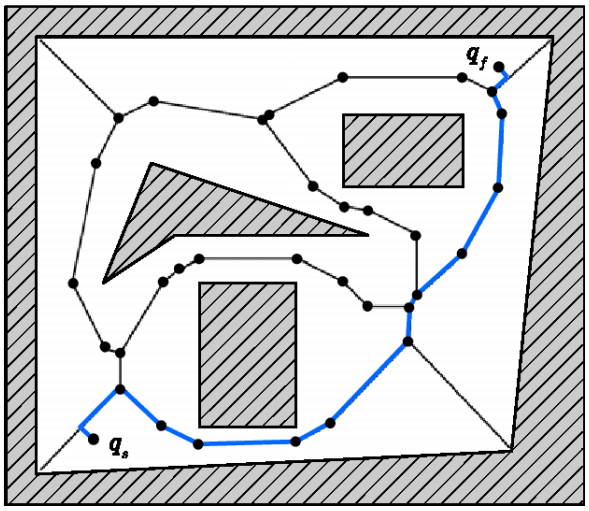


FIGURE 4.7 : A partir d'un certain nombre de points (appelés germes) disposés aléatoirement dans l'espace C_{free} on trace des cellules qui enferment un seul germe, chaque cellule enferme l'ensemble des points du plan plus proches de ce germe que d'aucun autre. Les parois des cellules forment ainsi un graphe partir duquel on calcul un chemin entre le point de départ et le point d'arrivée.

Les méthodes présentées ici (la liste des méthodes présentées ici n'est pas exhaustive, nous n'avons cité que les méthodes les plus répandues dans la littérature.), sont des méthodes dites exactes, elles ont pour avantage de bien fonctionner pour un robot dans l'espace des configurations $C = R^2$. Elles se basent généralement sur des graphes, donc il est facile de calculer les propriétés (exemple : le chemin le plus cours, le coût total, etc.), elles offrent aussi certaines garanties théoriques (complétude, bornes sur le temps d'exécution, etc.). Mais ces méthodes s'utilisent uniquement quand l'environnement et les obstacles sont connus par avance et peuvent s'avérer très lentes si le nombre de dimensions est supérieur à deux, et souvent elles requièrent une représentation algébrique des obstacles ce qui rend leurs implémentations difficile.

Une autre catégorie d'algorithmes dit *algorithmes probabilistes* s'appuient sur l'utilisation de l'aléatoire pour construire un graphe connexe dans l'espace libre C_{free}. Permettant ainsi de s'affranchir d'une représentation exacte de l'environnement. On trouve dans la littérature deux grandes classes :

- **Approche roadmap probabiliste (PRM)** Dites méthodes de réseaux probabilistes, elles sont proposées simultanément par deux équipes [Kavraki, 1994] [Kavraki et al., 1998] sous l'acronyme PRM pour *Probabilistic Roadmap Methods* et [Kavraki et al., 1996] sous l'acronyme PPP pour *Probabilistic Path Planner*. Comme précédemment la technique est constituée de deux phases, la première phase consiste à la construction d'un graphe, cette phase est dite *phase d'apprentissage* et la seconde dite *phase de recherche*, consiste à la recherche d'un chemin dans ce graphe.

  Le graphe représente l'espace libre C_{free}. Pour assurer une exploration totale de l'espace libre, chaque nœud du graphe est tiré aléatoirement selon une distribution uniforme dans C_{free}. À chaque fois qu'un nœud est tiré, on vérifie qu'il existe une ligne droite sans collision avec les nœuds les plus proches du graphe, et qui respecte les contraintes du mobile. Si cette condition est vérifiée alors on ajoute ce nouveau nœud au graphe et on le

connecte aux plus proches nœuds avec des arêtes (Figure 4.8). L'algorithme 4.2.1 décrit plus en détail le fonctionnement.

Algorithme 8 Algorithme Probabilistic Roadmap

Require: i ▷ nombre d'itérations
Ensure: G
  $G \leftarrow \emptyset$
  while $i = i - 1 > 0$ **do**
    $P\_rand \leftarrow Rand()$ ▷ sommet tiré au hasard
    if $P\_rand \in C_{free}$ **then**
      $P\_rand \leftarrow Rand()$ ▷ sommet tiré au hasard
      $k \leftarrow TrouverSommetsProches(G, P\_rand)$ ▷ trouver les k points les plus proches de P_rand dans G
      if $k \neq \emptyset$ **then**
        $G.AjouterSommets(G, P\_rand)$
        $G.AjouterArete(G, k, P\_rand)$ ▷ Essayez de connecter P_rand à chacun des k voisins
      end if
    end if
  end while

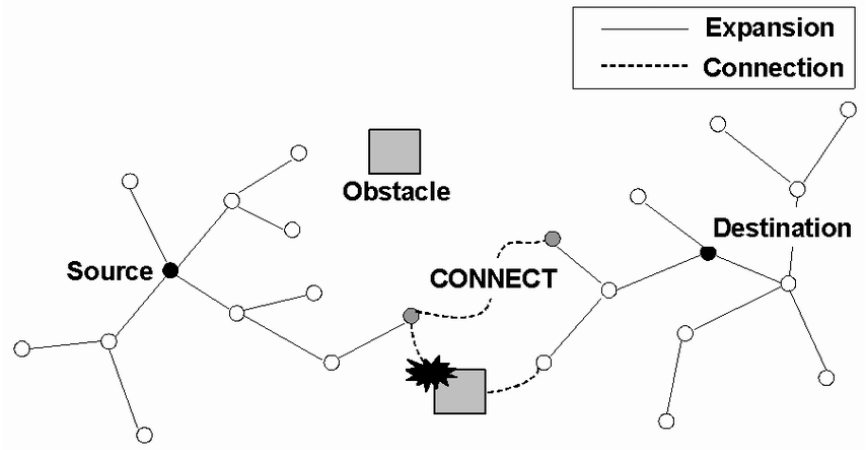


FIGURE 4.8 : Probabilistic Roadmap.

- **Rapidly Exploring roadmap Random Trees (RRT)** Dites méthodes des arbres aléatoires d'exploration rapide connu sous l'acronyme RRT, proposées pour la première fois dans [LaValle, 1998]. Ces méthodes effectuent une recherche aléatoire dans C_free, jusqu'à trouver la configuration finale désirée. Cette méthode ne génère pas une roadmap qui représente d'une façon exhaustive la connectivité de l'ensemble de l'espace C_free, mais explore uniquement une portion de l'espace C_free qui est pertinent à la solution du problème, ce qui réduit le temps de calcul.

  À partir de la position initiale du mobile, on construit successivement un arbre par intégration successive des nœuds. Pour créer un nouveau nœud *new* (Figure 4.9), on génère un échantillon aléatoire de l'espace des configurations *rand* en utilisant une distribution aléatoire pour maximiser l'exploration. La configuration *near* la plus proche à

rand est déterminée, et une nouvelle configuration candidate *new* produit alors un segment joignant *near* à *rand* , à une distance préfixée δ de *near*. Finalement, on vérifie que le segment de *near* à *new* est sans collisions. Si cette condition est vérifiée, on ajoute *new* au graphe et en le relie par un segment *near* à *new*. L'algorithme 4.2.1 décrit plus en détail le fonctionnement. La figure 4.10 montre un exemple concret de l'utilisation de cette méthode.

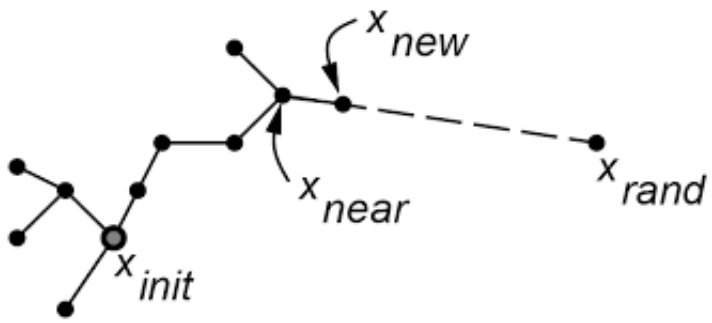


FIGURE 4.9 : Construction d'un graphe à l'aide de la méthode RRT.

Algorithme 9 Algorithme Rapidly Exploring roadmap Random Trees

Require: i ▷ nombre d'itérations
Ensure: G
  $G \leftarrow \emptyset$
  while $i = i - 1 > 0$ **do**
    $P\_rand \leftarrow Rand()$ ▷ sommet tiré au hasard
    $P\_near \leftarrow SommetPlusProche(P\_rand, G)$ ▷ sommet le plus proche de l'arbre G
    $P\_new \leftarrow NouveauSommet(P\_near, \Delta P)$ ▷ nouveau sommet dans C_{free}
    $G.AjouterSommet(P\_new)$
    $G.AjouterArete(P\_near, P\_new)$
  end while

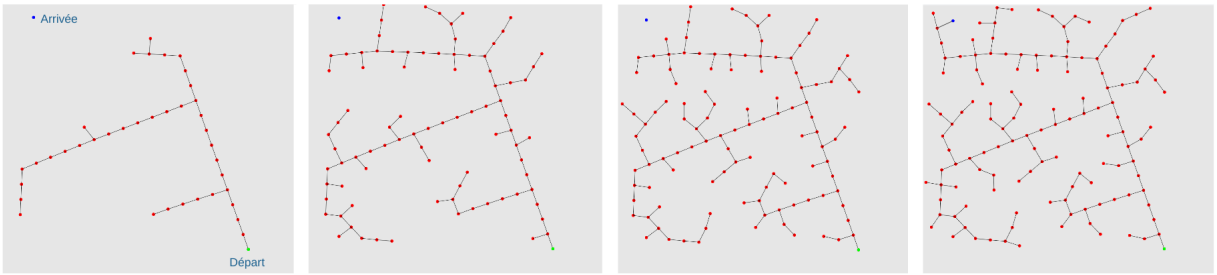


FIGURE 4.10 : Exemple d'expansion du graphe dans un espace sans obstacle, on observe une expansion dans la première image de gauche après 50 tirages aléatoire, puis successivement 100, 150 et 200.

Ces approches dites par échantillonnage ont pour avantage de bien fonctionner dans les environnements de haute dimension avec une représentation arbitraire des obstacles et du robot. Mais les résultats restent toutefois potentiellement non-déterministes et ne garantissent pas la complétude, les bornes sur le temps d'exécution, etc.

4.2.2 Localisation et cartographie simultanées

Les méthodes vues précédemment nécessitent de connaître la configuration de l'environnement et la position des obstacles, or dans un environnement dynamique tel que l'environnement domiciliaire en perpétuel changement, rend ces méthodes inefficaces voir inutilisables. La navigation dans un tel environnement nécessite donc que le mobile construise lui-même la carte de l'environnement et s'y localise.

Donc, un robot mobile se déplaçant dans un environnement partiellement inconnu, ne dispose pas au préalable d'une carte complète de son environnement, il doit relever le défi de se déplacer, en même temps de cartographier son environnement et finalement de se localiser par rapport à son environnement. Le problème a été discuté dans [Bailey and Durrant-Whyte, 2006], appelé couramment SLAM (Simultaneous Localization And Mapping). Un certain nombres d'algorithmes ont été proposés par la communauté scientifique, parmi eux on trouve : **FastSLAM**, **GraphSLAM**, **ORB-SLAM**, **HectorMapping**, *LagoSLAM*, etc. Nous allons présenter ci-après un seul algorithme afin de comprendre les principes, les autres variétés fonctionnent plus ou moins selon le même principe.

Le défi que tente de relever les méthodes SLAM consiste à créer une carte m de l'environnement ($m = [m_1, m_2, \ldots, m_M]^T$ (M points de repères), sur un plan 2D $m_i = [mi, x, mi, y]^T$). Et aussi se localiser dans cette carte en maintenant un estimé X (pose[6] du robot par rapport à la carte m). Lors du déplacement le robot prend des mesures z qui donnent des distances entre entre le robot les obstacles (donc z est la mesure entre X et les repères dans m). La méthode SLAM peut s'écrire sous forme d'une probabilité conditionnelle $P(x|z, m)$.

Cependant les mesures effectuées par les capteurs du robot sont sujettes à des erreurs (les capteurs de mesure de distance entre le robot, les obstacles. Et aussi les capteurs qui mesurent le déplacement du robot). Il est essentiel de réduire les erreurs induites par les capteurs, sans cela, la méthode devient inefficace. Généralement on utilise le Filtre de Kalman[7] ou le Filtre de Kalman Etendu (EKF pour Extended Kalman Filte)[8] pour réduire la part des erreurs[9]. La méthode *SLAM* s'écrit donc $P(x_t, m|z_{1:t}, u_{1:t})$ (u étant les commandes du déplacement du robot). Une autre version du SLAM qui intègre le lissage des trajectoires du robot connue sous le nom *Full-SLAM* s'écrit comme suit $P(x_{1:t}, m|z_{1:t}, u_{1:t})$

Les algorithmes SLAM utilisent majoritairement des scanners laser (2D ou 3D selon l'application) pour récupérer les données de l'environnement et l'odométrie (des roues ou visuelle) pour calculer le déplacement du robot, La Figure 4.11 montre le robot Turtulebot3 qui utilise la méthode SLAM pour cartographier et se localiser dans son environnement.

Les méthodes SLAM sont parmi les méthodes les plus abouties et les plus utilisées actuellement (voitures autonomes, robots aspirateurs, etc.). Elles comptent parmi les méthodes les plus

[6] pose : position + orientation

[7] Méthode de filtrage probabiliste. Elle fait l'hypothèse que les bruits sont gaussiens, le filtre de Kalman délivre une estimation optimale de l'état à filtrer (au sens du minimum de variance)

[8] version non-linéaire du Filtre de Kalman classique

[9] Il existe d'autres méthodes de filtrage de données. On peut citer par exemple *Fast SLAM* qui utilise un filtre à particule au lieu du Filtre de Kalman.

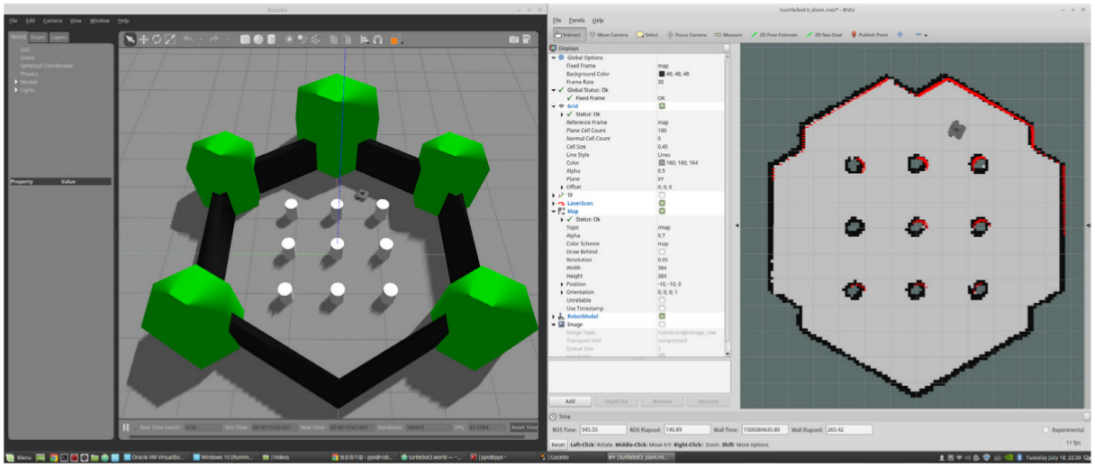


FIGURE 4.11 : Le robot à gauche de l'image évolue dans un environnement simulé il cartographie son environnement à l'aide d'un scanner laser et utilise l'optométriedes roue pour calculer son déplacement. À droite on voit le résultat de SLAM sous forme d'une carte avec la position des obstacles et la position du robot.

robustes en environnements statiques, structurés et de tailles limitées. Mais restent néanmoins un défi pour les environnements non-structurés et dynamiques.

4.2.3 Les algorithmes de recherche de chemins

Jusqu'à présent nous avons présenté quelques techniques permettant de modéliser l'environnement en vue d'appliquer un algorithme de recherche de chemin. Il existe une grande variété d'algorithmes, la différence entre ces algorithmes sont : l'efficacité, la capacité de réagir à des environnements dynamiques ou la possibilité qu'ils puissent opérer en environnement partiellement inconnu.

Parmi les algorithmes de recherche de chemin les plus classiques, on trouve Dijkstra qui n'est rien d'autre qu'un parcours en largeur d'un Graphe. Un autre grand classique, A^* basée sur Dijkstra utilise une heuristique ($f(n) = g(n) + h(n)$) pour diriger la recherche et éviter de parcourir tout le graphe. Parmi les variétés de A^* on trouve $ARA*$ (Anytime Repairing A^*) [Likhachev et al., 2003] qui utilise une heuristique légèrement différente ($f(n) = g(n)+\varepsilon \times h(n)$) où ε est un facteur qui aide à diriger le parcours en largeur.

Il existe un autre variété d'algorithmes qui s'utilisent en environnement dynamiques et inconnues, mais utilisent les mêmes techniques à base d'heuristiques. Parmi ces algorithmes, on trouve $Theta*$, $Field\ D*$, $Lifelong\ Planning\ A*$, D^*, $D*\ Lite$, etc.

Étant donné notre problématique en environnement dynamique partiellement inconnu, nous allons présenter uniquement quelques algorithmes (D^* et $D*\ Lite$) parmi ceux qui sont capables d'opérer dans un tel environnement.

4.2.4 Les algorithmes D^* et D^* Lite

D^* est un algorithme de recherche heuristique incrémentale introduit pour la première fois dans [Stentz, 1997]. Il s'agit d'un algorithme de recherche de chemin complet et optimal. Il est capable de trouver un ou plusieurs chemins dans un environnement partiellement inconnu ou dynamique. Il est capable de le faire en raison de sa capacité à réagir aux changements des coûts du trajet (apparition d'un nouvel obstacle par exemple), il met à jour son chemin trouvé en réutilisant les données précédemment établies. D^* ressemble beaucoup à A^* , sauf qu'il est dynamique, d'où le nom D^*.

Contrairement aux algorithmes basés sur A^*, D^* commence de la position finale vers la position de départ. En d'autres termes, il commence sa propagation d'informations au niveau du nœud de but et s'arrête lorsque il rencontre la position du robot ou le nœud de départ. Si aucun de ces nœuds n'est rencontré et la liste des nœud à traiter est vide, alors il n'existe pas de chemin entre ces deux positions.

Étant donnée la complexité à implémenter D^* , il a été remplacé par D^* *Lite* qui fonctionne selon le même principe en étant beaucoup plus simple à implémenter et à comprendre.

Tout comme D^*, l'algorithme D^* *Lite* [Koenig and Likhachev, 2002] fait une recherche heuristique incrémentale, il utilise les heuristiques pour concentrer sa recherche, et aussi les informations obtenues lors de recherches précédentes pour trouver une solution plus rapidement. D^* *Lite* a été construit sur les bases de LPA^* (Lifelong Planning A^*) [Koenig et al., 2004], un algorithme de recherche de chemin développé avant D^*. De plus, D^* *Lite* utilise la même stratégie de navigation que D^*

D^* *Lite* contient deux variables de coût pour chaque nœud n. Le premier est $g(n)$, qui est l'estimation du chemin du but jusqu'à n. Ce n'est qu'une estimation, car cette valeur peut changer si un chemin plus court est trouvé. Le second est $rhs(n)$, qui est une estimation à une étape avant. Cette estimation est basée sur la valeur de $g(n)$ et ses successeurs. Par conséquent, il est potentiellement mieux informé que $g(n)$.

$$rhs(n) = min_{n' \in Succ(n)}(g(n') + c(n, n')$$
$$c(n, n') : \textit{coût du déplacement de } n \textit{ à } n'$$

La valeur g et la valeur rhs déterminent si un nœud est consistant ou pas. Si $g(n) = rhs(n)$, alors le nœud est consistant, sinon il est inconsistant. De plus, si $g(n) > rhs(n)$, alors n est sous-consistant, alors que si $g(n) < rhs(n)$ alors n est sous-cohérent.

De la même façon que les autres algorithmes, D^* *Lite* utilise une queue à priorité *OpenList* pour stocker les nœuds à traiter, mais pas une *CloseListe*, donc on ne stocke pas les nœuds déjà visités. La liste *OpenList* contient seulement les nœuds inconsistants. L'ordre dans lequel les nœuds sont triés dans la liste ouverte est déterminé par leur valeur de clé k. Cette valeur est composée de deux clés la première est utilisée pour discriminer les nœuds, la second est utilisée comme discriminateur uniquement dans le cas ou deux nœud présentent la même valeur.

$$k(n) = [min(g(n), rhs(n)) + h(n_{start}, n) + k_m; min(g(n), rhs(n))]$$

Comme on peut le constater dans cette formule D^* *Lite* utilise l'heuristique $h(n_{start}, n')$ et un facteur k_m pour calculer la valeur de la clé. L'utilisation de $h(n_{start}, n')$ est comparable à l'utilisation d'une heuristique $h(n)$ pour calculer la valeur f dans A^*. Ici, l'heuristique $h(n_{start}, n')$ doit être positive et consistante.

Le facteur k_m est nécessaire pour éviter d'avoir à réorganiser la liste ouverte chaque fois que le nœud de départ change, cela peut arriver par exemple si le robot détecte une modification de coût après un mouvement. En mettant à jour le coût du chemin, l'algorithme considère l'emplacement actuel du robot comme le nœud de départ. En supposant que le robot soit déplacé d'un nœud n au nœud n', où s'il détecte une modification de coût, $h(n, n')$ est ajouté à k_m, qui à son tour est alors pris en compte lors du calcul de nouvelles valeurs de clés. Cela est nécessaire, car après le déplacement les premiers éléments des nœuds actuellement sur la liste *OpenListe*, peuvent avoir diminué de $h(n, n')$ dans le pire des cas. De même, lorsque de nouvelles valeurs de k sont calculées, leurs premiers éléments seraient $h(n, n')$ trop bas par rapport aux valeurs-clés déjà dans la liste *OpenListe*. Ceci est compensé par l'ajout de k_m à toutes les nouvelles valeurs de k.

L'algorithme $D*$ *Lite* est présenté en détail dans Algorithme 10.

4.3 Algorithmes inspirés des insectes (*bug algorithms*)

Les algorithmes inspirés d'insectes (en anglais *bug algorithms*) comme leurs noms l'indiquent ont une origine biologique. Ils se basent sur des techniques inspirées des déplacements des insectes. On trouve une description et une comparaison d'une large variété de ces algorithmes dans les articles suivants [Noborio et al., 2000] [Ng and Bräunl, 2007] [McGuire et al., 2019]. Le principe de ces algorithmes est qu'ils ne connaissent pas la position des obstacles dans leur environnement ni la position relative du but à atteindre. Ils ne réagiront localement qu'au contact d'obstacles et de murs qui constituent leur environnement immédiat, de manière à permettre au mobile de progresser vers son objectif en suivant les limites des obstacles et des murs, comme illustré dans la Figure 4.12. La nature de ces algorithmes est idéale pour la navigation en intérieur dont la carte de l'environnement n'est pas connue à l'avance et/ou la structure de l'environnement est en constant changement. Les algorithmes inspirés d'insectes se basent sur les critères suivants :

- Contrairement à plusieurs algorithmes de planification qui supposent une connaissance globale de l'environnement, ces algorithmes supposent seulement une connaissance locale de l'environnement et un objectif global.
- Leur comportements sont simples : (1) suivre les contours des obstacles, (2) se déplacer en ligne droite vers l'objectif.
- La portée des capteurs est limitée et admet une certaine plage d'incertitude de la position finale.

Algorithme 10 Calcule de chemin avec D* Lite

function KEY(s)
return $cle = [min(g(s), rhs(s)) + h(s_{start}, s) + k_m; min(g(s), rhs(s))]$
end function
function UPDATEVERTEX(s)
if $s \neq s_{goal}$ **then**
$rhs(s) \leftarrow min_{s' \in Succ(s)}(cost(s, s') + g(s'))$
end if
if $s \in OPEN$ **then**
$OPEN.remove(s)$
end if
if $g(s) \neq rhs(s)$ **then**
$OPEN.insert(s, Key(s))$
end ifreturn 1
end function
function COMPUTEPATH( )
while $True$ **do**
$OPEN.TopKey() < Key(S_{start})$ OR $rhs(S_{start} \neq g(S)$
$K_{old} = OPEN.Pop()$
if **then**$K_{old} < Key(s)$
$OPEN.insert(s, Key(s))$ $g(s) > rhs(s)$
$g(s) = rhs(s)$
for all $s' \in Pred(s)$ **do**
UPDATEVERTEX(s')
end for
else
$g(s) = \infty$
end if
for all $s' \in Pred(s)$ **do**
UPDATEVERTEX(s')
end for
end while
return 1
end function
function MAIN
for all s **do**
$rhs(s) = g(s) = \infty$
end for
$s_{last} = s_{start}$
$OPEN = \emptyset$
COMPUTEPATH( )
while $s_{last} \neq s_{start}$ **do**
$s_{start} = argmin_{s' \in Succ(S_{start})}(const(s_{start}, s') + g(s'))$
Move to s_{start}
Scan for cekk changes in environnement (e.g. sensor ranges)
if $Cell\_cahnges\_detected$ **then**
$k_m = k_m + h(s_{last}, s_{start})$
$s_{last} = s_{start}$
for all $s \in CHANGES$ **do**
Update cell s state
for all $s' \in Pred(s) \cup \{s\}$ **do**
UPDATEVERTEX(s')
end for
end for
COMPUTEPATH()
end if
end while
end function

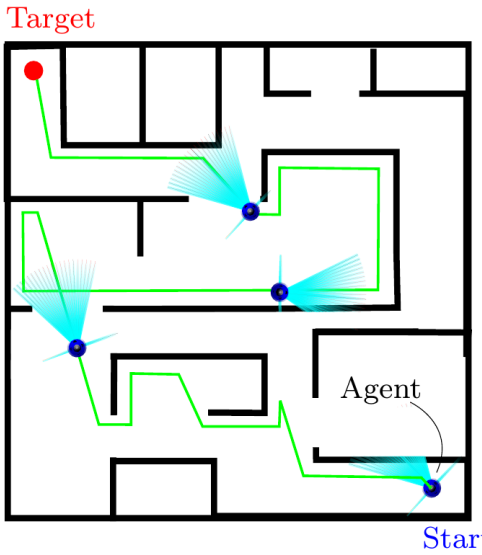


FIGURE 4.12 : Un exemple d'un mobile utilisant un algorithme *bug* dans un environnement intérieur. À partir d'une position de départ, il se déplace dans la direction du but, dans ce cas de figure, le mobile essaie de se déplacer vers le but chaque fois qu'il le peut, et suit le contour des obstacles lorsqu'il en rencontre un. (Cette Figure est extraite de l'article [McGuire et al., 2019]).

4.3.1 Algorithme Bug simples

Parmi les algorithmes les plus simples, on trouve les algorithmes **Com** [Lumelsky and Stepanov, 1986] ils ont été qualifiés par les auteurs de [McGuire et al., 2019] d'algorithmes « de bon sens » et ils les ont abrégés sous l'acronyme **Com** pour « Common sense algorithm ».

L'idée consiste à se déplacer en ligne directe à chaque fois que c'est possible, jusqu'à attendre le but, dans le cas où le mobile rencontre un obstacle alors il suit le conteur de l'obstacle jusqu'à ce qu'il est possible d'aller en ligne droite vers le but. Le premier point de contact avec l'obstacle est appelé **hit-point**, et le point où le mobile quitte le contour de l'obstacle pour aller en ligne droite vers le but est appelé **leave-point**. Cet algorithme peut résoudre plusieurs situations, mais les auteurs montrent quelques cas qui posent problème (voir Figure 4.13(a). Pour résoudre le problème posé par **Com** les auteurs propose l'algorithme **Bug1** qui propose d'explorer la totalité de l'obstacle tout en gardant en mémoire le dernier point **leave-point** comme le montre la Figure 4.13(b). Cependant **Bug1** propose généralement un chemin plus long. Pour cela, les auteurs proposent une version optimisée de cet algorithme, nommée **Bug2** dont l'idée consiste à dessiner une ligne imaginaire **M-line** entre la position de départ et le but. Tout comme **Bug1** on explore l'obstacle, mais cette fois-ci jusqu'à rencontrer la **M-line**, cette situation est illustrée dans la Figure 4.13(c). Une autre équipe de recherche [Sankaranarayanan and Vidyasagar, 1990] propose une amélioration de l'algorithme **Bug** sous le nom de **Bug1**, ils suggèrent de garder en mémoire en plus de **M-line** la distance du dernier point de contact, ainsi utiliser cette distance comme critère pour décider de ne plus suivre le contour de l'obstacle, cette amélioration est illustrée dans la Figure 4.13(b).

Dans le même article [Sankaranarayanan and Vidyasagar, 1990] les auteurs, indiquent qu'il existe encore des cas où **Bug2** proposent des chemins non optimisés et très long. Ils considèrent que la cause principale vient du fait que l'algorithme ne stocke pas les points visités précédemment lors du contour des obstacles. Pour cela, les auteurs proposent un algorithme basé sur **Bug2** sous le nom de **Alg1**, il a pour principe de changer la direction de suivi du contour si le mobile rencontre un **hit-point** déjà visité (voir 4.13(e)). Les auteurs développent aussi une autres version de cet algorithme sous le nom de **Alg2**, l'idée consiste à rajouter une **M-line**, cela permet dans certain cas d'optimiser les trajectoires (voir Figure 4.13(f)). Toujours dans le même article les auteur propose une nouvelle version de l'algorithme **Com** nommé **Com1**, l'idée consiste à mémoriser la distance du dernier point de repère **leave-point** par rapport à au but (voir Figure 4.13(f)).

DistBug est un algorithme développé par [Kamon and Rivlin, 1997], il est similaire à **Alg2** à la différence qu'il ne garde en mémoire que la dernière position du **hit-points**, ce qui le rend plus performent au niveau mémoire. Un autre aspect qui caractérise **DistBug**, qu'il n'impose pas une direction de contournement des obstacles cela se fait toujours dans la direction de la **M-line** comme illustré dans le Figure 4.13(g), dans certaines configurations cette stratégie échoue complètement.

Les auteurs [Horiuchi and Noborio, 2001] proposent deux algorithmes **Rev1** et **Rev2**, la stratégie consiste a alterner la direction à chaque **leave-point** pour suivre une trajectoire qui ressemble à un slalome. L'idée derrière cette stratégie, consiste à changer de direction si le mobile passe le même point, ce qui augmente les chances de trouver un chemin.

4.3.2 Algorithme Bug pour les mobiles capables de voir loin devant

Les algorithmes montrés précédemment, sont généralement adaptés pour des environnements tels que les labyrinthes, ils sont peu adaptés pour des robots mobiles capables de voir loin devant (sans toucher les obstacles). Car dans un environnement où il est possible de voir loin devant cette information peut-être exploitée pour améliorer sa trajectoire.

Nous avons décrit le fonctionnement de quelques algorithmes inspirés d'insectes, mais il en existe beaucoup d'autres, la Figure 4.14 montre les différents algorithme organisés par complexité (du plus simple en haut de la figure au plus complexes en bas).

À première vue, ces méthodes semblent fournir des méthodes de navigation efficace, idéales pour des implémentations sur de petits robots avec des ressources limitées. Une inspection plus approfondie, cependant, montre que bon nombre de ces algorithmes inspirés des insectes supposent une estimation de la position globale parfaite du robot, ce qui les rend très efficaces dans un environnement où les signaux GPS ne fonctionnent pas comme dans l'environnement domiciliaire. De plus, ils offrent une réelle alternative aux méthodes SLAM qui ne sont efficaces qu'après une exploration complète de l'environnement, donc pas fonctionnelles immédiatement.

4.4 Les limites de ces algorithmes pour la NAMO

Les techniques SLAM consistent à explorer l'environnement pour construire une carte, par la suite, on utilise cette carte pour calculer un chemin optimal entre deux positions. Dans le

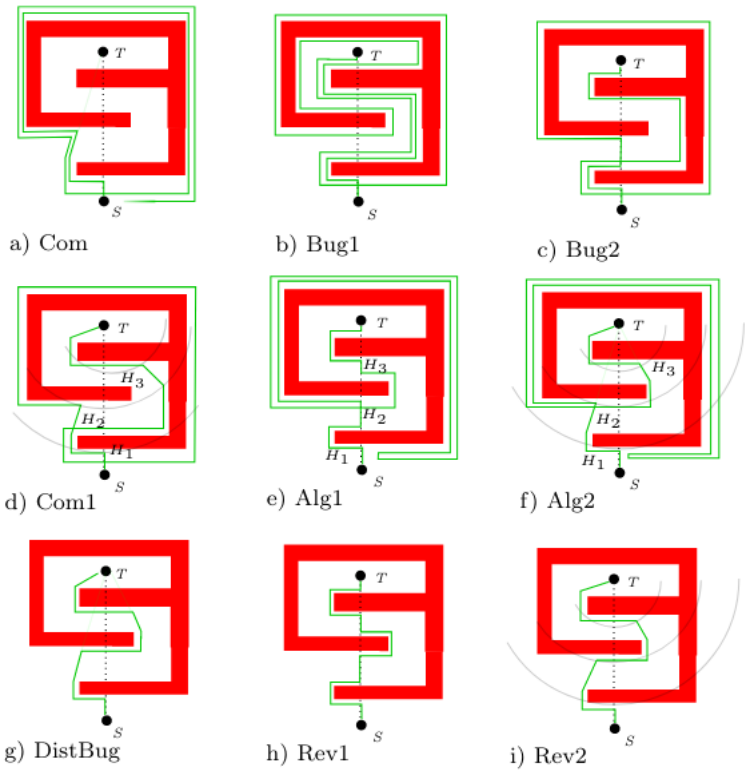


FIGURE 4.13 : Comportement des différents algorithmes de bogue. Ici, la configuration de l'environnement est choisie pour montrer des cas typiques, dans certains cas des algorithmes peuvent être favorisés par rapport à d'autres ou inversement. (Cette Figure est extraite de l'article [McGuire et al., 2019]).

cas d'un environnement avec beaucoup d'obstacles, engendre beaucoup de zones d'ombres[10], il est évident que la plupart des régions cartographiées ne sont pas utilisées lors du calcul d'un chemin optimal entre deux positions. Donc il est évident que cette technique explore inutilement certaines régions. De plus dans le cas d'un environnement dynamique, la carte construite devient vite obsolète, si les obstacles changent de place. Dans le cadre de la NAMO cette technique s'avère inadaptée dans le cas où certaines zones ne sont pas accessibles par évitement d'obstacles.

Dans le cas des algorithmes Bug, il est vrai que les mobiles sont immédiatement opérationnels et ne nécessitent pas une exploration préalables de l'environnement. Mais dans certaines configurations de l'environnement (en cause des minima locaux[11]) ces algorithmes peuvent aboutir à ces cycles (passer par le même chemin) car ils ne mémorisent pas la configuration de l'environnement déjà explorée. De plus, ces algorithmes ne proposent aucune amélioration de chemin dans le cas où le mobile est à refaire un même chemin déjà emprunté auparavant. De plus, ces algorithmes ne se basent pas sur des métriques ou des heuristiques, ce qui rend difficile

[10]Zones où il faut contourner l'obstacle pour cartographier l'environnement (derrières les obstacles, endroits inaccessibles par évitement d'obstacles, etc.)

[11]Le problème des minima locaux survient lorsqu'un mobile naviguant sans connaissance a priori de l'environnement est piégé dans une boucle. Cela se produit surtout si l'environnement se compose d'obstacles concaves, de labyrinthes, etc. Pour sortir de la boucle, le robot doit comprendre sa traversée répétée à travers le même environnement, ce qui implique de mémoriser l'environnement déjà exploré.

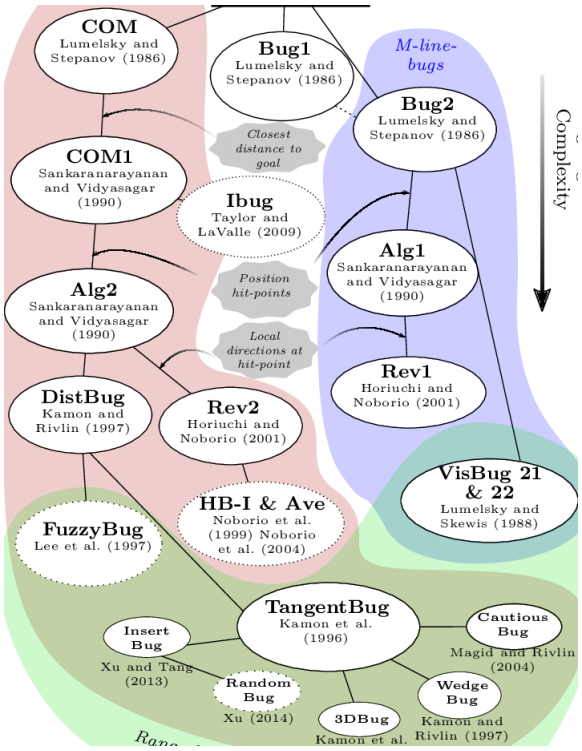


FIGURE 4.14 : Vue globale des algorithmes inspirés d'insectes, ils sont classés de haut en bas selon leur complexité. En bleu on trouve les algorithmes qui se repose sue une **M-Line** pour prendre des décisions, en rouge les algorithmes qui n'utilisent pas. En vert on trouve les algorithmes plus complexes, qui se reposent sur des donnés capteurs pour regarder loin devant. (Cette Figure est extraite de l'article [McGuire et al., 2019]).

l'utilisation d'algorithmes de recherche de chemins tel que $D*$ ou autres.

Un robot mobile avec évitement d'obstacle doit être doté d'un planificateur lui permettant d'éviter des situations où il doit rebrousser chemin. Dans le cas où le mobile dispose d'un planificateur local capable de bouger les obstacles, il n'est pas préoccupant de disposer d'un planificateur global qui conduit à situations de blocage tant que le planificateur local débloque la situation. Il est même important dans ce cas d'utiliser un planificateur permettant de déterminer le chemin le plus court possible en prenant en compte les capacités de gestion des obstacles du mobile. Or, les algorithmes Bug essayent de trouver des trajectoires en ligne droite vers l'objectif, mais ne mémorisent pas les endroits déjà explorés.

Nous pensons qu'il est important d'utiliser dans notre cas un planificateur qui puisse bénéficier des deux techniques, à savoir se déplacer efficacement vers l'objectif et repérer les situations de blocage potentiellement solvables par le planificateur local, mais aussi sauvegarder les traits de l'environnement afin d'améliorer l'efficacité de navigation dans le temps.

4.5 Un planificateur globale pour la NAMO

Dans cette section, nous allons proposer une méthode de navigation en environnement inconnu adaptée au milieu domiciliaire. Cette méthode est inspirée des *Bug Algortithms (BAs)* en rajoutant une représentation à base de graphe et des heuristiques sur les distances. Ce graphe a pour but de donner des indications sur les zones déjà explorées, ou il est inutile d'explorer. La recherche de chemin se fera au fur et à mesure que le mobile explore son environnement.

En navigation intérieure avec des obstacles amovibles (NAMO), nous pensons qu'il est essentiel d'avoir une méthode qui n'utilise pas de carte préalable, car les obstacles sont nombreux et n'occupent pas une position fixe dans l'environnement et certains obstacles sont dynamiques (personnes, animaux de compagnie, etc.). Ce qui élimine la première catégorie d'algorithme que nous avons présenté dans ce chapitre[12].

Quant aux méthodes SLAM, elles montrent de très grandes performances en environnement inconnu, ces méthodes cartographies l'environnement afin d'avoir une carte de l'environnement puis à partir de ces cartes, on utilise par exemple les roadmap, RRT, etc. Pour construire un graphe et par la suite trouver un chemin dans ce graphe. Malheureusement, ces méthodes ne fonctionnent pas en environnement NAMO avec une forte présence d'obstacles dynamiques, car elles nécessitent la reconstruction régulière de carte (ou une partie de cartes), ce qui est énergivore et chronophage.

Les Bug Algorithms (BAs) ont été parmi les premiers algorithmes à êtres développés en robotique leurs apparitions remonte aux années 70. Leurs avantages résident dans le fait qu'ils consomment peu d'énergie et ne nécessitent pas d'une grande puissance de calcul ce qui était un souci majeur à l'époque. Depuis les six dernières années nous avons remarqué de nouveau un réel intérêt de la communauté robotique à ces algorithmes, car ils ont la capacité de réagir rapidement au changement de la configuration de l'environnement ce qui est essentiel pour un environnement NAMO. Ces algorithmes peuvent avoir un réel apport si on combine leur simplicité de l'époque avec les capteurs et la puissance de calcul de nos jours. Effectivement, leur application dans un environnement congestionné nécessite des améliorations et n'est pas applicable dans l'état, tel qu'ils sont présentés dans la section précédente.

Le planificateur global pour la NAMO a pour but de rapprocher le robot de son objectif en évitant les obstacles, si un obstacle bloque le passage (obstacle qui ne peut être / ou coûte cher à écarter par le planificateur local) alors il faut que cet algorithme puisse recalculer un chemin rapidement, en évitant les chemins explorés précédemment.

Comme nous l'avons brièvement évoqué dans le chapitre précèdent, l'approche que nous proposons repose sur le but assigné au robot. Le robot reçoit comme instruction un cap à suivre et un objet à trouver (Figure 3.34 (a)). La localisation du robot est relative à son but, donc il peut déterminer s'il s'éloigne ou se rapproche de son objectif. Un second paramètre facultatif qui consiste à indiquer au robot la distance à laquelle se trouve l'objet, peut être indiqué pour le robot pour faciliter la tâche. Dans le cas où ce paramètre n'est pas indiqué alors

[12]Ça ne veut pas dire que ces méthodes ne sont pas utilisées en environnement inconnu, mais elles sont utilisées en complément d'autres méthodes. On voit souvent l'utilisation de RRT avec SLAM par exemple.

le robot suppose que l'objectif est le plus proche possible, mais hors d'atteinte de son champ de vision.

L'algorithme présenté ici, permet d'atteindre un objectif en environnement inconnu, il reçoit une indication de direction et fixe un sous-objectif lui permettant d'avancer dans les zones connues (visible ou déjà visitées). Une illustration de ce processus est présentée dans la Figure 4.15.

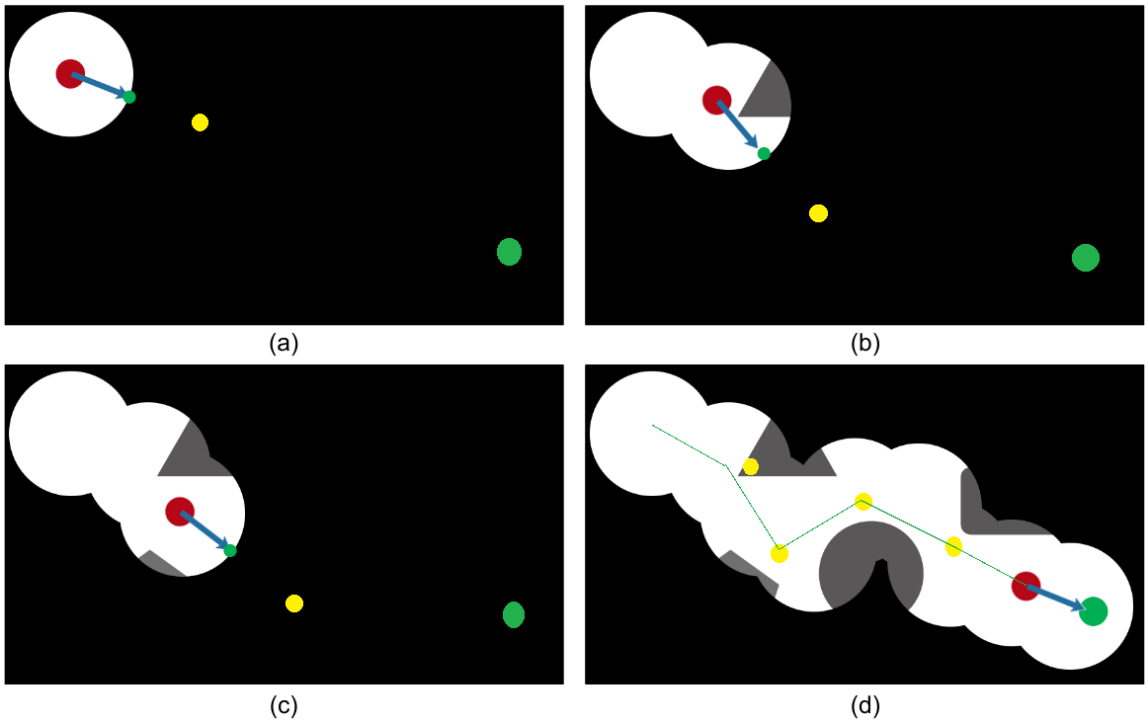


FIGURE 4.15 : Les différentes Figures montrent le robot en point rouge, et les sous-objectifs fixés avec un petit point vert. (a) Montre la commande de l'utilisateur qui indique une direction. (b) et (c) montre l'avancement du robot l'adaptation de la trajectoire, avec de nouveaux sous-objectifs. (d) Le robot atteint l'objectif, matérialisé par un point vert plus grand.

Il est important de souligner que cet algorithme doit fonctionner en environnement inconnu, car comme nous l'avons mentionné précédemment, dans un environnement congestionné avec des obstacles dynamiques, il est inutile de connaître la position de ces obstacles vu leur nature mouvante.

Pour cela, nous proposons l'algorithme **HeadStar** (H^*) qui permet au robot de trouver un chemin plus au moins optimal tout en explorant son environnement. Cet algorithme est décrit ci-après.

4.6 L'agorithme HeadStar

L'algorithme **HeadStar** (H^*), est à mi-chemin entres les algorithmes bug et les algorithmes de recherche de chemins. Il fonctionne en environnement inconnu. L'algorithme utilise unique-

ment une seule heuristique[13] pour se diriger $f(x) = h$. À l'instar des algorithmes bug, H^* ne dispose pas préalablement d'une carte de l'environnement, pour explorer son environnement, il doit se déplacer. Le mobile dispose d'un capteur d'une certaine protée tel que c'est représenté dans la Figure 4.16. Tout comme les algorithmes de recherche de chemin classiques, H^* utilise des heuristiques pour optimiser les trajectoires de navigation et éviter des explorations inutiles. L'algorithme H^* dans sa version classique utilise une division cellulaire la portée du capteur est montrée dans la Figure 4.16 avec les cases blanches les cases grisées montrent les zones non explorées, les obstacles sont représentés avec des cases hachurées. L'heuristique est calculée par rapport à la case but.

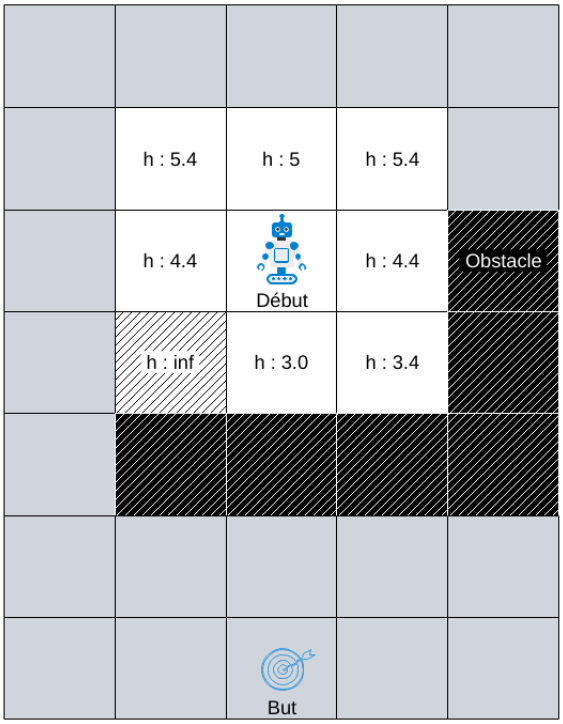


FIGURE 4.16 : Environnement de H^*. Les cases grisées représentent les zones libres non explorées, en blanc sont représentées les zones explorées (à la portée du capteur du mobile), les cases hachurées représentent les obstacles.

Pour éviter que l'algorithme de s'étaler en largeur et le forcer à aller de l'avant ou à suivre les contours, nous avons introduit la notion d'*inconsistance*. Une celle marquée inconsistante ne sera pas explorée tant qu'il reste des cellules dans la liste ouvert. Le choix des cellules inconsistantes est fait lors du déplacement d'une cellule à une autre. En horizontal et en vertical en marque les cellules adjacentes de la case de départ. En déplacement diagonal, on marque les cellules adjacentes à la transition. Les différents cas, sont illustrés dans la Figure 4.17.

H^* utilise des files de priorité pour stocker les nœuds à visiter et une pile pour stocker les nœuds déjà visités. Les files de priorité, sont ordonnées selon $f(x) = h$, si deux éléments possèdent la même valeur alors on utilise g comme discriminant $f(x) = h+g$. h est l'estimation

[13]Dans ce rapport l'heuristique est calculée en utilisant la distance euclidienne, d'autres méthodes de calcul sont possibles comme par exemple la distance de Manhattan, la distance de Tchebychev, etc.

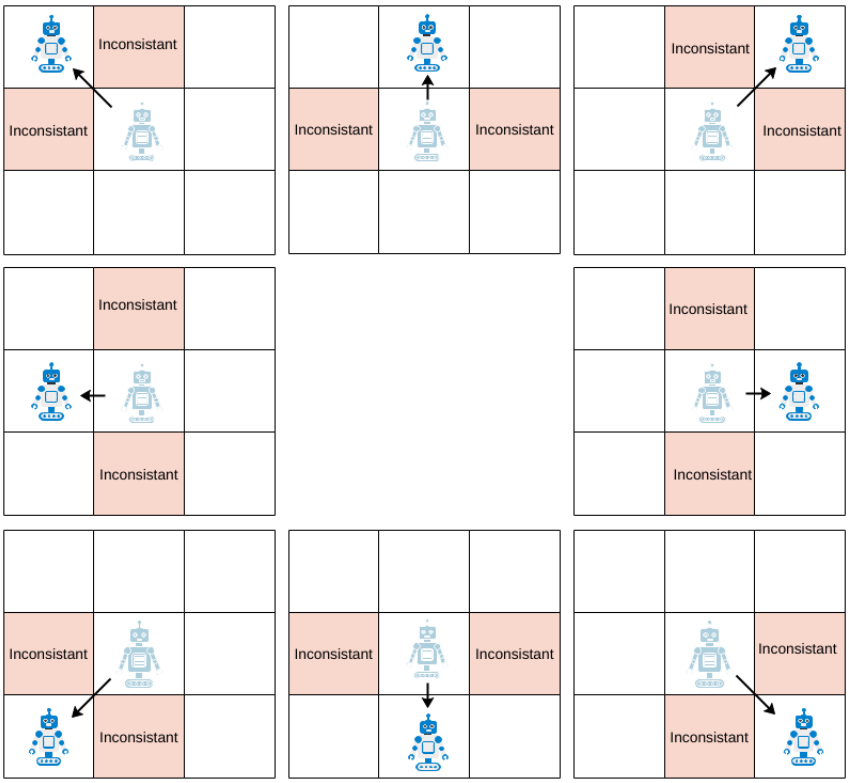


FIGURE 4.17 : Comment déterminer les cases inconsistantes.

de la distance entre cette position et le *but*, g est la distance entre le mobile et cette position. Les différentes listes utilisées par l'algorithme sont les suivantes :

- **Liste ouvert** : file de priorité (ordonnée selon h, $g + h$) stock les sommets nouvellement découverts.
- **Liste inconsistant** : file de priorité (ordonnée selon h, $g + h$) stock les sommets inconsistants.
- **Liste fermé** : pile qui stock les sommets déjà visités.

À partir de la position de départ, H^* insère la case de départ dans la liste ouvert pour initier le processus. Tant que la *liste ouvert* et la *liste inconsistant* ne sont pas vides, on répète le processus suivant : On calcule les heuristiques des cases adjacentes et on les insère dans la liste à priorité *liste ouvert*. On retire le premier élément de la *liste ouvert*, si cette liste est vide alors on retire la première position dans la *liste inconsistant*, et on se déplace vers cette position si le déplacement est direct (une seule case de déplacement), sinon, on rappel l'algorithme récursivement (les listes sont définies dans la pile, lors de l'appel récursif, les listes sont vidées, et retrouvent leur contenu lors de dépilement). Lors du déplacement (dans le cas où le déplacement est d'une seule case) on insère la position visitée dans la *Liste fermé* et on calcule les positions inconsistantes et on les insère dans la *Liste inconsistant*, à chaque fois en prenant le soin de retirer les positions dans leurs listes de départs. L'algorithme se termine une fois les deux liste *liste ouvert* et *liste inconsistant* sont vides. Cet algorithme est décrit dans l'Algorithme 11.

Algorithme 11 Algorithme HeadStar

```
1: function Main
2:     path = ∅
3:     s_start = read()
4:     s_goal = read()
5:     ComputePath(s_start, s_goal, path)
6:     print(path)
7: end function
```

```
1: function ComputePath(S_start, S_goal, Path)
2:     OPEN ← ∅
3:     INCONSISTANCE ← ∅
4:     CLOSE ← ∅
5:     s ← S_start
6:     OPEN.insert(s)
7:     while OPEN.size() ∉ ∅ or INCONSISTANCE.size() ∉ ∅ do
8:         AddToCloseList(s, OPEN, INCONSISTANCE, CLOSE)
9:         Path.insert(s)
10:        if sS_goal then
11:            return True                                  ▷ Chemin trouvé
12:        end if
13:        for all s' ∈GetNeighbors(s) do
14:            AddToOpenList(s, OPEN, INCONSISTANCE, CLOSE)
15:        end for
16:        next_pos ← FindMinCell(OPEN, INCONSISTANCE, S_goal)
17:        if next_pos∅ then
18:            return False                                 ▷ Pas de chemin possible
19:        end if
20:        if |next_pos − s| ≤ √2 then
21:            inc_list ←GetInconsistant(s, next_pos)
22:            for each s' ∈ inc_list do
23:                AddToInconsistantList(s, OPEN, INCONSISTANCE, CLOSE)
24:            end for
25:            s ← next_pos
26:        else
27:            if ComputePath(s, next_pos, Path) then
28:                s ← next_pos
29:            end if
30:        end if
31:    end while
32:    return
33: end function
```

4.6.1 Illustration du fonctionnement de H^*

La Figure 4.18 montre le fonctionnement de H^*. On commence par ajouter la position de départ (D2) dans la *liste ouvert*, on calcule la distance euclidienne vers l'objectif sur les cellules

```
1: function GetNeighbors(s)
2:     neighbors ← ∅
3:     for all s' ∈ S do
4:         if |s' − s| ≤ √2 then
5:             neighbors.insert(s')
6:         end if
7:     end for
8:     return neighbors
9: end function
```

```
1: function FindMinCell(OPEN, INCONSISTANCE, S_goal)
2:     s' ← ∅
3:     if OPEN.size() > 0 then
4:         s_min ← OPEN[0]
5:         for each s' ∈ OPEN do
6:             if |s_min − S_goal| > |s' − S_goal| then
7:                 s_min ← s'
8:             end if
9:         end for
10:        return s_min
11:    end if
12:    if INCONSISTANCE.size() > 0 then
13:        s_min ← INCONSISTANCE[0]
14:        for each s' ∈ OPEN do
15:            if |s_min − S_goal| > |s' − S_goal| then
16:                s_min ← s'
17:            end if
18:        end for
19:        return s_min
20:    end if
21:    return ∅
22: end function
```

```
1: function AddToCloseList(s', OPEN, INCONSISTANCE, CLOSE)
2:     CLOSE.insert(s')
3:     if s ∈ OPEN then
4:         OPEN.remove(s')
5:     end if
6:     if s ∈ INCONSISTANCE then
7:         INCONSISTANCE.remove(s')
8:     end if
9: end function
```

adjacentes, on attribue une valeur infinie aux cellules infranchissables et on les ajoute dans la *liste ouvert* (uniquement les cellules avec $h \neq \infty$). On se déplace dans la cellule avec un h minimum (E2) et noter les cellules (D1 et D3) comme inconsistantes (ajouter ces deux cellules dans *liste inconsistant* et les retirer de la *liste ouvert*). Lorsque on se déplace dans une cellule,

```
1: function AddToInconsistantList(s', OPEN, INCONSISTANCE, CLOSE)
2:     if s' ∈ CLOSE then
3:         return
4:     end if
5:     if s' ∈ OPEN then
6:         OPEN.remove(s')
7:     end if
8:     if s' ∉ INCONSISTANCE then
9:         INCONSISTANCE.insert(s')
10:    end if
11: end function
```

```
1: function AddToOpenList(s', OPEN, INCONSISTANCE, CLOSE)
2:     if s' ∈ INCONSISTANCE Or s' ∈ CLOSE then
3:         return
4:     end if
5:     if s' ∉ OPEN then
6:         OPEN.insert(s')
7:     end if
8: end function
```

on la retire des deux listes (*liste ouvert* et *liste inconsistante*) et la mettre dans *liste fermé*. La prochaine cellule avec h minimum dans la *liste ouvert* est (E3), donc on se déplace vers cette case directement (dans ce cas l'algorithme fait un appel récursif, et se relance comme position de départ (E3) et position d'arrivée C3). Tant que la *liste ouvert* n'est pas vide, on poursuit le même processus. Dans le cas la *liste ouvert* est vide alors on regarde s'il reste des éléments dans *liste inconsistante*.

4.6.2 Résultats

Ici, nous allons comparer l'algorithme H^* aux algorithmes $D*Lite$ et $A*$. Les résultats sont obtenus statistiquement, en utilisant 3000×26 essais. Le nombre d'obstacles commence à 0 pour atteindre 300 obstacles sur une surface de 20×20. Le nombre d'obstacles est augmenté progressivement par tranches de 10 chaque 3000 essais. Au-delà de 300 obstacles sur une surface de 20×20, soit une densité de 0.75 il devient difficile de trouver un chemin entre la position de départ et la position d'arrivée.

Le graphique dans la Figure 4.19 (B) montre une comparaison des trois algorithmes en terme de temps d'exécution. On remarque que $D*Lite$ est chronophage lorsque le nombre d'obstacles est faible, cela est dû au fait que beaucoup de nœuds se retrouvent dans la *open liste* (comme on peu le voir dans 4.19 (D)), et le calcule de clé est basée sur deux critères (dans l'implémentation utilisée ici, la *open liste* est implémenté dans un dictionnaire, la clé étant une paire d'entiers indiquant l'abscisse et l'ordonnée de la cellule, quant à la valeur, elle est composée d'un paire de doubles $<min(g(s), rhs(s)) + h(s_{start}, s)\ ,\ min(g(s), rhs(s)) >$. L'obtention de la cellule avec une valeur minimale dans la *open list* implémenté dans la fonction `TopKey(open_list)` en utilisant la fonction sort de la **STL** (voir Listing 4.1)). Pour les deux algorithmes restant, ils occupent un temps linière malgré que H^* est un peu plus performant que $A*$. Encore une

```
1: function GetInconsistant(s', s'')
2:     list_inconsist ← ∅
3:     dx ← s''.x − s'.x
4:     dy ← s''.y − s'.y
5:     s_tmp = (null, null)
6:     if (dx = 0 and dy = −1) or (dx = 0 and dy = +1) then
7:         list_inconsist.insert(s_tmp(s'.x − 1, s'.y))
8:         list_inconsist.insert(s_tmp(s'.x + 1, s'.y))
9:     end if
10:    if (dx = −1 and dy = 0) or (dx = +1 and dy = 0) then
11:        list_inconsist.insert(s_tmp(s'.x, s'.y − 1))
12:        list_inconsist.insert(s_tmp(s'.x, s'.y + 1))
13:    end if
14:    if dx = −1 and dy = −1 then
15:        list_inconsist.insert(s_tmp(s'.x − 1, s'.y))
16:        list_inconsist.insert(s_tmp(s'.x, s'.y − 1))
17:    end if
18:    if dx = 1 and dy = 1 then
19:        list_inconsist.insert(s_tmp(s'.x + 1, s'.y))
20:        list_inconsist.insert(s_tmp(s'.x, s'.y + 1))
21:    end if
22:    if dx = 1 and dy = −1 then
23:        list_inconsist.insert(s_tmp(s'.x, s'.y − 1))
24:        list_inconsist.insert(s_tmp(s'.x + 1, s'.y))
25:    end if
26:    if dx = −1 and dy = 1 then
27:        list_inconsist.insert(s_tmp(s'.x − 1, s'.y))
28:        list_inconsist.insert(s_tmp(s'.x, s'.y + 1))
29:    end if
30: end function
```

fois, cela est dû au nombre de nœuds qui se retrouvent dans *open list*, elle est plus importante dans $A*$.

Listing 4.1 : Cellule avec la plus faible clé

```
1  bool comparison(pair<pair<int,int>,pair<double,double>> x1,
2                  pair<pair<int,int>,pair<double,double>> x2){
3      return (x1.second.first  <= x2.second.first) &&
4             (x1.second.second <= x2.second.second);
5  }
6
7  pair<double, double> D_Light_Star::TopKey(op_lst &ls_open){
8      ls_open.sort(comparison);
9      return ls_open.front().second;
10 }
```

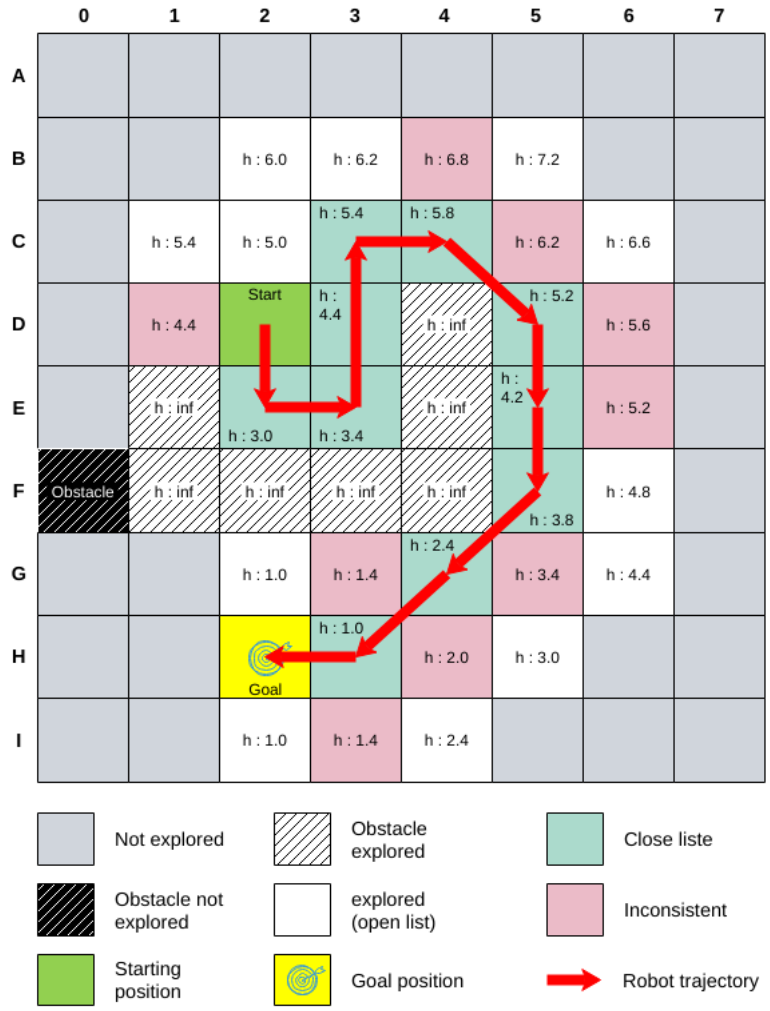


FIGURE 4.18 : Fonctionnement de l'algorithme H^*.

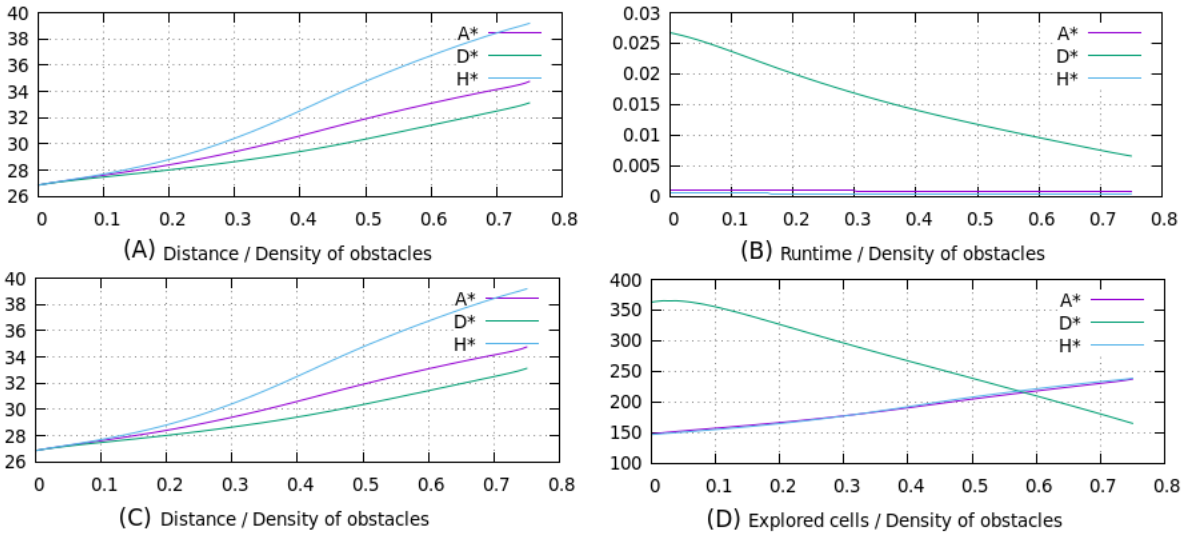


FIGURE 4.19 : Comparaison des distances, nombre de nœuds et temps d'exécution en seconde, entre les algorithmes H^*, $D*Lite$ et $A*$.

Nous souhaitons souligner que H^* doit se déplacer pour explorer son environnement, contrairement aux algorithmes avec lesquels nous l'avons comparé. Pour effectuer une comparaison équitable, nous avons affecté des tests en supposant que $D * Lite$ n'as aucune connaissance préalable de son environnement. Techniquement cette opération est réalisée en faisant la génération des obstacles après la fonction `ComputePath()` (voir Algorithme 10). Le graphique 4.19 montre des performances équivalentes – en terme de distance parcourue– (légèrement en faveur de H^*). Mais $D * Lite$ n'est pas capable de travailler dans un environnement où la position finale n'est pas connue à l'avance ou cette dernière est mouvante.

Le Tableau 4.6.2 montre une comparaison de H^* avec d'autres algorithmes A^*, D^*Lite et RRT. Les résultats sont obtenus avec l'application $MPAL$ (pour *Motion Planning ALgorithms*) que nous avons réalisé (voir Figure 4.20). Le code source de cette application est disponible sur l'url suivante : https://github.com/hdd-robot/MPAL.git. Pour générer les résultats du tableau 4.6.2 nous avons utilisé une version de MPAL permettant de générer les obstacles et de lancer les tests automatiquement, cette version est disponible sur l'url suivante : https://github.com/hdd-robot/head_star_compare.git

N-obs.	D-Obs.	Densité	Dst. A*	Dst. D*	Dst. H*	R.T. A*	R.T. D*	R.T. H*	E.C. A*	E.C. D*	E.C. H*
0	26.87	0.000	26.87	26.87	26.87	0.00106	0.02672	0.00053	148	363	147
10	26.87	0.025	27.09	27.08	27.10	0.00106	0.02621	0.00053	151	368	149
20	26.87	0.050	27.27	27.24	27.29	0.00105	0.02551	0.00052	153	367	151
30	26.87	0.075	27.44	27.36	27.47	0.00104	0.02469	0.00052	155	363	153
40	26.87	0.100	27.61	27.49	27.67	0.00102	0.02373	0.00051	157	357	155
50	26.87	0.125	27.79	27.62	27.91	0.00101	0.02274	0.00050	159	350	157
60	26.87	0.150	27.98	27.74	28.18	0.00099	0.02180	0.00050	162	342	160
70	26.87	0.175	28.17	27.88	28.42	0.00098	0.02088	0.00049	164	335	162
80	26.87	0.200	28.37	28.01	28.70	0.00097	0.01994	0.00048	166	327	164
90	26.87	0.225	28.59	28.14	29.07	0.00095	0.01907	0.00048	169	319	167
100	26.87	0.250	28.84	28.32	29.49	0.00095	0.01824	0.00048	172	311	170
110	26.87	0.275	29.06	28.44	29.82	0.00095	0.01767	0.00048	174	303	173
120	26.87	0.300	29.40	28.64	30.38	0.00093	0.01674	0.00047	177	296	177
130	26.87	0.325	29.69	28.82	30.80	0.00092	0.01601	0.00047	180	289	180
140	26.87	0.350	29.91	28.98	31.36	0.00091	0.01533	0.00047	183	281	184
150	26.87	0.375	30.25	29.18	31.92	0.00091	0.01468	0.00047	187	274	188
160	26.87	0.400	30.57	29.37	32.39	0.00090	0.01407	0.00046	190	267	191
170	26.87	0.425	30.98	29.62	33.15	0.00090	0.01341	0.00047	195	260	197
180	26.87	0.450	31.26	29.84	33.67	0.00089	0.01285	0.00046	197	253	201
190	26.87	0.475	31.67	30.07	34.35	0.00089	0.01227	0.00047	202	246	205
200	26.87	0.500	32.00	30.35	34.95	0.00088	0.01170	0.00046	206	239	209
210	26.87	0.525	32.28	30.62	35.47	0.00087	0.01117	0.00046	209	232	213
270	26.87	0.675	34.05	32.25	38.26	0.00085	0.00800	0.00045	228	189	232
280	26.87	0.700	34.23	32.54	38.54	0.00083	0.00751	0.00045	231	181	234
290	26.87	0.725	34.24	32.69	38.78	0.00082	0.00704	0.00044	232	173	235
300	26.87	0.750	34.76	33.13	39.19	0.00082	0.00661	0.00044	238	165	239
Moy.	-	-	**30.01**	**29.16**	**31.58**	**0.00094**	**0.01672**	**0,00048**	**184**	**287**	**184**

Tableau 4.1 : N-Obs. = Nombre d'obstacles ; D-Obs. = Distance euclidienne ; Dst. = Distance parcourue ; R.T. = Temps d'exécution en seconde ; E.C. = Nombre de nœuds (cellules) explorés ;

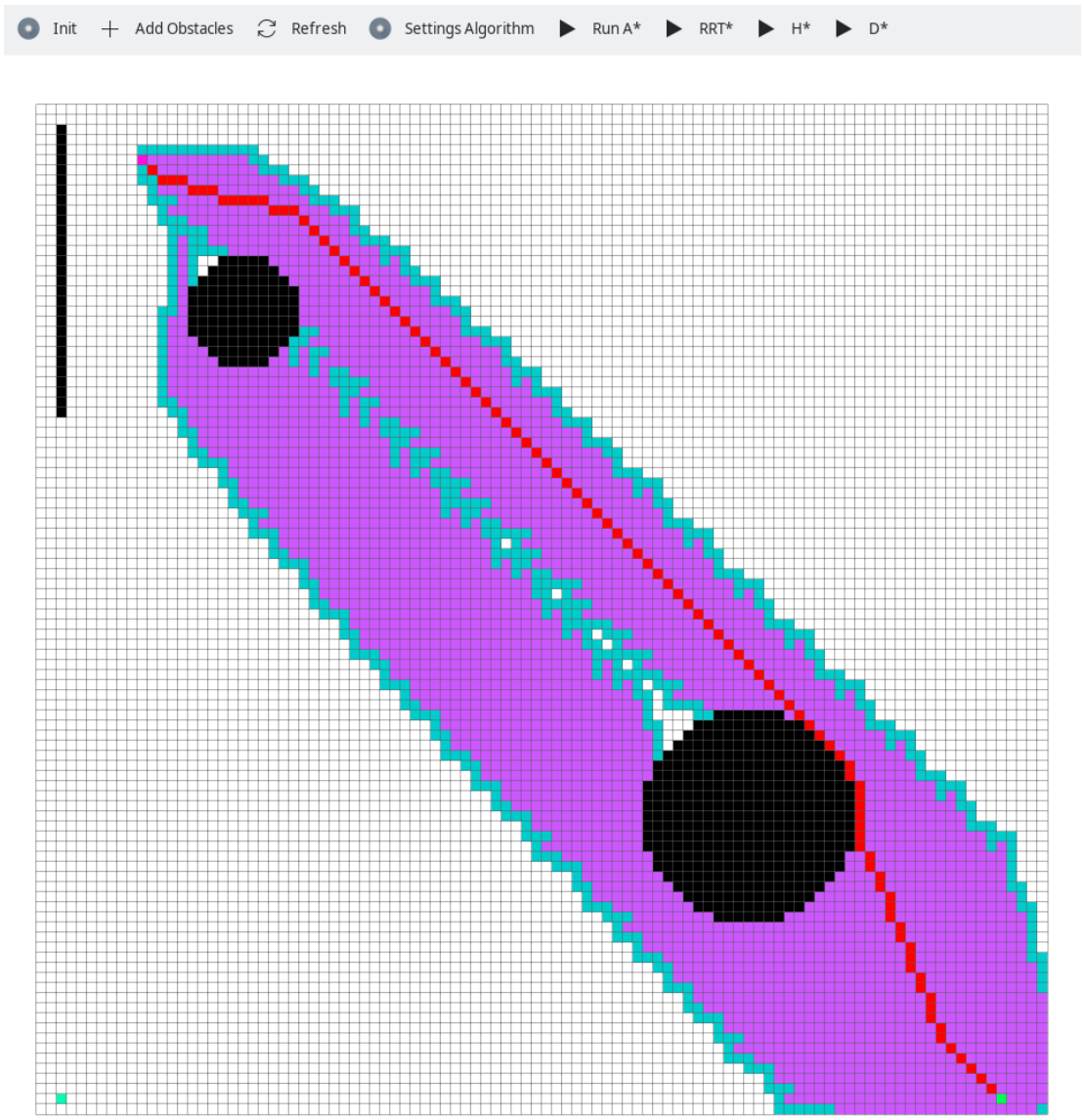


FIGURE 4.20 : Le logiciel MPAL qui permet simuler et de comparer les résultats de différents algorithmes A^*, D^* *Lite* et *RRT* avec H^*. Il permet aussi à l'utilisateur de paramétrer l'environnement en ajoutant des obstacles. Les algorithmes aussi sont paramétrables, par exemple on peut spécifier le type d'heuristique à utiliser.

L'image 4.21 montre l'utilisation de H^* avec le robot *TurtleBot3* avec le simulateur *Gazebo* et *RViz*. L'image (A) montre l'environnement du robot dans le simulateur Gazebo. La position de départ du robot et la position d'arrivée sont montrées avec les cercles rouges. Le robot cartographie son environnement à l'aide d'un LIDAR. Les figures de (1) à (8) son générées par *RViz*, elles montrent les différentes étapes par lesquelles le robot passe pour atteindre son objectif. Les zones en gris foncé montrent les zones non reconnues et en gris les espaces libres, les obstacles sont représentés par des zones noires. La position du robot est toujours indiquée avec le cercle rouge. La Figure (B) montre la trajectoire du Robot de sa position de départ à sa position d'arrivée.

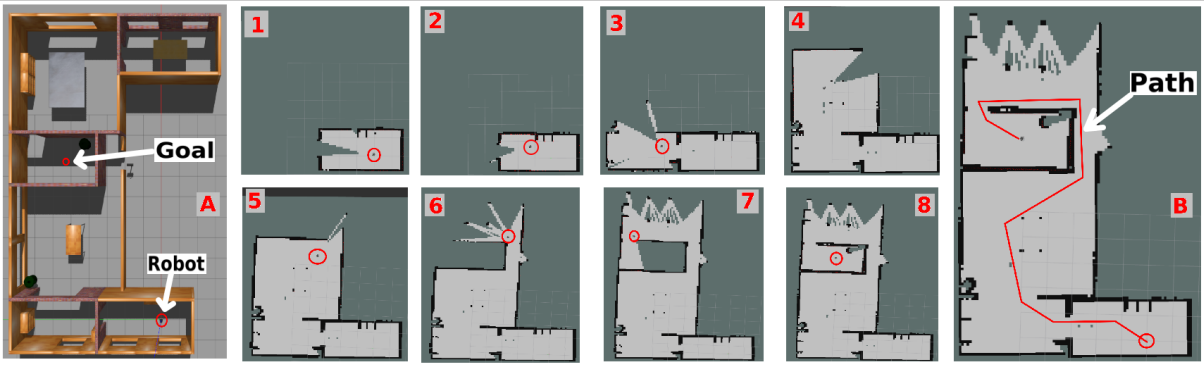


FIGURE 4.21 : Simulation de H^* avec le robot TurtleBot 3 dans *Gazebo*.

4.7 Extension de H^* en environnement partiellement connu

Nous avons montré précédemment que les algorithmes en environnements partiellement connus (par exemple $D*$) sont difficilement utilisables en environnement congestionné, dans le cas de [14], en raison de la multiplicité des obstacles [Moghaddam and Masehian, 2016]. Pour cela, nous avons conçu l'algorithme H^*. Cet algorithme fonctionne dans un environnement inconnu, or en environnement domiciliaire certains obstacles ne peuvent changer de place comme par exemples les murs, et les configurations générales (les plans) sont généralement connues à l'avance. Étant donné la nature *on-line* de l'algorithme (se déplace au même temps qu'il découvre son environnement), effectivement ces informations peuvent éviter dans certaines configurations des trajectoires inutiles. La Figure 4.22 montre l'algorithme H^* dans un environnement structuré sans obstacles, on remarque que la trajectoire n'est pas optimale. Pour résoudre ce problème et améliorer les performances de H^* on peut proposer d'utiliser les informations du plan global de l'environnement pour orienter l'algorithme et tendre vers des trajectoires tel que c'est illustré dans la Figure 4.23.

La méthode que nous proposons, consiste à diviser l'environnement connu en zones (voir Figure 4.24) afin de construire un graphe. Chaque nœud du graphe (voir Figure 4.25) représentera alors les sous-objectifs pour le robot. Dans l'exemple de la figure suivante, le mobile qui se trouve dans la zone (1) à pour objectif de rejoindre la zone (9). À l'aide du graphe (voir Figure 4.25) on peut identifier deux chemins[15] possibles $\{1, 4, 5, 9\}$ ou $\{1, 2, 5, 9\}$. Donc, pour rejoindre l'objectif (9), il fait choisir un chemin (par exemple $\{1, 4, 5, 9\}$) et entre chaque nœud $\{\{1, 4\}, \{4, 5\}, \{5, 9\}\}$ on applique H^*.

Pour construire le graphe (Figure 4.25), on peut proposer par exemple d'utiliser le diagramme de Voronoï. Dans l'exemple précèdent le diagramme de Voronoï abouti à la création de zones montrées dans Figure 4.26, il reste à connecter chaque centre de deux zones adjacentes à l'aide d'arêtes.

[14] D^*, il doit connaître exactement le point d'arrivée au préalable, de plus pour qu'il fonctionne d'une façon optimale il doit connaître une partie de l'environnement au préalable.

[15] Dans ce travail nous avons utilisé l'algorithme Dijkstra pour trouver le chemin le plus court. D'autres méthodes sont possibles.

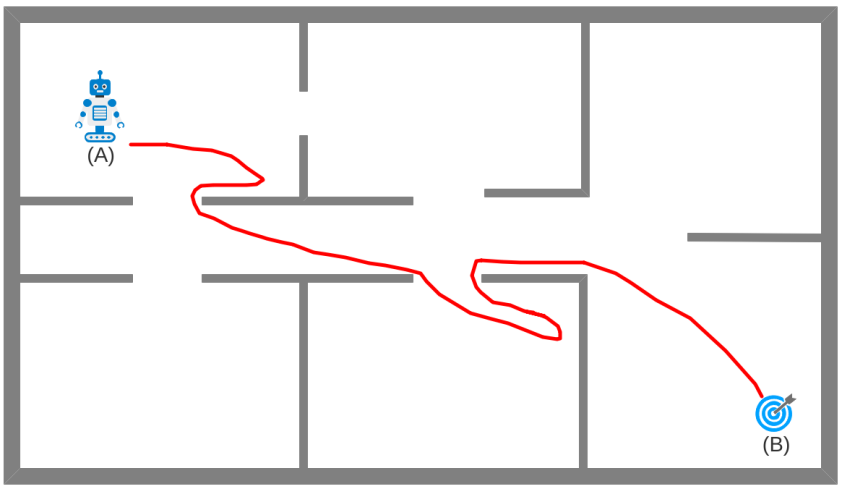


FIGURE 4.22 : Trajectoire du mobile en utilisant l'algorithme H^*, sans utiliser les informations fournies par le plan.

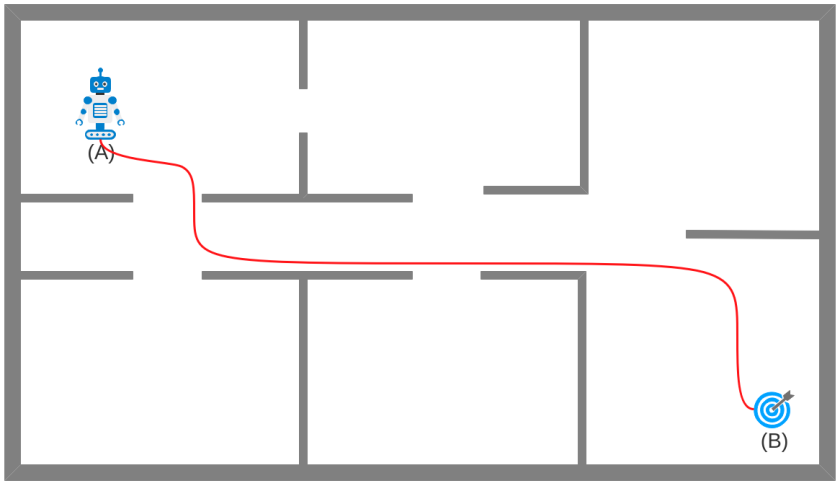


FIGURE 4.23 : Trajectoire du mobile en utilisant l'algorithme H^*, en utilisant les informations fournies par le plan.

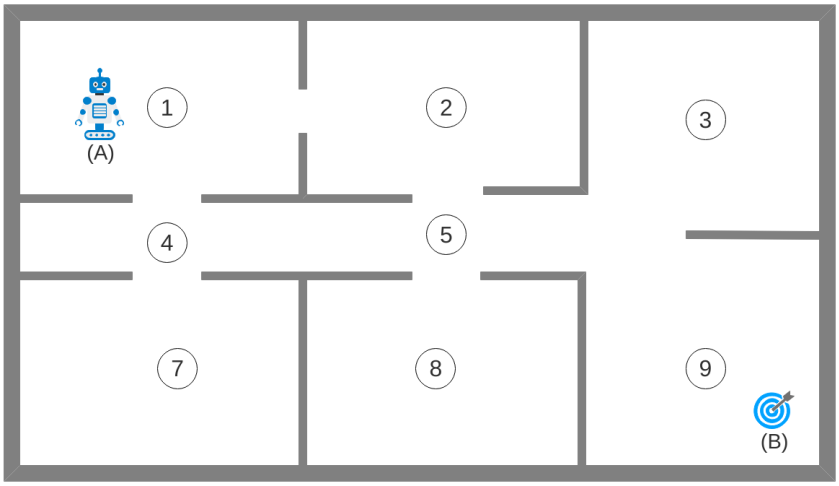


FIGURE 4.24 : Division de l'espace des configurations connu en zones.

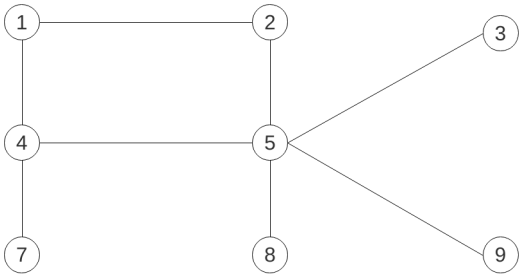


FIGURE 4.25 : Représentation du plan sous forme d'un graphe, chaque nœud représente une zone.

FIGURE 4.26 : Représentation du plan sous forme d'un graphe, à l'aide du diagramme de Voronoï.

5
Planification locale

Dans le chapitre 3 nous avons décrit la structure générale et le fonctionnement de l'architecture robotique permettant la navigation parmi les obstacles amovibles. Nous avons proposé une architecture robotique composée de deux planificateurs : (1) *un planificateur global* permettant l'avancement du robot dans les espaces libres (décrit dans le chapitre 4), et (2) *un planificateur local* qui se base sur la simulation multi-agents pour gérer les obstacles. Ce dernier planificateur sera présenté dans la suite de ce chapitre.

L'état de l'art de la NAMO présenté au chapitre 2 a montré diverses approches de résolution de la planification en environnement domiciliaire. Mais les résultats ont montré aussi que ces approches ne fonctionnent que sous certaines conditions (obstacles de forme géométrique uniquement, déplacement des obstacles dans une seule direction, etc.), donc nous avons conclu à la première section du chapitre 2 que ces méthodes ne sont pas applicables dans un environnement domiciliaire tel qu'elles sont présentées actuellement. Pour cela, nous avons proposé une approche basée sur deux planificateurs. Le premier, pour la navigation dans les espaces libres et le second pour la gestion des obstacles. Ce dernier utilise une approche basée sur des systèmes multi-agents (SMA) pour gérer les obstacles. Dans la suite de ce chapitre, nous allons détailler son architecture et son fonctionnement.

5.1 Vue d'ensemble

Le *planificateur local* a pour but de conduire le robot à une position déterminée en manipulant les obstacles qui gêne son passage. Dans notre proposition, la manipulation d'obstacles se fait par poussée (le robot pousse les obstacles pour les éloigner du passage.) ou par interaction (le robot demande à des obstacles interactifs de lui céder le passage.). Donc, le but est de trouver une séquence d'actions (plan d'actions) à exécuter pour atteindre une position définie par l'utilisateur en toute sécurité, c'est-à-dire :

- Éviter les obstacles fixes et fragiles (tables, télévision, vases de décoration, etc.).
- Pousser uniquement les obstacles amovibles (chaises, jouets, etc.).

- Interagir avec les obstacles tel que les humains et d'autres robots s'ils existent dans l'environnement du robot.
- Éviter les animaux de compagnie et les obstacles inconnus.

Pour ce faire, nous avons proposé un planificateur qui permet de faire une représentation de l'environnement sous forme d'un système multi-agents (SMA). Cette représentation permet d'effectuer des simulations pour déterminer le comportement de chaque élément de l'environnement et prédire l'état futur de l'environnement. Plus précisément, le SMA permettra de déterminer un plan d'actions. Puis pendant l'exécution de ce plan, le système vérifiera que les résultats de la simulation prédits sont conformes à la réalité, dans le cas contraire, le plan d'action est abandonné ou adapté.

Nous avons fait le choix d'utiliser les SMA pour représenter le monde dans lequel le robot évolue et nous avons discuté des avantages de cette représentation au chapitre 2. Pour y parvenir, nous avons besoin d'un SMA qui puisse représenter les différents éléments de l'environnement (obstacles fixes ou amovibles, les différents acteurs tel que les personnes et d'autres robots) ainsi que les lois qui régissent cet environnement, c'est-à-dire les comportements de chaque élément dans le monde. Nous proposons d'utiliser un SMA qui puisse représenter les éléments du monde sous forme d'agents (chaque obstacle est représenté sous forme d'un agent dans le SMA). Mais aussi, qui représente le monde du robot afin de rendre compte de la disposition des agents et de leurs comportements les uns envers les autres (en cas de collisions avec des obstacles, le comportement des objets lors de la pousser, pousser un obstacle sur une surface lisse réagit différemment si la surface est rugueuse comme par exemple sur un tapis par exemple, etc.), le comportement des agents suite à des actes de langage[1] (demander à un obstacle de se déplacer par exemple), etc. C'est-à-dire que l'environnement intégré au SMA permettra d'avoir une version virtuelle la plus proche possible de la réalité observée par le robot, le SMA engendré permettra d'élaborer des simulations dans le but de prédire le comportement réel des différents acteurs. Il permettra aussi, lors des actions du robot, de comparer les résultats prédits par la simulation aux comportements réels des objets dans le monde, ceci s'apparente à une boucle de rétroaction de haut niveau.

Le processus que nous proposons se déroule en trois parties (voir Figure 5.1), *agentification*, *simulation* et *action*. L'enchaînement de ces trois processus devrait conduire le robot à un nouveau sous-objectif. À chaque itération, le superviseur fixe de nouveaux sous-objectifs.

- **Agentification :** permet de traduire la scène observée sous forme d'agents dans le SMA. Le module de perception permet l'accès aux capteurs du robot (odomètres, caméras, capteur de force, etc). Ce module est doté de capacités de calculs, permettant la reconnaissance d'objets, estimer la position du robot relative aux obstacles, la position des obstacles, leur taille et le coefficient de frottement su sol. Le processus d'agentification permet de placer les agents dans le SMA en indiquant leurs types, tailles, positions, etc. Le SMA intègre une base de données (voir Tableau 5.2) afin de déterminer les actions possibles sur le type d'objets détectés (objets : fixes, amovibles, interactifs), et selon l'objet les actions associées (ensemble des messages $\mathbb{M}$ acceptés par un tel objet, etc.). La base de données est préalablement définie.

[1]Un acte de langage est un moyen mis en œuvre par un agent pour agir sur son environnement par des messages.

- **La supervision :** Déterminer les situations à simuler et choisir un plan d'action parmi ces choix. Le choix se fait sur la base d'une métrique déterminée sur le rapport entre l'énergie à dépenser et la distance à parcourir. Ici, on peut utiliser d'autres métriques tel que le *travail*[2]. Dans le cadre de ce travail, nous n'avons pas testé d'autres méthodes pour déterminer la plus efficace, il existe sans doute d'autres façons de produire une estimation.
- **Simulation :** permet de déterminer un plan d'actions pour atteindre une position définie à travers une simulation. Le superviseur détermine un objectif à atteindre et lance plusieurs simulations afin de déterminer le meilleur plan d'action conduisant à l'objectif et retourne un plan d'action avec des sous-objectifs. Avant la simulation, un instantané du SMA est sauvegardé afin d'y revenir une fois les simulations terminées.
- **Action :** permet d'exécuter un plan d'action issu de la précédente étape. L'exécution est assurée avec une boucle de contrôle[3], à chaque mouvement du robot le superviseur veuille à ce que le mouvement produit les effets attendus (prédit lors de la simulation), dans le cas contraire des corrections sont apportées afin de réajuster le mouvement du robot, au-delà d'un certain seuil l'action est bondonnée afin d'assurer la sécurité.

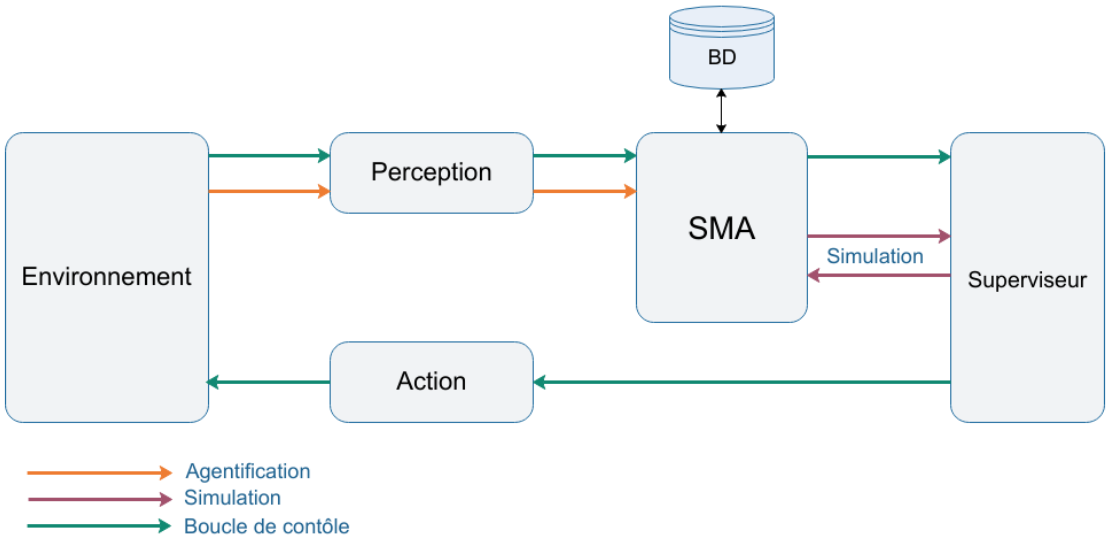


FIGURE 5.1 : Interaction des modules de planificateur local.

La Figure 5.1 montre le cheminement des actions des trois processus. Le processus d'agentification, montré avec les flèches oranges, utilise le module de perception pour alimenter le SMA avec les données de l'environnement, de nouveaux agents sont créés s'ils ne sont pas représentés dans le SMA, ou bien leurs données sont mis à jour dans le cas où ces objets ont déjà été observés. Le processus de simulation, montré avec les flèches rouges est piloté par le module de supervision. Ce dernier établi une liste d'objectifs à simuler, et demande au SMA d'effectuer une simulation pour atteindre chaque objectif au retour le SMA attribut un coût pour chacun des objectifs. Au préalable, le SMA sauvegarde son état avant la simulation et le recouvre une

[2]Ici le travail représente une grandeur physique, c'est-à-dire le produit de la force par le déplacement de son point d'application estimé suivant la direction de la force.

[3]Cette fonction est réalisée à l'aide d'une boucle de contrôle fermé, l'implémentation du système est donnée dans le chapitre 3.

fois les simulations terminées, ce qui permet de garder une représentation fidèle à l'observation. Le processus d'action, montré dans cette figure avec les flèches vertes, est aussi piloté par le module de supervision, ce dernier choisi un plan d'action (simulé auparavant), et l'applique à l'environnement à travers le module Action, mais au même temps le superviseur vérifie que le l'environnement se comporte conformément aux prédictions simulées préalablement. On peut remarquer que le superviseur utilise directement le processus d'agentification pour réaliser ces observations.

5.2 Agentification

À partir du monde observé, le robot construit une représentation en SMA (voir Figure 5.2). Chaque élément du monde est représenté dans le SMA sous forme d'un agent réactif, le robot lui-même est représenté sous forme d'un agent cognitif. Pour chaque agent dans l'environnement du SMA (sauf le robot) lui est associé des attributs (caractéristique e.g. taille, type, coefficient de frottement, type des messages qu'il est capable de traiter, etc.) et des actions possibles (les services e.g. déplacement, communication, etc.). Ces informations, selon leur type, soit ils sont connues par avance, est se trouve dans une base de données pré-remplie, ou déduites lors de la perception (par exemple les dimensions).

Certains obstacles ne sont pas détectés par le système de reconnaissance tel que les murs, les obstacles qui n'existent pas dans la base de données ou tout simplement les conditions n'ont pas permit la reconnaissance de ces objets (luminosité, distance, mise au point, etc.). Pour ces raisons, nous utilisons le LIDAR pour détecter et représenter ces obstacles dans le système multi-agent. Dans la pratique, nous utilisons en premier lieu la carte fournie par le LIDAR, chaque partie (cellule) de cartes est représentée par un agent qui représente un obstacle fixe. Par la suite, on utilise le système de reconnaissance, si des objets dont détectés alors on replace les agents crée à l'étape précédente (uniquement, les agents à la même position ou l'objet est détecté) avec de nouveaux agents qui correspondent réellement aux objets détectés.

Le processus d'agentification permet au robot d'analyser la scène observée afin de produire une représentation du monde sous forme d'un SMA. Pour cela, le robot utilise de multiples capteurs pour percevoir son environnement et se situer par rapport à d'autres obstacles. Un capteur RGB est couplé à une technique de reconnaissance d'obstacles permet d'identifier les obstacles. Un capteur de profondeur permet de d'indiquer la taille des obstacles ainsi que leur distance (voir Figure 5.3).

La reconnaissance des obstacles via la caméra RGB permet d'identifier les objets détectés, une base de données (c.f. § Base de données des objets) permet de connaître leur type (fixes, amovibles, interactifs). De plus, les objets détectés sont délimités par un rectangle (rouge dans la figure 5.3), ce dernier est transposé sur les données issues du capteur de profondeur et permet ainsi de déterminer les dimensions des objets ainsi que leur distance par rapport au robot.

L'exemple illustré dans la Figure 5.3 produit en sortie de l'analyse de la scène les données montrées dans le tableau 5.1 :

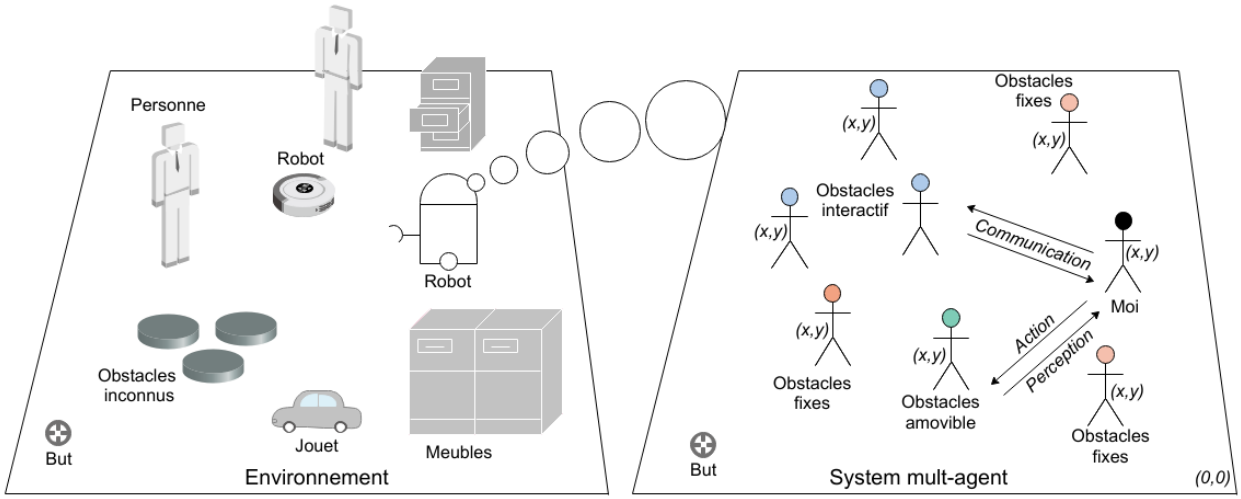


FIGURE 5.2 : Exemple de représentation d'un environnement réel en SMA. À gauche de l'image, se trouve l'environnement réel du robot et à droite, on voit la représentation multi-agents que fait le robot de son environnement. Chaque objet est représenté sous forme d'un agent situé. Le type des agents est représenté avec une couleur sur la tête des acteurs comment suit : bleu pour les agents interactifs, orange pour les agents fixes et vert pour les agents fixes, le robot lui-même est représenté en noir.

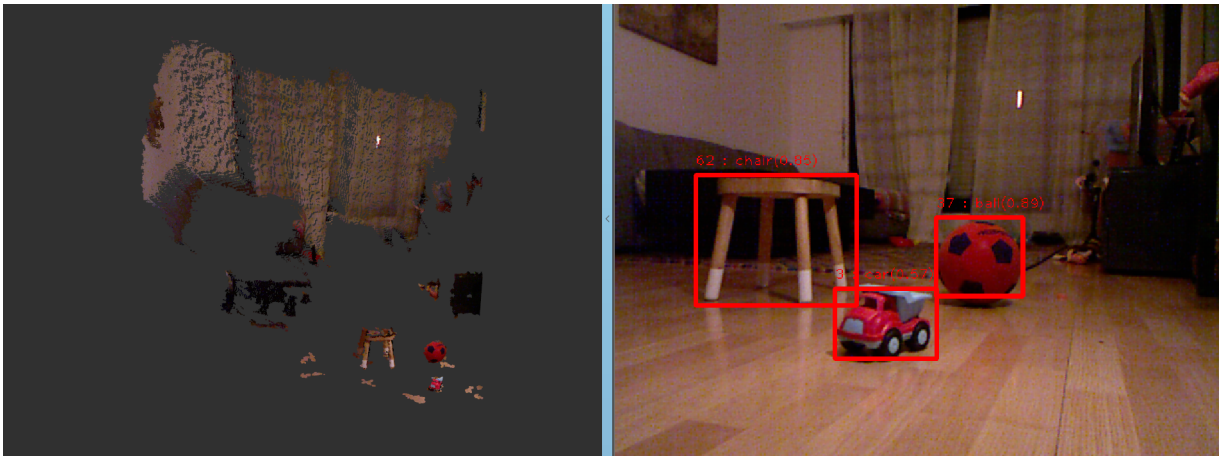


FIGURE 5.3 : Détection des objets à l'aide d'un capteur RGB (image de droite) et un capteur de profondeur (image à gauche). Les objets reconnus sont délimités par un rectangle rouge et dernier et transposés dans le nuage de données de profondeurs (image de gauche) afin d'extraire les dimensions de ces objets.

Base de données des objets

La liste des objets se trouve dans une base de données pré-remplit décrite ci-après.

Id	Type	Prob. certi-tude	Cat.	Dist.	Orienta-tion	Dim. X	Dim. Y	Dim. Z	Coef. frottem-ent
62	Chair	0.82	Mouvable	1418.41	343	22.67	24.14	32.95	0.4
3	Car	0.63	Mouvable	87.72	356	122.78	52.29	77.10	0.4
37	Ball	0.9	Mouvable	1861.54	9	36.41	70.84	27.75	0.4

Tableau 5.1 : **Id** : identifiant de l'objet dans la base de données, **Type** : nom de l'objet, **cat.** : catégorie de l'objet (fixe, amovible ou interactif), **Dist.** : Distance de l'objet par rapport au capteur, **Orientation** : Orientation en degrés du centre de l'objet par rapport au capteur, **Dim. X** : Largeur en mm, **Dim. Y** : Profondeur en mm, **Dim. Z** : Largeur en mm.

5.2.1 Structure des Agents

Pour représenter les obstacles sous forme d'agents, nous avons choisi une l'architecture réactive de subsomption proposée par Rodney Brooks [Brooks, 1986]. Cette architecture est composée de plusieurs niveaux, chaque niveau est responsable de la réalisation d'une seule tâche simple.

Les modèles d'agents réactifs (obstacles)

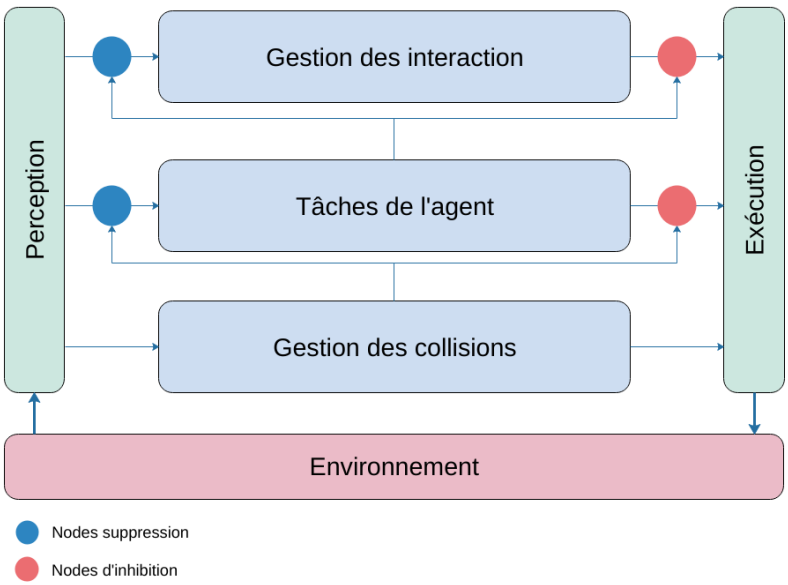


FIGURE 5.4 : Structure d'un agent réactif.

Les modèles d'agents :

- **Agent obstacle fixe :** Seul le niveau de gestion des collisions est implémenté. Toute collision provoque l'envoi d'un message d'alerte à l'agent cognitif (le robot).
- **Agent obstacle amovible :** Seul le niveau de gestion des collisions est implémenté. Toute collision provoque le déplacement dans le sens opposé.
- **Agent obstacle non-interactif :** Toute collision provoque l'envoi d'un message d'alerte à l'agent cognitif (le robot)

Tableau 5.2 : * Improbable en milieu domiciliaire, il s'agit sûrement d'un jouet. ** Probable en milieu domiciliaire

ID	Objet	Type	Description	Coefficient de frottement
1	person	Humain		0.4
2	bicycle	Fixe	**	0.4
3	car	Amovible	Jouet*	0.4
4	motorcycle	Amovible	Jouet*	0.4
5	airplane	Amovible	Jouet*	0.4
6	bus	Amovible	Jouet*	0.4
7	train	Amovible	Jouet*	0.4
8	truck	Amovible	Jouet*	0.4
9	boat	Amovible	Jouet*	0.4
10	traffic light	Inconnu		0.4
11	fire hydrant	Amovible	Jouet*	0.4
12	stop sign	Amovible	Jouet*	0.4
13	parking meter	Inconnu		0.4
14	bench	Fixe		0.4
15	bird	Animal	Animal domestique**	0.4
16	cat	Animal	Animal domestique**	0.4
17	dog	Animal	Animal domestique**	0.4
18	horse	Amovible	Jouet*	0.4
19	sheep	Amovible	Jouet*	0.4
20	cow	Amovible	Jouet*	0.4
21	elephant	Amovible	Jouet*	0.4
22	bear	Amovible	Jouet*	0.4
23	zebra	Amovible	Jouet*	0.4
24	giraffe	Amovible	Jouet*	0.4
25	backpack	Amovible		0.4
26	umbrella	Fixe	Objet fragile	0.4
27	handbag	Amovible		0.4
28	tie	Amovible		0.4
29	suitcase	Amovible		0.4
30	frisbee	Amovible		0.4
31	skis	Fixe		0.4
32	snowboard	Fixe		0.4
33	ball	Amovible		0.4
34	kite	Fixe		0.4
35	baseball bat	Amovible		0.4
36	baseball glove	Amovible		0.4
37	skateboard	Amovible		0.4
38	surfboard	Fixe		0.4
39	tennis racket	Fixe		0.4
40	bottle	Amovible		0.4

- **Agent obstacle interactif :** La réception d'un message, l'agent se déplace à la nouvelle position indiquée dans le message. Toute collision provoque l'envoi d'un message d'alerte

ID	Objet	Type	Description	Coefficient de frottement
41	wine glass	Fixe		0.4
42	cup	Fixe		0.4
43	fork	Fixe		0.4
44	knife	Fixe		0.4
45	spoon	Fixe		0.4
46	bowl	Fixe		0.4
47	banana	Fixe		0.4
48	apple	Fixe		0.4
49	sandwich	Fixe		0.4
50	orange	Fixe		0.4
51	broccoli	Fixe		0.4
52	carrot	Fixe		0.4
53	hot dog	Fixe		0.4
54	pizza	Fixe		0.4
55	donut	Fixe		0.4
56	cake	Fixe		0.4
57	chair	Amovible		0.4
58	couch	Fixe		0.4
59	potted plant	Fixe		0.4
60	bed	Fixe		0.4
61	dining table	Fixe		0.4
62	toilet	Fixe		0.4
63	tv	Fixe		0.4
64	laptop	Fixe		0.4
65	mouse	Fixe		0.4
66	remote	Amovible		0.4
67	keyboard	Fixe		0.4
68	cell phone	Amovible		0.4
69	microwave	Fixe		0.4
70	oven	Fixe		0.4
71	toaster	Fixe		0.4
72	sink	Fixe		0.4
73	refrigerator	Fixe		0.4
74	book	Amovible		0.4
75	clock	Fixe		0.4
76	vase	Fixe		0.4
77	scissors	Fixe		0.4
78	teddy bear	Amovible		0.4
79	hair drier	Fixe		0.4
80	toothbrush	Fixe		0.4

à l'agent cognitif (le robot).

La représentation interne du robot

Contrairement aux obstacles, le robot est représenté sous forme d'un agent cognitif dont le fonctionnement est décrit dans la section qui suit.

Choisir les situations à simuler

Le but de cet SMA étant de faire des simulations afin de prédire le comportement de l'environnement. Pour cela, l'agent cognitif (la représentation du robot) doit avoir un comportement proactif lors des simulations, son but est de modifier l'environnement afin d'atteindre un objectif, il doit essayer de trouver le meilleur chemin en essayant de bouger les obstacles, de communiquer pour demander qu'on lui cède le passage, etc. Sa stratégie consiste à représenter son environnement sous forme d'une grille[4] (voir Figure 5.6). Sur ces cellules, chaque type d'obstacle est représenté. Pour déterminer la meilleure option à choisir, on applique l'algorithme $D*$ *Lite* [Koenig and Likhachev, 2002], (voir algorithme 10).

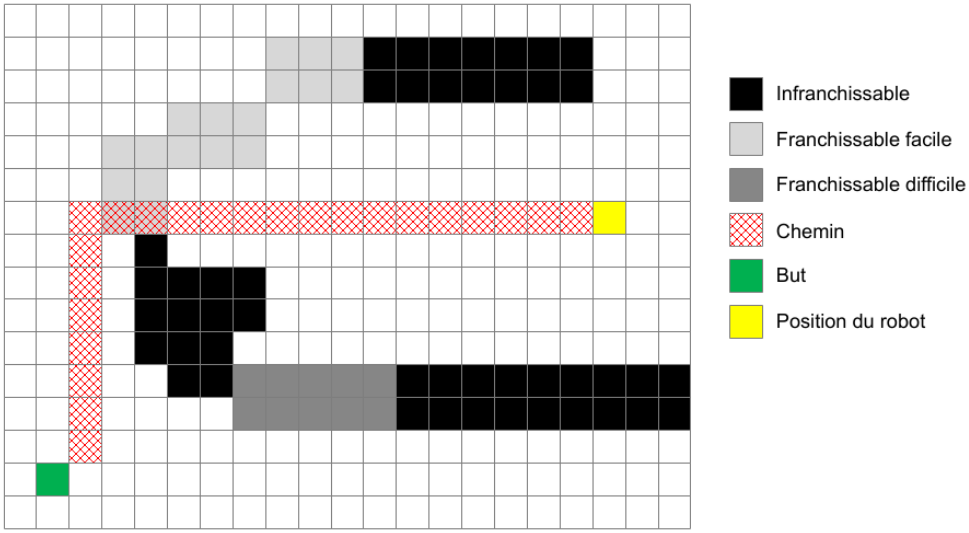


FIGURE 5.5 : Sélection de l'action.

La taille d'une cellule correspond à la largeur du robot, les obstacles peuvent occuper plusieurs cellules selon leur taille. Les obstacles sont de trois types (fixes, amovibles et interactifs) les espaces libres sont représentés en blanc dans la Figure 5.6. Les obstacles interactifs (gris clair) sont plus faciles à franchir, car ils nécessitent uniquement l'envoi d'un message. Les obstacles amovibles (en gris foncé) nécessitent plus ou moins d'énergie pour les pousser (selon leur taille et le coefficient de frottement). L'algorithme 10 prend en paramètres les contraintes suivantes afin de déterminer un chemin optimal. Si le chemin déterminé rencontre un obstacle franchissable, alors il est nécessaire d'effectuer des actions pour dégager le passage, dans le cas d'impossibilité, un nouveau chemin est recalculé. Le choix de $D*$ *Lite* permet une replanification au cours de la simulation sans avoir à relancer une nouvelle.

[4]Cette technique est connue sous le nome de la décomposition cellulaire, la grille représente un graphe où chaque cellule communique avec ses voisines. Le coût de passage d'une cellule à une autre en horizontale et vertical est de 1, et en diagonal est de $\sqrt{2}$.

La principale difficulté dans la mise en œuvre de cette solution consiste à déterminer un SMA capable de représenter l'environnement de façon assez détaillée. C'est-à-dire représenter l'environnement avec la position des agents, leur taille,... et aussi représenter les lois de cet environnement, par exemple gérer les collisions, (si le robot tente de simuler par exemple une situation dans laquelle il pousse un obstacle amovible, alors que celui-ci se trouve bloqué par un autre obstacle fixe dans le sens opposé), environnement en plan incliné (pousser un obstacle dans un environnement incliné peut se comporter d'une façon différente que sur un plan droit), deux agents ne peuvent pas être au même endroit, etc.

La tendance actuelle consiste à réduire l'environnement des SMA à de simples messages de communication, cependant ce type de représentation rendra difficile l'implémentation de cette solution. La section suivante traitera des environnements dans les SMA passe en revue certaines plateformes multi-agents susceptibles correspondre à nos attentes.

Simulation

L'agent commence par choisir les situations à simuler, et les placer dans une file à priorité, ordonnée selon la distance minimal (calculé à l'aide de l'algorithme D^* *Lite* à l'étape précédente). Pour chaque sous-objectif dans la liste, l'agent suit le chemin défini par D^* *Lite*, dans le cas de la rencontre avec un obstacle, une action est choisie selon le type de l'obstacle :

- Obstacle fixe : choisir un nouveau sous-objectif dans la liste.
- Obstacle amovible : déplacer l'obstacle dans le sens opposé.
- Obstacle non-interactif : choisir un nouveau sous-objectif dans la liste.
- Obstacle interactif : envoyer un message pour le déplacer dans le sens opposé.

L'historisation

Entre chaque simulation, il est essentiel de réinitialiser le SMA afin qu'il retrouve la situation initiale. Cette situation correspond à l'environnement observé lors de l'agentification. Il est essentiel que le SMA implémente un système d'historisation permettant de sauvegarder des situations afin d'y revenir à un autre moment. Dans l'idéal, il faudrait une plateforme multi-agents permettant l'historisation (snapshot ou instantané) de l'ensemble des agent de la plateforme, et l'historisation doit se faire sous forme de pile, ainsi, il est possible de naviguer entre plusieurs instantané.

Action

Lors de l'action est est essentiel que vérifier à tout moment que les actions entreprises dans le monde, correspondent aux simulations. Et lors de chaque action, il est essentiel de reporter les modifications apportées au monde dans le SMA afin de garder une représentation du monde cohérente. Pour des raisons d'optimisation, il est aussi essentiel de supprimer de la pile tout situation simulée qui n'a pas aboutie à une action.

5.2.2 Choix d'une plateforme SMA pour l'implémentation

Dans les sections précédentes, nous avons proposé d'utiliser un SMA dans lequel l'environnement[5] joue un rôle important. Les agents occupent une position définie dans l'environnement et la nature de l'environnement lui-même influe sur le comportement des agents. Dans la littérature, ce genre de SMA est connu sous le nom *SMA situé*.

Cependant, la plupart des recherches dans le domaine des SMA minimisent les responsabilités de l'environnement en le réduisant à la communication inter-agents ou en négligeant de l'intégrer en tant qu'élément principal, ce qui peut être suffisant en fonction des objectifs visés et selon le domaine d'application. Dans certains cas, l'environnement est un élément clé qui ne peut être réduit à une communication inter-agent, comme dans le cas de notre application, car le réduire à la communication inter-agents prive toute action de l'environnement alors dans ce cas, il devient un simple médium de communication. Dans notre application, l'environnement est une entité active avec ses propres processus pouvant changer d'état, indépendamment de l'activité des agents qu'il intègre. Donc, pour les besoins de notre application, nous avons besoin d'un SMA qui inclut l'environnement en tant qu'une entité dotée d'un ensemble de lois, ces dernières représentent les caractéristiques et contraintes de l'environnement. Les lois peuvent être considérées comme des règles que les agents ne peuvent enfreindre.

Notre recherche d'un SMA situé existant répondant à de telles exigences à souvent débouchée sur des travaux théoriques. Cependant, certains chercheurs se sont intéressés à l'intégration de l'environnement en tant qu'élément principal et ont proposé des modèles intéressants. Malheureusement, ils ne sont pas soutenus par des applications pratiques, comme nous pouvons le constater dans la suite de cette section. Pour cela, il nous est apparu raisonnable de proposer et d'implémenter un système adapté a nos besoins.

Plus précisément nous nous intéressons ici, à l'élaboration d'un système multi-agents situé doté d'un environnement, pour pouvoir réaliser le planificateur local de l'architecture robotique proposée au chapitre 3. Nous allons dans un premier temps étudier le rôle de l'environnement dans les systèmes multi-agents, par la suite nous allons nous intéresser aux plateformes existantes pour voir les possibilités de leur utilisation dans note architecture. Puis, nous allons proposer une plateforme adaptée à nos besoins. Finalement nous allons présenter l'implémentation du planificateur et les les résultats obtenus.

La plupart des plateformes multi-agents utilisées par la communauté scientifique et l'industrie, tels que Mobile-C [Chen et al., 2006], JADE [Bellifemine et al., 1999], JACK [Howden et al., 2001], Restina [Sycara et al., 2003], Zeus [Nwana et al., 1999] réduisent l'environnement à un système bésé sur l'échange de messages ou à une infrastructure de courtier (appelée *broker* en anglais). Certaines plateformes comme MadKit [Gutknecht and Ferber, 2000] et des plateformes éducatives telles que NetLogo [Tisue and Wilensky, 2004] sont dotées un système qui permet de localiser des agents dans un espace 2D ou 3D, ont peut

[5]Le mot "environnement" peut prêter à confusion. Pour éviter toute confusion quant à l'utilisation du mot *environnement*, dans la suite de ce chapitre, nous considérons l'environnement comme l'espace dans lequel les agents évoluent et non pas le monde réel du robot et nous allons utiliser le mot *monde* pour faire référence à ce dernier.

considérer cet espaces comment étant un environnement minimaliste, car on ne peut inclure les lois de l'environnement (collision, gravité, etc). Même dans d'autres travaux importants tels que les spécifications FIPA [Suguri, 1999], Il est difficile de trouver des fonctionnalités qui défini l'environnement autres que les systèmes de transport de messages ou de broker. Certaines méthodologies telles que Message [Bergenti et al., 2006], Prometheus [Padgham and Winikoff, 2002] et Trops [Bresciani et al., 2004] ne représentent pas l'environnement comme une entité fondamentale dans les SMA. Mais on trouve systématiquement dans la littérature de brèves discussions sur le rôle de l'environnement et son importance [Russell and Norvig, 2016] [Ferber and Weiss, 1999] [Briot and Demazeau, 2001]. Ces dernières années, la communauté scientifique a de plus en plus parlé de l'intégration de l'environnement en tant que partie principale de SMA. Parmi ces travaux, nous pouvons citer SODA [Omicini, 2000], une méthodologie dans laquelle l'environnement est pris en compte et fournit des abstractions et des procédures spécifiques pour la conception des infrastructures d'agents. Dans SODA, l'environnement est l'espace dans lequel les agents fonctionnent et interagissent.

Plus récemment D.Weyns et al. ont montré dans [Weyns et al., 2004] [Weyns et al., 2015] [Weyns and Michel, 2015] que l'environnement est important pour les systèmes multi-agents, et que l'environnement doit être considéré comme un élément de premier ordre et il doit être considéré comme étant une entité à part entière. Cependant, les implémentations des SMA dotées d'une telle capacité ne sont pas au rendez-vous. Pour surmonter le concept d'absence d'environnement dans SMA, la communauté scientifique a lancé une série d'ateliers E4MAS (Environments for multi-agents Systems) afin de relancer le débat et de trouver de nouvelles idées. La première série a eu lieu entre 2004-2007[6], et la deuxième série dix ans plus tard, de 2013-2014[7].

Dans la littérature, nous retrouvons le concept de *situated agents* (agents situés) ils vivent et agissent dans l'environnement. Les agents situés choisissent leurs actions en fonction de leur position, de l'état de leur monde perçu et de leur état interne. Contrairement aux agents basés sur la connaissance, les agents situés ne mettent pas l'accent sur la modélisation interne de l'environnement. Au lieu de cela, ils utilisent l'environnement comme source d'informations. Ce type de SMA est censé inclure la position des agents dans l'environnement, mais Wooldridge et Jennings [Wooldridge and Jennings, 1995] définissent un agent situé comme suit : "... un système informatique situé dans un environnement et capable d'action autonome dans cet environnement afin de répondre à ses objectifs de conception". Dans cette définition, « agents situés » fait référence à un agent de l'environnement, mais le concept d'environnement reste abstrait. La définition ne précise pas ce que signifie pour un agent d'être situé dans un environnement, rien dans la définition ne fait explicitement référence au fait que l'existence d'un agent dans un environnement comporte une composante sociale.

Dans ce travail, notre objectif est de reproduire le monde observé par le robot sous forme d'un système multi-agents. Ce dernier, servira à comprendre le monde observé et prédire son état futur pour une action donnée. Mais une simple représentation des différents acteurs qui composent le monde, sous forme d'agents ne suffira pas, car il faut aussi représenter les lois qui régissent ce monde, par exemple, on peut pousser sur un obstacle fixe et ce dernier ne bougera

[6]E4MAS - Environments for multi-agents Systems : https://distrinet.cs.kuleuven.be/events/e4mas/
[7]E4MAS - 10 Years Later : http://homepage.lnu.se/staff/daweaa/events/E4MAS/2014.htm

pas, pousser un obstacle qui a son tout pousse un autre, etc. Donc nous cherchons à produire SMA qui peut avoir quelques caractéristiques communes avec les moteurs physiques.

Il existe une alternative à l'utilisation des SMA, on aurait pu utiliser un moteur physique pour représenter l'environnement. Cette solution peut sembler efficace mais difficile à mettre en œuvre ; les moteurs physiques utilisent des lois newtoniennes pour simuler le comportement de l'environnement, mais pour utiliser ces lois, le robot doit connaître les caractéristiques exactes de chaque élément de l'environnement, ce qui rend la tâche complexe ou pratiquement impossible à mettre en œuvre dans un robot dans un environnement réel. L'utilisation de la SMA semble être un bon compromis.

5.3 Étude des modèles d'environnement pour les SMA

Pour concevoir un SMA situé, il est important de se baser sur des travaux axés sur les environnements dans les SMA. Les articles [Weyns et al., 2004] [Weyns et al., 2015] [Weyns and Michel, 2015] Présent un état un large état de l'art du domaine, ainsi plusieurs pistes de recherche qui incluent une certaine notion de l'environnement sont présentées. Dans la suite de cette section, nous allons résumer les plus importants d'entre elles.

Dans [Russell and Norvig, 2016], les auteurs proposent une représentation simple de l'environnement et de ses interactions avec les agents. Selon ces auteurs, "un agent est tout ce qui peut être perçu comme percevant son environnement à travers des capteurs et agissant sur l'environnement à travers des effecteurs". Selon Russell et Norvig, l'environnement contient les propriétés suivantes :

- **Accessible or inaccessible** : indique si les agents peuvent obtenir des informations complètes et exactes sur l'état de l'environnement ou non.
- **Déterministe ou non déterministe** : indique si un changement d'état de l'environnement est uniquement déterminé par son état actuel et les actions sélectionnées par les agents ou non.
- **Statique ou dynamique** : indique si l'environnement peut changer pendant qu'un agent délibère ou non.
- **Discret ou continu** : indique si le nombre de percepts et d'actions est limité ou non.

Ces propriétés sont maintenant adoptées par la plupart des chercheurs dans le domaine des SMA.

Dans [Ferber and Weiss, 1999], J. Ferber propose un autre point de vue sur l'environnement :

- L'environnement est discret et composé d'un ensemble de **cellules** (probablement pour simplifier les processus calculatoires et réduire la puissance de calcul nécessaire, ce n'est pas bien précisé (voir page 212-214 [Ferber and Weiss, 1999])).

- **Environnement centralisé ou décentralisé** : l'environnement peut être centralisé (les cellules sont regroupées dans un système monolithique) ou décentralisé (les cellules sont reliées entre elles par un réseau). Les agents évoluent et perçoivent leur environnement à travers ces cellules. Cependant, une cellule de l'environnement distribué diffère à plusieurs égards d'un environnement centralisé.

- **Environnement généralisé ou spécialisé** : un modèle généralisé de l'environnement est indépendant du type d'actions pouvant être effectuées par les agents. Un modèle spécialisé d'environnement est caractérisé par un ensemble d'actions bien définies.

- **Influences et réagit aux influences** : les influences proviennent d'agents et tentent de modifier le cours des événements dans l'environnement. L'environnement produit des réactions - qui à leur tour provoquent des changements d'état en combinant les influences de tous les agents, compte tenu de l'état local de l'environnement et des lois du monde.

Dans [Parunak, 1997] [Odell et al., 2003] Parunak, Odell et al. définissent l'environnement comme un espace fournissant les conditions dans lesquelles des agents existent. Les auteurs font la distinction entre environnement physique et environnement de communication.

- **L'environnement physique** fournit les lois, règles, contraintes et des politiques qui régissent et soutiennent l'existence physique d'agents.

- **l'environnement de communication** fournit des processus qui régissent et soutiennent l'échange d'idées, de connaissances et d'informations.

Les fonctions et les structures sont couramment utilisées pour échanger des communications, telles que des rôles, des groupes et des protocoles d'interaction entre les rôles et les groupes. Ils donnent une spécification dans [Parunak, 1997], un SMA est défini comme un triple-tuple : un ensemble d'agents, un environnement et un couplage entre eux, comme suit :

$$MAS =$$

$$Agents = Agent_1, ..., Agent_n$$

$$Agent_i =$$

$$Environment =$$

Un agent est défini comme l'ensemble des *états*, *entrées*, *sorties* et *processus*. Un état est l'ensemble des attributs qui définissent l'agent, les différences entre ces attributs étant responsables de la variation entre différents types d'agents. Les entrées et les sorties sont des sous-ensembles d'état, dont les variables sont couplées à l'environnement. Les entrées et les sorties peuvent représenter les capteurs et les effecteurs d'un agent. ils relient l'agent à son environnement. Ce sont les mécanismes qui implémentent le couplage entre l'environnement et les agents. Le processus est un mappage à exécution autonome qui modifie les états de l'agent.

L'environnement est défini comme une entité active. Il a son propre processus qui peut changer d'état, indépendamment des actions de ses agents intégrés. Les entrées et les sorties des agents incorporés sont couplées à des éléments de l'état de l'environnement, mais l'environnement ne fait pas la distinction entre les éléments d'état couplés de cette manière. Cette distinction dépend des agents qui existent à tout moment et des capacités de leurs capteurs et effecteurs.

Dans [Rao et al., 1992] Rao et al. décrivent les caractéristiques d'un environnement générique :

- Les agents peuvent évoluer de nombreuses manières différentes dans l'environnement ;
- L'environnement peut être affecté par plusieurs actions en même temps ;
- Différents objectifs peuvent ne pas être réalisables simultanément ;
- Les actions qui répondent le mieux aux différents objectifs dépendent de l'état de l'environnement ;
- L'environnement ne peut être détecté que localement ;
- La vitesse à laquelle les calculs et les actions peuvent être effectués est raisonnable liés au taux de changement de l'environnement. Rao et ses collègues décrire les caractéristiques typiques du monde extérieur dans lequel les systèmes d'agents sont déployés et avec lesquels les systèmes d'agents interagissent.

Dans [Demazeau, 2003] Demazeau considère que les quatre composants essentiels des systèmes multi-agents sont :

- Les agents ;
- Les interactions (éléments structurants des interactions internes entre entités),
- L'organisation (éléments structurants des ensembles d'entités au sein de la SMA),
- L'environnement défini en tant qu'éléments dépendants du domaine pour structurer les interactions externes entre les entités.

Dans [Ferber, 1997] J. Ferber a souligné que les systèmes multi-agents peuvent être utilisés pour la résolution de deux grandes catégories de problèmes : la simulation de phénomènes complexes et la résolution distribuée de problèmes. Nous pensons que les modèles décrits ci-dessus sont trop génériques pour développer un modèle qui puisse traiter efficacement ces deux problèmes, et nous pensons que c'est la raison principale pour laquelle de nombreux chercheurs ignorent l'intégration de l'environnement dans leurs modèles. Souvent, l'environnement n'est indispensable que dans la simulation de phénomènes complexes. Pour notre part, nous pensons que les systèmes multi-agents sont une abstraction du système que nous voulons simuler. Il est donc important de prendre en compte le fait que les agents agissent sur l'environnement et que l'environnement agit également sur les agents, ce qui est également le cas dans les modèles précédents. Cependant, nos avis divergent en :

- Tous les agents ne subissent pas les mêmes effets de l'environnement.

- Les agents perçoivent l'environnement à l'aide de capteurs, de même que les informations susceptibles d'être asymétriques.
- Les agents peuvent pas ressentir certains effets de l'environnement sur eux.
- L'environnement est le support physique des agents et ressemble à un moteur physique, il consiste en un ensemble de lois qui déterminent les actions possibles des agents. Nous pouvons dire que notre approche consiste à proposer un modèle centré sur l'environnement pour la représentation de phénomènes complexes et à incorporer les idées que nous jugeons pertinentes à partir des modèles décrits ci-dessus.

5.4 Revue des plateformes SMA disposant ou pas d'un environnement

Les plateformes multi-agents sont essentielles, car généralement, c'est à travers elles que les SMA sont implémentés. Selon les plateformes, elles peuvent ou ne peuvent pas convenir à l'implémentation d'un paradigme particulier. On peut affirmer avec certitude par exemple que certaines plateformes ne sont pas adaptées à l'implémentation d'un SMA doté d'un environnement tel que c'est décrit précédemment. Donc le choix d'une plateforme est important pour la simplicité voire même la réussite de l'implémentation d'un SMA. Dans cette section, nous allons passer en revue certaines plateformes existantes afin d'arrêter notre choix sur la plateforme à utiliser dans nos travaux.

JADE [Bellifemine et al., 1999] (Java Agent Development Framework) est un middleware qui facilite le développement de systèmes multi-agents, il est conforme à la norme FIPA développée en JAVA. Il inclut un environnement d'exécution avec des agents JADE, sur lequel un ou plusieurs agents peuvent être exécutés sur l'hôte. Une bibliothèque de classes que les programmeurs doivent/peuvent utiliser pour développer leurs agents. Une suite d'outils graphiques qui permet l'administration et la surveillance de l'activité des agents en pleine exécution. JADE ne présente aucun outil permettant de visualiser l'environnement. Cette plateforme ne possède pas d'environnement à proprement parler, mais utilise un système de communication tel que c'est décrit dans la norme FIPA, c'est à dire un *Système de Gestion d'Agents* (Agent Management System - AMS) pour superviser l'acces des agents à la plateforme. Un *Canal de Communication entre Agents* (Agent Communication Channel - ACC) qui correspond à un bus de communication qui route les messages. Et finalement *Le Facilitateur d'Annuaire* (Directory Facilitator - DF) qui correspond un service de pages jaunes qui permet d'identifier les compétences des agents.

Netlogo [Tisue and Wilensky, 2004] est un outil de modélisation multi-agents centré sur l'environnement [Macal and North, 2008] [Kornhauser et al., 2007], il intègre son propre langage de programmation qu'on peut qualifier d'un langage de haut niveau. L'environnement est discret, il est représenté sous forme 2D ou 3D selon la version utilisée. Netlogo représente les agents obligatoirement dans l'environnement et ne peuvent communiquer uniquement via ce dernier. Dans Netlogo, il est possible de représenter un troisième type d'objet qu'on appel les liens, ils mettent en relation deux agents afin de symboliser une relation entre agents. Son utilisation est limitée en raison d'un manque d'une API permettant à d'autres logiciels

de communiquer directement avec lui. Généralement ils sont utilisation est limitée aux milieux pédagogiques.

Gama [Drogoul et al., 2013] La plate-forme GAMA (Gis & Agent-based Modelling Architecture) à l'image de Netlogo, elle offre un langage complet de modélisation - le GAML (GAma Modeling Language) - permettant aux modélisateurs de construire simplement et rapidement des modèles. Néanmoins, contrairement à Netlogo qui se limite à la construction de modèles simples, GAMA permet la construction de modèles très complexes, aussi riches que ceux construits par un informaticien à partir d'outils tels que Repast Simphony. En particulier, GAMA propose des outils très avancés pour ce qui concerne la gestion de l'espace. Son utilisation avec d'autres applications reste difficile, car ce dernier se base sur des modules et non pas une API à proprement parler.

Mason [Luke et al., 2005] MASON est une bibliothèque de simulation multi-agents à événements discrets et rapide. développée en Java, conçu pour servir de base à de grandes simulations Java personnalisées, et pour fournir suffisamment de fonctionnalités pour de nombreux besoins de simulation légère. MASON contient à la fois une bibliothèque de modèles et une suite optionnelle d'outils de visualisation en 2D et en 3D. À l'image de Netlogo, elle permet une visualisation des agents, mais l'aspect de communication entre les agents est négligé.

5.5 Intégration d'un environnement pour JADE

Parmi les plateforme multia-gents les plus populaires, nous trouvons JADE [Bellifemine et al., 1999]. Cette plateforme est beaucoup utilisée dans les travaux de recherche, car elle implémente le standard FIPA [Fipa, 2002], donc facilement interopérable avec d'autres plateformes qui implémentent le même standard, d'autre part elle est particulièrement bien documenté [Bellifemine et al., 2007] est a fait ses preuves en vue de la quantité de systèmes déjà implémentés avec JADE.

Malheureusement JADE ne présente pas un système de représentation spatial (environnement) en comparaison avec les plateforme tel que GAMA et Netlogo. Afin de faire profiter JADE de tous les avantages que proposent les plateformes qui intègrent la représentation spatiale, nous avons conçu une librairie Java qui s'intègre facilement avec JADE sous le nom de JEX pour (JADE Environnement Extension) [Djerroud and Cherif, 2018].

L'idée consiste à doter JADE d'un environnement pour utiliser cette plateforme dans le planificateur local pour la gestion des obstacles décrit dans le Chapitre 5. Nous avons équipé JADE d'un environnement à travers le composant JEX, mais nous ne l'avons pas équipé d'un système permettant de gérer les lois de l'environnement, mais seulement d'un système permettant la localisation et la visualisation des agents. L'intégration d'un environnement à JADE c'est avéré complexe et nous n'avons pas pu intégrer un environnement avec des lois comme nous l'avons souhaité, car cela impose des changements dans la structure interne de JADE, notamment pour que l'environnement accède aux attributs des agents pour les manipuler. Un problème de compatibilité est apparu lors des expérimentation avec le middleware ROS. Ce qui nous a motivé pour changer d'approche. Nous souhaitons comme même montrer brièvement dans la prochaine section l'architecture JEX.

5.5.1 Architecture de JEX

JEX est l'acronyme pour JADE Environnement Extension. L'idée générale consiste à créer trois modèles d'agents à partir des agents JADE. L'environnement est créé à partir d'un ensemble d'agents, où chaque agent représente une partie de l'environnement (patch ou cellule). Les agents JEX, sont des agents JADE auxquels nous avons rajouté des attributs de position, actions et de déplacement dans l'environnement. Les liens entre les agents JEX que l'utilisateur peut utiliser pour créer des liens entres deux agents JEX à l'instar de Netlogo et GAMA.

La Figure 5.6 illustre le diagramme UML de l'architecture interne de JEX. Afin de représenter les différents éléments qui composent l'environnement nous faisons appel à trois types d'agents. (a) Agents : représentent les agents à proprement parler qui peuvent agir sur l'environnement. (b) Patchs : représentent les différentes parties qui composent l'environnement. (c) Links : représentent les liens entre agents. Ces différents agents sont eux même des agents JADE comme illustré dans la figure avec la relation d'héritage qui existe entre ces différents agents et les agents JADE. Nous avons fait le choix de cette implémentation afin de profiter au maximum des avantages qu'offrent les agents JADE et garder une compatibilité avec ce dernier.

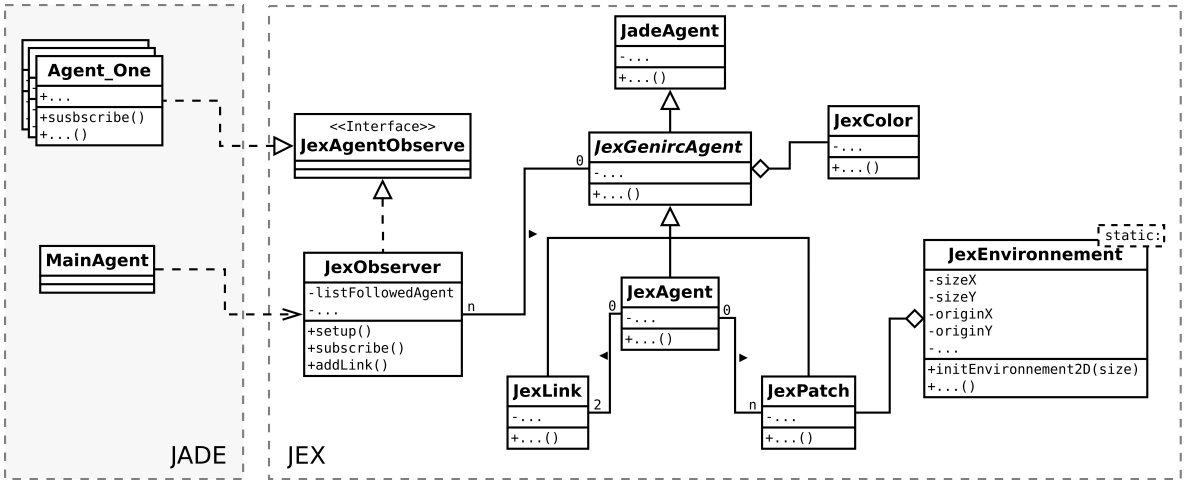


FIGURE 5.6 : Architecture JEX : diagramme UML.

Dans l'architecture que nous proposons, l'environnement est composé d'un ensemble de Patchs. L'utilisateur a le choix de définir les dimensions de l'environnement, l'enroulement peut-être autorisé ou non, les dimensions des Patchs peuvent elles aussi être choisies. Chaque élément (Patchs) peut être manipulé indépendamment des autres éléments par les différents agents qui évoluent dans l'environnement. Concrètement, l'environnement est représenté par une classe aux membres statiques, ce choix se justifie par le fait de ne pas autoriser l'existence que d'un seul et unique environnement. D'autres paramètres globaux s'ajoutent par exemple le délai de rafraichissement, le point origine de l'environnement ainsi que d'autres paramètres indiqués dans la documentation de JEX[8].

L'agent observateur JexObserver a pour rôle d'inscrire tout agent JADE désirant bénéficier d'une représentation dans l'environnement. Ce dernier offre aussi d'autres services comme

[8] https://github.com/hdd-robot/JEX

la création de liens (Links) et l'initialisation de l'environnement. Nous insistons sur le fait que ces différentes actions sont totalement transparentes pour l'utilisateur, elles sont réalisées d'une façon automatique. Pour terminer, JEX propose une Interface que les agents souhaitant bénéficier d'une représentation graphique doivent implémenter.

JEX se présente sous forme d'une librairie java *jex.jar*. Cette librairie permet de doter JADE d'un environnement graphique qui permet de visualiser les agents et l'environnement.

L'intégration de JEX dans un projet JADE, ne nécessite pas de modifications du projet JADE, il faut simplement créer un agent de type **JexObserver**, cet agent permet de configurer l'environnement, par exemple (la longueur et la largeur de l'environnement, le temps de rafraîchissement ...). Si on spécifie aucun de ces paramètres, des valeurs par défauts seront prises. Dans la portion de code qui suit on présente la façon dont l'agent **JexObserver** est créé.

Le choix de construire une plateforme SMA :

Malgré nos efforts pour utiliser une plateforme déjà existante dans notre projet de recherche, JADE et JEX ont révélé des limites techniques. On peut citer par exemple la difficulté de création d'instantané du SMA pour pouvoir sauvegarder le contexte actuel et lancer des simulations dans un bac à sable. La difficulté d'interaction avec ROS, malgré l'existence de certaines bibliothèques permettant cela, des problèmes de compatibilité se sont très vite manifestés. Pour cela, nous avons fait le choix de construire une plateforme multi-agents adapté a nos besoins.

5.6 Proposition d'une plateforme pour les SMA situés (gAgent)

D'après ce que nous avons vu dans les sections précédentes, nous pouvons distinguer deux catégories principales d'utilisation de SMA : la première est l'étude de phénomènes complexes et la seconde est la résolution de problèmes distribués [Ferber, 1997]. Dans le premier cas, l'environnement joue un rôle essentiel. Des études de SMA axés sur l'environnement [Weyns et al., 2004, Weyns et al., 2015] [Weyns and Michel, 2015, Russell and Norvig, 2016] [Ferber and Weiss, 1999, Odell et al., 2003] [Rao et al., 1992, Demazeau, 2003] montrent deux types de difficultés : premièrement, la définition de l'environnement est trop large et le terme *environnement* a plusieurs interprétations, ce qui porte à confusion. Deuxièmement, les fonctionnalités associées à l'environnement sont intégrées et traitées par les agents. De notre point de vue, l'environnement doit être une entité séparée, et doit interagir avec, et influencer le comportement des agents. Dans la suite de cette section, nous décrivons notre vision de l'environnement et proposons un nouveau modèle SMA et son implémentation sous forme d'une plateforme (framework) multi-agents sous le nom de**gAgent**. Le modèle proposé ci-dessous est fortement inspiré des SMA et des plateformes examinées dans la section précédente. Nous avons choisi les points que nous pensons être importants pour l'implémentation de VICA.

Nous pensons que l'environnement doit être une entité autonome et active et non intégrée aux agents, comme nous l'avons vu précédemment, par exemple dans les messages ou les courtiers [Djerroud and Cherif, 2019]. Notre approche consiste essentiellement à représenter

l'environnement en tant qu'entité à part entière, comme c'est le cas dans [Parunak, 1997]. Premièrement, nous définissons l'environnement comme un espace dans lequel les agents évoluent. Deuxièmement, l'environnement est comme un moteur physique, Il doit inclure des règles inviolables. Troisièmement, comme dans les SMA classiques, les agents agissent sur l'environnement et changent d'état. Enfin, l'environnement peut modifier l'état et les attributs des agents afin de maintenir ses règles inviolables. Si un agent tente de transgresser une de ces règles, l'environnement en modifie l'état de l'agent afin de maintenir son intégrité. De manière plus précise, les agents agissent sur l'environnement et l'environnement agit également sur les agents. Cependant, les agents ne peuvent ressentir les effets de l'environnement que s'ils disposent de capteurs pour ressentir ces effets.

Donc nous pouvons retenir en conclusion qu'un système multi-agents (MAS) peut être défini comme un triple : un ensemble d'agents, un environnement et leurs interactions :

$$MAS =< Agents, Environment, Interactions >$$
$$Agents = Agent_1, ..., Agent_n$$

5.6.1 L'environnement

Dans notre proposition, l'environnement est divisé en deux parties, la première étant représentée dans un processus monolithique, tandis que la seconde serait intégrée aux agents (répartis sur tous les agents). 1) La partie monolithique de l'environnement représente les caractéristiques physiques et fonctionnelles de l'environnement, telles que : les dimensions de l'environnement, type de l'environnement (discret / continue), taille des cellules, les lois sous forme de règles à définir en fonction de l'environnement que nous voulons représenter, etc. 2) Chaque agent du système contient une partie des informations de l'environnement, comme illustré à la Figure 5.7. Généralement, cette information concerne l'agent lui-même ; par exemple : sa position dans l'environnement, son énergie, etc. Il inclut également toutes les actions possibles pouvant être effectuées sur l'environnement, telles que se déplacer, laisser une trace sur une cellule, etc.

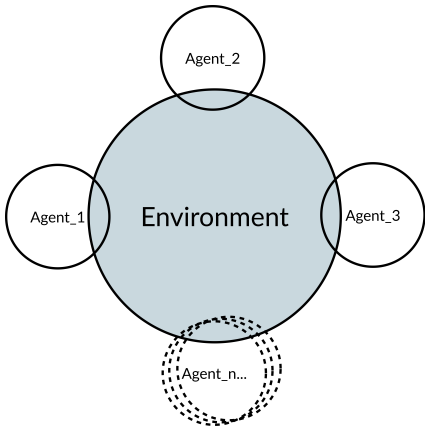


FIGURE 5.7 : L'environnement est partagé entre une partie monolithique et l'autre partie est intégrée aux agents qu'il héberge.

Dans notre proposition, l'environnement est régi par des lois. L'environnement peut être assimilé à un **moteur physique**[9]. Les agents qui évoluent dedans sont donc soumis à ces lois. Nous pouvons donc considérer que l'environnement est une entité qui agit sur les agents. Ces agents ne peuvent réellement ressentir les effets de l'environnement que s'ils sont équipés de capteurs pour percevoir l'impact de ces lois. Plus concrètement, les agents possèdent deux types d'attributs : (1) les croyances et les connaissances qui sont des informations auxquelles l'agent peut consulter et modifier. (2) des attributs que l'environnement peut manipuler, mais l'agent n'a pas connaissance que ces attributs sont modifiés. Cette technique permet par exemple à implémenter des obstacles sous forme d'agents, ces derniers n'ont pas la capacité de percevoir le monde, mais doivent réagir lorsque ils sont poussés par un mobile.

Bien entendu, on considère qu'il incombe à l'utilisateur de la plateforme de déterminer les lois qui seront mises en œuvre par l'environnement, et il appartient également à l'utilisateur de déterminer les attributs des agents et de l'environnement. L'environnement et les agents sont deux entités distinctes : (1) les agents sont constitués d'un ensemble d'attributs et des connaissances sur leurs actions possibles sur l'environnement ; ces agents peuvent consulter leurs attributs et effectuer des actions sur l'environnement. (2) l'environnement a des lois ; il a la responsabilité de les faire respecter et vérifie le respect de ces lois, dans le cas contraire, l'environnement change automatiquement les attributs des agents afin de respecter ces lois. Par exemple un agent mobile qui avance dans une position déjà occupée (change ses coordonnées de position), l'environnement intervient et remet les coordonnées à des valeurs plus raisonnables, bien entendu si la loi interdit la présente de deux agents au même endroit. Ainsi, nous pouvons définir l'environnement comme un triplet, États, agents et processus :

$$Environment =< States, Rules, Process >$$

$$States =< SharedAttributes, InternalAttributes >$$

States (État) : Est un ensemble d'attributs qui définissent complètement l'environnement. Les valeurs des attributs représentent l'état de l'environnement à un moment donné. Les attributs de l'environnement représentent ses caractéristiques, telles que la taille de l'environnement, les positions des agents qui y évoluent, etc. Nous distinguons deux types d'attributs : *SharedAttributes (attributs partagé* et *InternalAttributes (attributs internes)*. Le premier type d'attribut est partagé avec les agents. La seconde n'est pas partagée avec les agents, il s'agit généralement des propriétés internes de l'environnement lui-même.

Rules (lois ou règles) : Les lois de l'environnement sont des règles qu'il doit respecter. Par exemple, deux agents ne doivent pas être au même endroit à la fois et les règles sont écrites comme suit :

$$rule =< expression\ ?\ action1 : action2 >$$

$$expression =< Attribut_i\ Operator\ Attribut_j >$$

[9]Une alternative à l'utilisation des systèmes multi-agents le robot peut utiliser un moteur physique pour représenter son environnement. Cette solution peut paraître efficace mais difficile à mettre en œuvre. Car les moteurs physiques utilisent les lois newtoniennes pour simuler le comportement de l'environnement, or pour utiliser ces lois le robot doit connaître les caractéristiques exactes de chaque élément de l'environnement ce qui rend la tâche complexe voir impossible. Donc l'utilisation des SMA est un bon compromis.

$$Operator =< Opérateurs\ logiques >$$
$$Attribut_x =< Agent_x \mid\mid Environement >$$

Une règle est représentée comme une expression. Elle correspond à la loi que l'environnement doit vérifier, les deux actions correspondant respectivement aux actions devant être effectuées selon que la règle est respectée ou non. Dans l'exemple précédent (la règle qui interdit à deux agents de se trouver au même endroit en même temps), cela peut être représenté comme suit : si un agent tente de se déplacer vers un emplacement déjà occupé par un autre agent, empêchez le mouvement ; sinon ne fais rien.

L'environnement est représenté comme un processus monolithique avec un ensemble d'attributs et effectue des contrôles de temps en temps afin de respecter ses lois. Les agents utilisant l'environnement doivent d'abord s'inscrire auprès de l'environnement. À chaque fois que les agents modifient leurs attributs, ils doivent informer l'environnement via un message. Par exemple, un agent qui se déplace doit chaque agent est un tuple d'état et de processus : donc modifier un ou plusieurs de ses attributs et à ce stade, il informe l'environnement.

L'environnement a également la capacité de changer les attributs des agents. Par exemple, si le décalage n'est pas possible en raison d'une règle non respectée, il en informe l'agent et renvoie les valeurs des attributs à leurs anciennes valeurs.

Process (processus) : Un environnement a son propre processus qui peut changer d'état, indépendamment des actions des agents intégrés. Le but principal de *Process* est d'implémenter le dynamisme dans l'environnement, par exemple. Les processus qui vérifient les règles et exécutent les actions, le comportement des objets dans l'environnement, etc.

5.6.2 Agent

Chaque agent est un tuple de States et de Process :

$$Agents = Agent_1, ..., Agent_n$$
$$Agent_i =< States, Process >$$
$$States =< SharedAttributes, InternalAttributes >$$

States : Les attributs d'agent sont un ensemble de valeurs qui définissent l'agent. La définition et la structure de ces valeurs ne sont pas contraintes par cette définition, et les différences entre ces caractéristiques sont responsables de la majeure partie de la variation entre les différents types d'agents.

Les agents sont définis avec deux types d'attributs : "Attributs partagés" et "Attributs internes". Les attributs partagés sont la partie de l'environnement concernant l'agent, par exemple. Position de l'agent dans l'environnement. Les attributs internes définissent l'état interne de l'environnement, par exemple le journal de ses mouvements (ensemble des mouvements effectués), son but, le temps interne, etc.

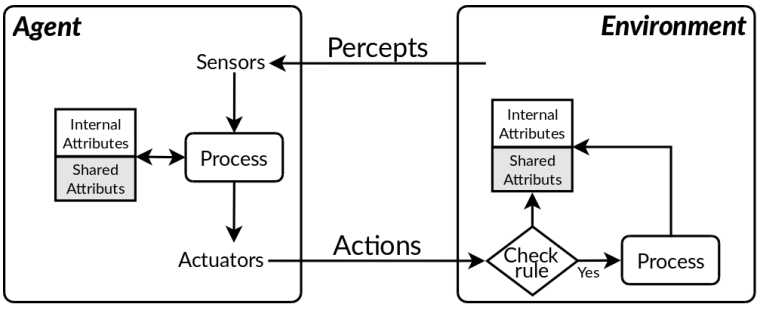


FIGURE 5.8 : Cette Figure illustre la relation entre un agent et son environnement. L'agent et l'environnement ont des attributs partagés et chacun d'eux a ses propres attributs qui le définissent. L'environnement est également composé de lois. Les agents agissent sur l'environnement par le biais d'actions. L'environnement peut directement modifier les attributs de l'agent si les lois sont transgressées.

Process : Les processus sont des mappages autonomes qui changent l'état de l'agent. L'agent peut effectuer ces processus sans être appelé par une entité externe. En termes de calcul, un agent dispose de son propre processeur. Dans **gAgent** chaque agent est implémenté dans un processus.

5.6.3 Interactions entre agents et environnement

Nous proposons que le système multi-agents soit composé d'agents et de l'environnement ; lorsque les agents et l'environnement sont basés sur le temps, les valeurs d'état peuvent varier de manière continue. La variation d'une variable résultant d'un tel flux peut être infinitésimale, en fonction du processus dans l'entité réceptrice et des autres flux d'énergie dans le système.

Pour résumer notre approche, nous pouvons définir un SMA situé sur un moteur d'environnement[10] comme suit :

- L'agent a une connaissance filtrée (partielle) et parfois déformée de la connaissance de son environnement [11].
- L'environnement est régi par des lois et ces lois peuvent être connues ou inconnues des agents.
- L'environnement agit sur les agents.
- Les agents agissent sur l'environnement avec des actions.
- Les agents observent l'environnement au moyen d'attributs pouvant être influencés par l'environnement.

[10]Nous utilisons ici le nom *moteur d'environnement* par analogie au *moteurs physiques*

[11]À notre avis, cette définition n'est pas toujours vraie pour les environnements simples et non-complexes, raison pour laquelle nous avons insisté sur le fait que notre approche convient à : environnements complexes. Dans les contributions de Russell et Norvig [Russell and Norvig, 2016], il parle d'un environnement entièrement observable et partiellement observable. Nous pouvons prendre les deux exemples cités dans [Russell and Norvig, 2016] : 1) le taxi automatique qui est confronté à un environnement complexe et donc partiellement observable et 2) joueur d'échecs qui a une vue complète de son environnement observable.

5.7 Implémentation et résultats (gAgent)

Dans cette section nous allons décrire brièvement l'implémentation (sous forme d'un framework que nous avons appelé **gAgent**).

Au cours des dernières décennies, de nombreuses architectures, conceptions et modèles multi-agents ont vu le jour. Cette prolifération est un signe d'un grand intérêt pour les systèmes multi-agents. Cependant, dans de nombreux cas, les propositions sont conceptuelles et ne sont pas appuyées par des mises en œuvre pour les valider. Ce manque d'implémentation est dû à la difficulté de produire des systèmes multi-agents en raison de la complexité des concepts sous-jacents (coordination, interaction, organisation, etc.). Cette complexité rend la majorité des systèmes existants difficile à utiliser et pratiquement impossible à utiliser par des non-spécialistes des systèmes multi-agents.

Comme le montre notre brève analyse, nous devons surmonter deux problèmes : 1) présenter une réalisation pratique permettant de valider notre modèle multi-agents et 2) présenter une mise en œuvre facile de l'utilisation. Afin de proposer un cadre conforme à la norme et facile à utiliser pour la communauté SMA, nous avons choisi de mettre en œuvre la norme FIPA dans la mesure du possible. Actuellement, FIPA est limité dans sa capacité à décrire l'architecture du système dans son ensemble, la structure des agents et la communication entre les agents, etc. Comme nous l'avons déjà mentionné plus haut, il est malheureusement difficile de trouver le concept d'environnement. Dans FIPA. De ce fait, en ce qui concerne la mise en œuvre de l'environnement, nous n'avons respecté aucune norme. À l'heure actuelle, la communauté scientifique utilise beaucoup la plate-forme multi-agents JADE, et nous nous sommes efforcés de nous en inspirer autant que possible, toujours dans le but de proposer un cadre convivial, sans toutefois rendre la tâche difficile. Pour les utilisateurs avec ce nouvel outil.

La norme FIPA propose un modèle de référence pour les plates-formes multi-agents. Elle propose une architecture générale (voir image 5.9) pour laquelle elle nécessite l'existence d'un certain nombre d'agents spécialisés :

- Agent Management System (AMS) : agent qui exerce le contrôle de supervision sur l'accès et l'utilisation de la plateforme ; ils sont responsables de l'authentification de l'agent résident et du contrôle des enregistrements. Agent
- Communication Channel (ACC) : agent qui fournit la route pour les interactions de base entre les agents entrant et sortant de la plateforme ; c'est la méthode de communication implicite qui offre un service fiable et précis pour les messages de routage.
- Directory Facilitator (DF) : agent qui fournit un service de page jaune à la plate-forme multi-agents.

La norme spécifie également le langage de communication de l'agent (FIPA-ACL). La communication d'agent est basée sur l'envoi de messages. Il convient de noter qu'il n'y a aucune restriction sur la technologie utilisée pour la mise en œuvre de la plate-forme ; par exemple, pour la communication, le système peut utiliser la messagerie électronique, CORBA, etc. Pour notre infrastructure, nous avons utilisé le composant CORBA pour l'échange de plateforme à file d'attente offert par le système d'exploitation pour la communication de plateforme interne.

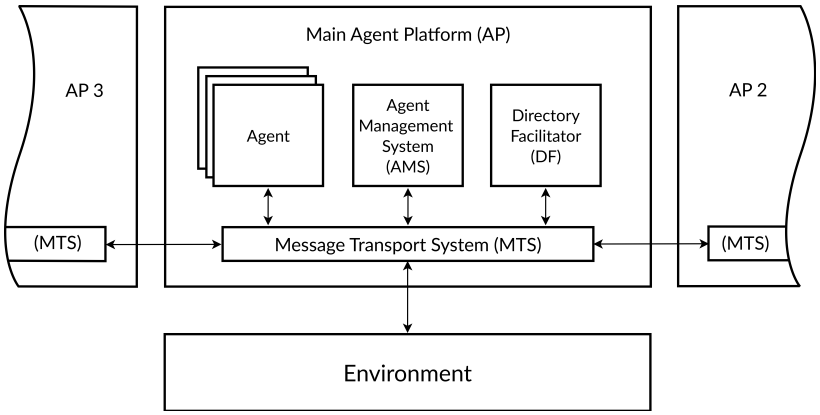


FIGURE 5.9 : Plateforme conforme FIPA à laquelle nous avons ajouté le module d'environnement ; la communication avec le module environnement et la plateforme s'effectue par échange de messages.

Afin d'intégrer notre contribution (environnement) dans une architecture FIPA, nous avons choisi d'utiliser le même système de messagerie pour assurer l'échange de données entre l'environnement et les agents. La Figure 5.9 montre comment l'environnement s'intègre à l'environnement. Architecture multi-agents FIPA.

Depuis le début, nous voulions fournir une architecture facile à utiliser sous la forme d'un framework. Nous avons donc fourni un ensemble de classes sous forme de bibliothèque C++ dynamique (voir Figure 5.10 ; les classes en gris sont les classes qui composent le cadre). Les utilisateurs doivent utiliser cette bibliothèque pour créer des agents et l'environnement (voir Figure 5.10, classes en blanc que l'utilisateur doit surcharger). Le framework offre également une interface graphique permettant à l'utilisateur de choisir les attributs à afficher afin de disposer d'un outil graphique, parfois utile, notamment dans la simulation et l'analyse de phénomènes complexes. Nous avons également intégré certaines fonctionnalités utiles pour VICA, ils consistent en un mécanisme permettant de créer un instantané et la possibilité de revenir à des états précédents.

Nous fournissons une implémentation de la solution décrite ci-dessus dans le but de simplifier la mise en œuvre SMA située. La solution est fournie sous forme d'un framework écrit en C++ et est disponible sous la licence GPL Ici : https://github.com/hdd-robot/gAgent.

5.8 Expérimentation

Pour expérimenter le planificateur local, nous avons utilisé le simulateur *Gazebo*. Il permet de disposer différents objets sur la surface de simulation. *Gazebo* dispose d'une base de données des objets limité ne permettant pas de créer un environnement d'expérimentation adapté à notre situation. Pour cela, nous avons utilisé une base de données d'objets *3DGEMS*[12] crée expressément pour *Gazebo* (voir Figure 5.11) dans le cadre d'un projet de recherche permettant de déterminer un ensemble de couleur qui améliore la détection [Rasouli and Tsotsos, 2017].

[12] http://data.nvision2.eecs.yorku.ca/3DGEMS/

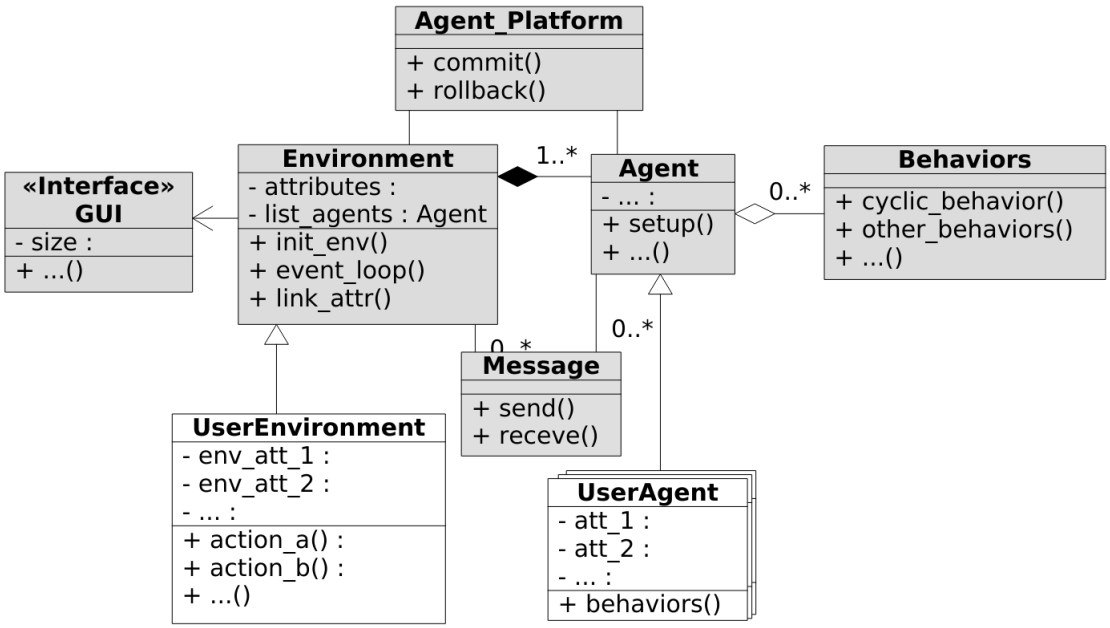


FIGURE 5.10 : Cette Figure illustre le diagramme de classes général de la plate-forme SMA ; en gris l'ensemble des classes qui constitue le framework, en blanc les des classes que l'utilisateur doit ajouter pour implémenter un environnement des agents.

FIGURE 5.11 : Base de données d'objets 3DGEMS.

L'utilisation 3DGEMS a permet d'améliorer grandement l'efficacité du système de reconnaissance dans *Gazebo* comme le montre la Figure 5.12.

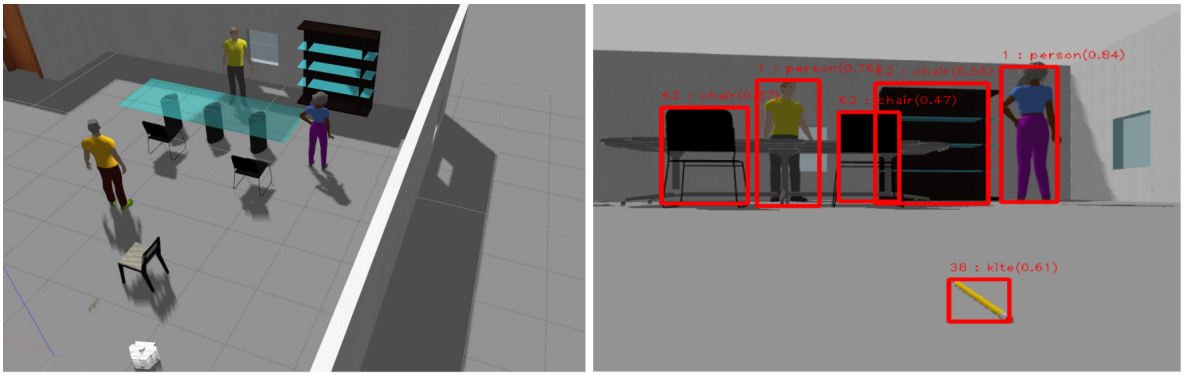


FIGURE 5.12 : Utilisation du système de reconnaissance dans *Gazebo*.

Malheureusement, la plupart des objets sont statiques et ne peuvent être déplacés par le robot car *Gazebo* à besoin des informations sur la masse des objets, leur matrice d'inertie, le coefficient de frottement, etc. Or, le *dataset 3DGEMS* n'a pas été créé dans le but de bouger les objets, donc ces informations sont manquantes. Pour cela, nous avons ajouté ces informations pour un grand nombre d'objets de ce *dataset*. Pour déterminer les informations manquantes, nous avons utilisé le logiciel *meshlab*. Il permet de calculer la masse et la matrice d'inertie à partir du modèle $3D$ des objets.

Le robot que nous avons mis en œuvre dans le cadre de cette thèse mesure 26 *cm* de hauteur, il est évident qu'il est incapable de bouger des obstacles qu'on trouve habituellement dans un environnement domiciliaire tel que des chaises par exemple. Pour cela, nous avons aussi réduit la taille des objets de 70 % pour les besoins de simulation.

La figure 5.13 montre un exemple d'expérimentation du planificateur local sous l'environnement *Gazebo*. La figure (A) montre l'environnement domiciliaire créé à partir de différents objets extraits de ma librairie de modèles *3DGEMS* que nous avons adaptés pour faire des objets amovibles. La Figure (B) montre l'image produite par la caméra RGB et le système de reconnaissance d'image. Les objets reconnus sont entourés automatiquement d'un rectangle rouge. La figure (C) (image générée par *RViz*) montre le nuage de points capturé par la caméra de profondeur, les rectangles rouges ont été ajoutés pour montrer la transposition des rectangles issus du système de reconnaissance d'images qui servent à estimer la masse des objets. La figure (C) (image générée par *RViz*) montre la carte de l'environnement générée par le LIDAR.

Nous avons récemment ajouté une interface graphique à *gAgent* afin de visualiser la disposition des agents (voir Figure 5.14). Chaque agent est représenté sous forme d'un rectangle, l'agent cognitif (le robot) est représenté sous forme d'un cercle. La couleur représente le type de l'obstacle, rouge pour les obstacles interactifs, bleu pour les obstacles amovibles et noir pour les obstacles fixes. La Figure 5.14 montre l'état du système multi-agents correspondant à situation montrée à la Figure 5.13.

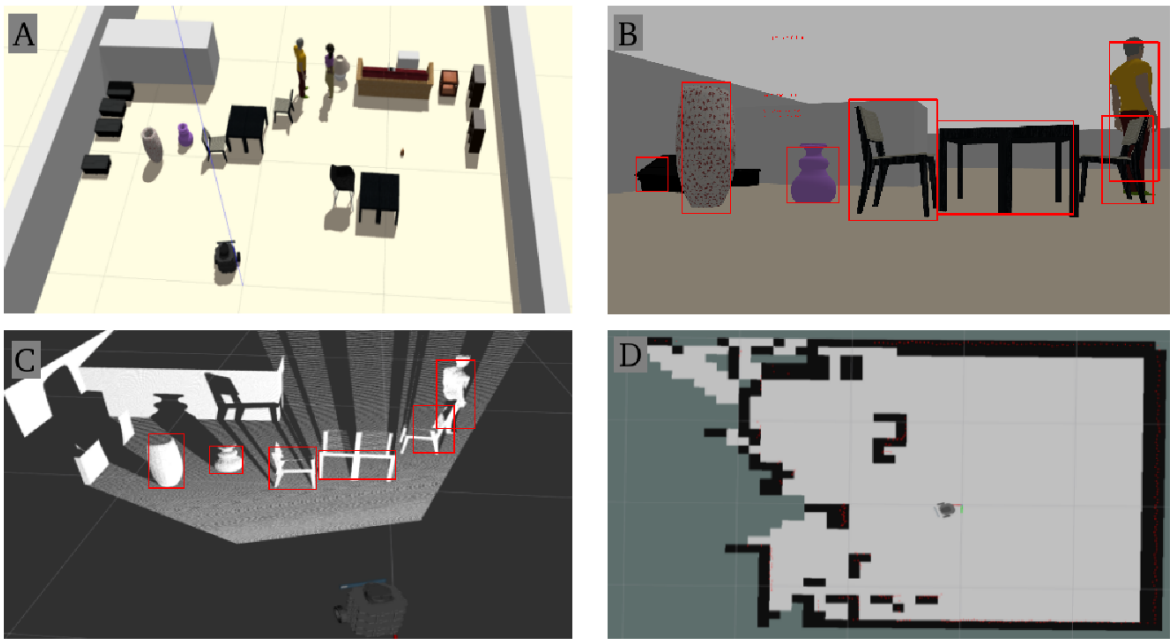


FIGURE 5.13 : Exemple d'utilisation du planificateur local.

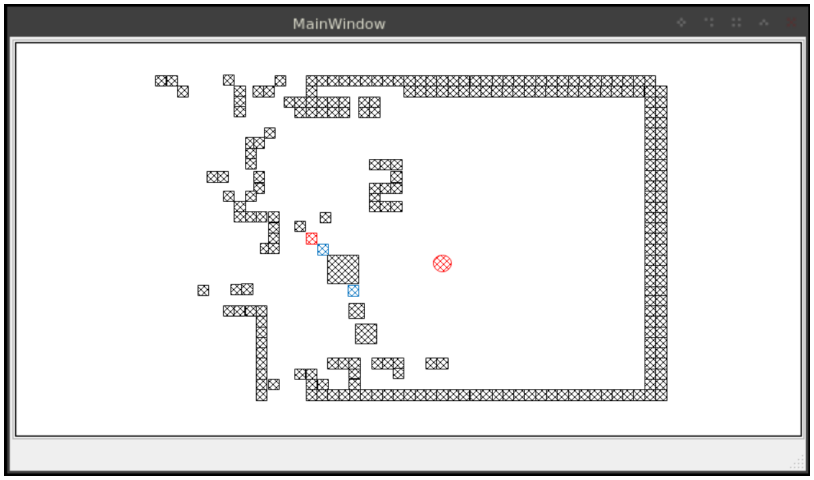


FIGURE 5.14 : Visualisation des agents.

Pour réaliser des expérimentations, il a fallu contourner plusieurs difficultés, par exemple réduire la taille des obstacles, modifier les caractéristiques les obstacles pour les rendre amovibles, réduire le bruit du capteur LIDAR, etc. Nous avons constaté aussi lors de ces expérimentations que les conditions de luminosité de *Gazibo* influent sur le système de reconnaissance d'obstacles ce qui affecte sensiblement les résultats. Nous pensons que pour expérimenter correctement le planificateur local, le système doit dépendre le moins possible des conditions d'environnement. Pour cela, nous avons effectué nos expérimentations avec uniquement des types d'obstacles les cylindres que nous avons considéré amovibles et des cubes que nous avons considéré comme étant des obstacles fixes comme illustré dans la Figure 5.15.

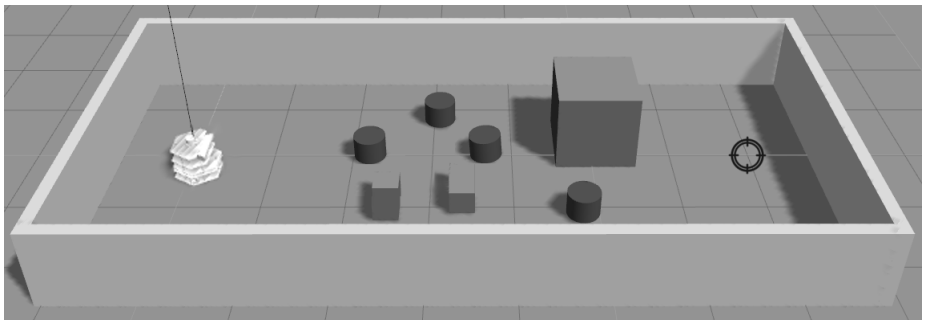


FIGURE 5.15 : Visualisation des agents.

Lors des simulations, nous avons comparé les distances obtenues avec notre solution et la distance la plus directe possible avec évitement d'obstacles calculée avec D^* *Lite*. Les résultats obtenus sont illustrés dans le tableau 5.8.

	VICA			D* Lite
Nb Obs.	Nbr Mouv.	Temp. (sec)	Dist. Parc. (cm)	Dist.
5	0	112	370	365
10	2	118	388	387
20	4	152	385	380
30	7	142	358	485
40	9	145	390	498
50	10	190	378	395
60	12	180	397	404
70	13	145	395	411

Tableau 5.3 : Résultats de simulations dans un environnement simplifié et comparaison avec D^* *Lite*.

5.9 Conclusion

L'utilisation du planificateur local a révélé beaucoup de difficultés lors des expérimentations. Parmi les difficultés majeures, on peut citer par exemple :

- Le simulateur *Gazebo* nous a permis de tester le planificateur dans différentes configurations de l'environnement ce qui est un atout. Malheureusement, la reconnaissance d'objets à partir de l'environnement de simulation est beaucoup moins efficace. On peut citer

quelques exemples : la chaise est quelques fois reconnue comme étant un avion, ce qui n'était pas un problème majeur vu que l'avion (jouet[13]) est enregistré comme étant un objet amovible. Pour citer un autre exemple, le vase (fragile) est reconnu comme étant un ours en peluche ces deux objets ne sont pas classés dans la même catégorie, l'un amovible et l'autre fixe.

- Nous avons pu réaliser des tests dans un environnement réel, mais la petite taille du robot et sa puissance limitée, engendre des problèmes de glissement. Les tests effectués se limitent à l'utilisation de jouets.

- La difficulté de trouver une plateforme SMA située, nous a contraint à développer par nous-même une plateforme adaptée. Cette plateforme est un projet important en terme de quantité de code, des bugs difficiles a identifier et a corriger sont apparus lors de l'exploitation.

Malgré les difficultés rencontrées, nous avons pu obtenir des résultats exploitables et prometteurs. Les scénarios expérimentés ont permis de confirmer note hypothèse de départ qui consiste à utiliser les systèmes multi-agents pour représenter et comprendre le contenu l'environnement plus efficacement. Mais nous pensons qu'il reste encore à perfectionner notre système pour l'adapter dans un environnement réel.

[13]Jouet : car improbable dans un environnement intérieur.

6

Conclusion et perspectives

La Navigation Parmi les Obstacles Amovibles (NAMO : Navigation Among Movable Obstacles) est une étape importante pour réussir à concevoir un robot capable de naviguer en milieu domiciliaire congestionné. Le but principal de ce domaine de recherche et de développer des robots de services pour l'aide à la personne. On peut citer quelques exemples d'utilisation : aide à la personne dans les structures accueillantes les personnes âgées et fragiles (Ehpad, hôpitaux, etc), robots aspirateurs dans des endroits encombrés, on peut même imaginer une utilisation dans des environnements hostiles pour l'homme tel la navigation en mieux radioactifs, etc. Bref, l'utilité de la NAMO ne fait aucun doute, c'est pour cela que des roboticiens s'y intéressent depuis plus de 20 ans. Malgré d'importants efforts fournis depuis ces deux décennies, les résultats actuels ne sont pas encore assez solides, pour prévoir une application en environnement réel. La difficulté majeure à laquelle se confrontent les roboticiens est triple : (1) La modélisation d'un environnement dynamique et congestionné sous forme d'une structure de données qui aura pour but de faciliter l'application d'algorithmes. (2) Trouver des trajectoires efficaces pour déplacer les obstacles en toute sécurité et (3) Trouver un compromis entre déplacer des obstacles ou les contourner, tout en considérant que l'environnement du robot est dynamique.

L'approche expérimentée dans cette thèse, consiste à s'inspirer des sciences cognitives, plus particulièrement du modèle de la simplixité proposé par Alain Berthoz pour une meilleure abstraction de l'environnement. L'implémentation de ce concept est faite à travers une architecture cognitive basée sur un système multi-agent. L'aspect d'abstraction n'est qu'une partie du système robotique, car il permet uniquement de fournir une représentation de l'environnement sous forme d'une structure de données. Les autres aspects (navigation, vision, etc.) sont implémentés dans une architecture robotiques.

Le travail proposé dans cette thèse se concentre essentiellement à la résolution du problème de la NAMO en environnement dynamique incertain et partiellement connu. Il propose une architecture robotique permettant à un robot mobile de se déplacer dans un environnement domiciliaire congestionné avec des obstacles amovibles. On peut dénombrer trois contributions apparentes : 1) Proposition et implémentation d'un planificateur global permettant de trouver

un chemin sans collision dans un environnement partiellement ou totalement inconnu. 2) Proposition et implémentation d'un planificateur local basé sur la simulation multi-agents permettant le choix optimal des obstacles à écarter en toute sécurité. 3) Proposition et implémentation d'une architecture robotique/cognitive permettant la perception, la modélisation de l'environnement, utilisation des deux planificateurs le premier global et le second local et finalement le choix de l'action.

Pour une NAMO efficace, il faut exécuter des trajectoires permettant d'atteindre l'objectif avec un minimum d'efforts, c'est-à-dire, il faut choisir en amont les trajectoires à suivre. Ce rôle est confié à un *pacificateur global*. Lors du déplacement, si le passage est obstrué avec des obstacles un seconde planificateur appelé *planificateur local* entre en jeu pour déplacer les obstacles en toute sécurité. Il est essentiel que ces deux planificateurs fournissent une estimation des actions à entreprendre avant l'exécution du mouvement, ce qui permet au robot de choisir le meilleur planificateur. Car parfois, il est plus efficace de contourner des obstacles au lieu d'essayer de déplacer de nombreux obstacles, quitte à parcourir une distance plus longe.

Le planificateur global a pour rôle de trouver des trajectoires dans les espaces libres, il est basé sur l'algorithme $H*$ que nous avons développé exprès pour ce planificateur. $H*$ est un algorithme qui combine (1) les techniques d'algorithmes inspirés d'insectes (*bug algorithms*) qui consiste à aller tout droit vers l'objectif et à suivre les contours des obstacles qui rencontre sur sa trajectoire, et (2) les algorithmes de recherche de chemins en utilisant des heuristiques. La combinaison de ces deux techniques permet de trouver rapidement des trajectoires et les plus droites possibles dans un environnement partiellement connu (voire même inconnu) tout en gardant un historique des positions parcourues afin d'optimiser les rebroussements de chemins en cas de blocage. L'utilisation des heuristiques permet aussi de donner une estimation du coût de déplacement au préalable. Cette estimation est utilisée pour le choix du planificateur. Les résultats obtenus lors des simulations ont montré que cet algorithme est de performance légèrement meilleur en terme de temps de calcul que $A*$ dans un environnement jusqu'à un remplissage 75% d'obstacles. En terme de distance parcourue, les deux algorithmes sont quasiment équivalents (très léger avantage pour $A*$) dans la plupart des situations (en écartant les situations ou les passages sont complètement obstrués). L'avantage principal de $H*$ par rapport à $A*$ est qu'il est capable de fonctionner dans un environnement complètement inconnu, comme dans un *bug algorithm* seule la direction de l'objectif est donnée. Sans cet algorithme, il aurait fallu utiliser des techniques telles que SLAM (cartographie et localisation simultanées). Or, l'utilisation des techniques SLAM ont montré des faiblesses dans les environnement dynamiques et/ou congestionnés.

Le planificateur local a pour objectif de gérer les obstacles, plus précisément, il permet d'atteindre une position donnée (un sous-objectif) en essayant d'écarter les obstacles par poussée en toute sécurité. Ce planificateur utilise un système, multi-agents pour réaliser une représentation de l'environnement. Grâce à un module de vision (basé sur une caméra RGB pour la reconnaissance des objets et une caméra de profondeur pour mesurer leur taille), le système représente chaque objet sous forme d'un agent réactif (ce processus est appelé agentification, il existe différents types d'agents (statiques, amovibles, interactifs), la génération se base des modèles (*templates*) prédéfinis pour chaque type). Le robot lui-même est représenté dans ce système multi-agents sous forme d'un agent cognitif. Le but du jeu étant de réaliser des simulations de déplacement de l'agent cognitif de la position de départ à l'objectif, afin de prédire de compor-

tement de l'environnement. Le planificateur local est capable de réaliser plusieurs simulations afin de choir la plus efficace. Lors de l'application de simulation choisie dans l'environnement réel, le système vérifie que les objets se comportent comme prévus dans la simulation, dans le cas contraire, le système s'arrête pour reprendre depuis le début. Le modèle de fonctionnement décrit ici, est inspiré du modèle proposé par Alain Berthoz du fonctionnement du cerveau pour gérer les mouvements.

L'utilisation d'un système multi-agents offre une flexibilité, car il est possible d'ajouter d'autres types d'agents pour une utilisation dans un autre domaine, il permet aussi d'améliorer le comportement de ces agents réactifs afin qu'il reproduise plus fidèlement le comportement des objets observés voire même adapter leur comportement selon les observations. Une alternative aux systèmes multi-agents serai d'utiliser un moteur physique, mais cette solution serai beaucoup trop rigide, car ce type de système se base sur des lois physiques qui ne permettent pas d'écart d'erreurs lors d'estimation de la position ou la taille des objets.

Pour réaliser une implémentation de ce modèle d'agent, nous avons été obligé de créer une plateforme multi-agents adaptée à nos besoins, car parmi les plateformes existantes, à notre connaissance, il n'existe pas de plateformes d'agents situés avec un environnement qui intègrent des lois programmables, on peut citer exemple une loi d'environnement qui empêche deux agents d'occuper la même position au même moment, un autre exemple serai de changer le coefficient de frottement du sol sur une partie de l'environnement (pour représenter un tapis de sol par exemple), un dernier exemple qui consiste à représenter un sol incliné ce qui rend le déplacement difficile dans un sens et plus facile dans l'autre, etc. Un second aspect important qui manque dans les plateformes existantes, serait la sauvegarde des simulations et la ré-initialisation. Nous proposons dans notre système que plusieurs simulations soient faites, puis retenir une seule de ces simulations pour l'appliquer à l'environnement. Il est important que lorsque des simulations sont réalisées, de revenir à l'état initial lors de l'application d'une simulation dans l'environnement réel et de ne retenir comme sauvegarde pérenne que les actions réellement effectuées sur l'environnement, pour servir de bases pour les prochaines simulations. Donc nous avons besoin d'une plateforme multi-agents capable de gérer les simulations sous forme d'une structure de données de type pile. Malgré nos efforts pour essayer d'adapter deux plateformes multi-agents à nos besoins, nous avons conclu qu'il est mieux et plus simple de proposer une nouvelle implémentation. Fort heureusement, de nombreux travaux théoriques existent qui traitent qui agents situés, malgré tout, nous avons apporté une contribution pour l'intégration de lois dans l'environnement.

Le planificateur local propose deux façons de déplacer les obstacles, (1) par poussée (exercer une force), par interaction (envoyer un signal sonore pour faire signe à l'obstacle de lui libérer le passage). Les résultats obtenus lors des simulations avec *gazebo* et tests avec le robot *Turtlebot 3*, de la gestion des obstacles par poussée, ont révélé une efficacité supérieure comparée aux solutions existantes. Dans la quasi-totalité des situations expérimentées, (1) le planificateur local assure la sécurité et ne déplacent que les obstacles amovibles, (2) le déplacement des obstacles pends en compte la force de poussée, si la force exercée nécessaire pour pousser l'objet dépasse celle prédite lors des simulations le déplacement est annulé pour éviter d'abîmer l'objet poussé ; les tests ont révélé que cette situation est souvent provoquée par existence d'un second objet caché derrière l'objet poussé qui empêche le déplacement. La seconde façon de gérer les obstacles (par interaction) apporte une nouvelle façon de gérer les obstacles, jusqu'à

présent, à notre connaissance, cette approche n'a pas été explorée. Nous avons montré que cette nouvelle approche de déplacer les obstacles apporte une nouvelle valeur ajoutée, mais il reste à améliorer son fonctionnement.

Les deux planificateurs sont assemblés dans une architecture robotique afin de combiner leur fonctionnement, pour ainsi proposer une solution complète pour la NAMO. Le choix du planificateur à appliquer à un moment donné est confié à une fonction de coût, elle-même se base sur les estimations de ces deux planificateurs pour un objectif donné. Pour réaliser cette tâche, l'architecture robotique implémente une boucle de contrôle qui enchaîne successivement les étapes (1) fixer un sous-objectif (2) demander des simulation pour atteindre cet objectif, (3) à choisir un planificateur vis la fonction de coût, (4) réaliser le mouvement via le planificateur choisi et (5) vérifier que le mouvement est réalisé conformément aux prédictions du planificateur choisi.

6.1 Perspectives

Les résultats obtenus lors des simulations, ont montré que notre système est tout à fait adapté à la NAMO. En comparaison avec les systèmes NAMO existants, le système proposé ici, permet une résolution efficace en adaptons le comportement du robot selon la situation. La dimension de sécurité n'a été abordée en NAMO que très rarement, or, c'est la sécurité doit être mise au centre de navigation dans les milieux fréquentés par les humains et les animaux de compagnie.

Les architectures robotiques basées sur la simulation ne sont pas nombreuses, nous faisons ici une expérimentation de cette solution et les résultats semblent concluants. Grâce à l'utilisation de la simulation comme outil de décision, ce genre d'architecture offre des perspectives diverse permettant d'améliorer le comportement du robot en NAMO. L'une des perspectives majeures est d'ajouter la dimension d'apprentissage lors des simulations notamment dans le planificateur local. C'est-à-dire les agents qui représentent les obstacles peuvent simuler plus fidèlement le comportement des objets réels au fur et à mesure de la navigation. On peut dire qu'au final ce que nous avons construit se réduit à un moteur physique sous forme d'un système multi-agent, avec l'avantage que des agent peuvent adapter leur comportement implémenter un système d'apprentissage pour une simulation plus fidèle.

Le système que nous avons présenté ici propose deux solutions pour écarter les obstacles, soit par poussée ou envoie de messages sonores pour signifier de lui céder le passage. Ces comportements doivent être améliorés pour une utilisation réelle en milieu domiciliaire. Il est tout à fait envisageable d'implémenter des solutions à base de bras articulés avec des grippes pour le déplacement des obstacles, d'ailleurs, c'est la solution choisie par de nombreuses études concourants montrée dans l'état de l'art de ce rapport. Le système de communication avec les obstacles interactifs peut lui aussi être amélioré en indiquant à l'interlocuteur la nouvelle position souhaitée à laquelle il doit se déplacer. Ici encore les obstacles interactifs peuvent être soit des humains ou d'autres robots avec lesquels le système peut entrer en communication. L'amélioration envisageable est que le robot puisse déterminer le type d'obstacle afin de déterminer le moyen de communication par exemple en mode sonore pour les humains et un protocole de communication adapté pour communiquer avec d'autres robots.

Bibliographie

[Abadi et al., 2016] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow : A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 265–283.

[Alami et al., 1998] Alami, R., Chatila, R., Fleury, S., Ghallab, M., and Ingrand, F. (1998). An architecture for autonomy. *The International Journal of Robotics Research*, 17(4) :315–337.

[Albus et al., 1987] Albus, J., McCain, H., and Lumia, R. (1987). Nasa/nbs standard reference model for telerobot control system architecture (nasrem), nbs tech. Technical report, Note 1235, Gaithersburg, MD.

[Albus, 2002] Albus, J. S. (2002). 4d/rcs : a reference model architecture for intelligent unmanned ground vehicles. In *Unmanned Ground Vehicle Technology IV*, volume 4715, pages 303–310. International Society for Optics and Photonics.

[Anderson, 2013] Anderson, J. R. (2013). *The architecture of cognition*. Psychology Press.

[Anderson et al., 1997] Anderson, J. R., Matessa, M., and Lebiere, C. (1997). ACT-R : A theory of higher level cognition and its relation to visual attention. *Human-Computer Interaction*, 12(4) :439–462.

[Arkin, 1998] Arkin, R. (1998). *Behavior-based robotics*. MIT press.

[Arkin, 1987] Arkin, R. C. (1987). *Towards cosmopolitan robots : Intelligent navigation in extended man-made environments*. PhD thesis, Georgia Institute of Technology.

[Arkin, 1989] Arkin, R. C. (1989). Motor schema—based mobile robot navigation. *The International journal of robotics research*, 8(4) :92–112.

[Arkin, 1995] Arkin, R. C. (1995). Reactive robotic systems.

[Baars, 1993] Baars, B. J. (1993). *A cognitive theory of consciousness*. Cambridge University Press.

[Baars, 2005] Baars, B. J. (2005). Global workspace theory of consciousness : toward a cognitive neuroscience of human experience. *Progress in brain research*, 150 :45–53.

[Bailey and Durrant-Whyte, 2006] Bailey, T. and Durrant-Whyte, H. (2006). Simultaneous localization and mapping (SLAM) : Part II. *IEEE Robotics & Automation Magazine*, 13(3) :108–117.

[Barbehenn, 1998] Barbehenn, M. (1998). A note on the complexity of Dijkstra's algorithm for graphs with weighted vertices. *IEEE transactions on computers*, 47(2) :263.

[Bellifemine et al., 1999] Bellifemine, F., Poggi, A., and Rimassa, G. (1999). JADE–A FIPA-compliant agent framework. In *Proceedings of PAAM*, volume 99, page 33. London.

[Bellifemine et al., 2007] Bellifemine, F. L., Caire, G., and Greenwood, D. (2007). *Developing multi-agent systems with JADE*, volume 7. John Wiley & Sons.

[Bergenti et al., 2006] Bergenti, F., Gleizes, M.-P., and Zambonelli, F. (2006). *Methodologies and software engineering for agent systems : the agent-oriented software engineering handbook*, volume 11. Springer Science & Business Media.

[Berthoz, 2009] Berthoz, A. (2009). *Simplexité (La)*. Odile Jacob.

[Berthoz, 2013] Berthoz, A. (2013). *La vicariance : le cerveau créateur de mondes*. Odile Jacob.

[Berthoz and Debru, 2015] Berthoz, A. and Debru, C. (2015). *Anticipation et prédiction : du geste au voyage mental*. Odile Jacob.

[Berthoz and Petit, 2014] Berthoz, A. and Petit, J.-L. (2014). *Complexité-Simplexité*. Collège de France.

[Borrego et al., 2018] Borrego, J., Dehban, A., Figueiredo, R., Moreno, P., Bernardino, A., and Santos-Victor, J. (2018). Applying domain randomization to synthetic data for object category detection. *arXiv preprint arXiv :1807.09834*.

[Braitenberg, 1986] Braitenberg, V. (1986). *Vehicles : Experiments in synthetic psychology*. MIT press.

[Bresciani et al., 2004] Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., and Mylopoulos, J. (2004). Tropos : An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, 8(3) :203–236.

[Briot and Demazeau, 2001] Briot, J.-P. and Demazeau, Y. (2001). *Principes et architecture des systèmes multi-agents*. Hermès-Lavoisier.

[Brooks, 1986] Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE journal on robotics and automation*, 2(1) :14–23.

[Cartwright and Collett, 1987] Cartwright, B. A. and Collett, T. S. (1987). Landmark maps for honeybees. *Biological cybernetics*, 57(1-2) :85–93.

[Castaman et al., 2016] Castaman, N., Tosello, E., and Pagello, E. (2016). A sampling-based tree planner for navigation among movable obstacles. In *Proceedings of ISR 2016 : 47st International Symposium on Robotics*, pages 1–8. VDE.

[Chatila, 2014] Chatila, R. (2014). Robotique et simplexité : modèles, architecture, décision et conscience. *A. Berthoz, & J. Petit, Complexité-Simplexité. Paris : Collège de France.*

[Chen et al., 2006] Chen, B., Cheng, H. H., and Palen, J. (2006). Mobile-C : a mobile agent platform for mobile C/C++ agents. *Software : Practice and Experience*, 36(15) :1711–1733.

[Chen and Hwang, 1990] Chen, P. C. and Hwang, Y. K. (1990). Practical path planning among movable obstacles. Technical report, Sandia National Labs., Albuquerque, NM (USA).

[Chen and Hwang, 1991] Chen, P. C. and Hwang, Y. K. (1991). Practical path planning among movable obstacles. In *Proceedings. 1991 IEEE International Conference on Robotics and Automation*, pages 444–449. IEEE.

[Cheng et al., 2006] Cheng, Y., Maimone, M. W., and Matthies, L. (2006). Visual odometry on the Mars exploration rovers-a tool to ensure accurate driving and science imaging. *IEEE Robotics & Automation Magazine*, 13(2) :54–62.

[Choi and Langley, 2018] Choi, D. and Langley, P. (2018). Evolution of the icarus cognitive architecture. *Cognitive Systems Research*, 48 :25–38.

[Chong et al., 2007] Chong, H.-Q., Tan, A.-H., and Ng, G.-W. (2007). Integrated cognitive architectures : a survey. *Artificial Intelligence Review*, 28(2) :103–130.

[Choset et al., 2005] Choset, H. M., Hutchinson, S., Lynch, K. M., Kantor, G., Burgard, W., Kavraki, L. E., and Thrun, S. (2005). *Principles of robot motion : theory, algorithms, and implementation*. MIT press.

[Cobzas and Zhang, 2001] Cobzas, D. and Zhang, H. (2001). Mobile robot localization using planar patches and a stereo panoramic model. In *vision interface*, pages 94–99. Citeseer.

[Comport et al., 2005] Comport, A. I., Marchand, E., and Chaumette, F. (2005). Efficient model-based tracking for robot vision. *Advanced Robotics*, 19(10) :1097–1113.

[Cousins, 2010] Cousins, S. (2010). Ros on the pr2 [ros topics]. *IEEE Robotics & Automation Magazine*, 17(3) :23–25.

[Coué et al., 2006] Coué, C., Pradalier, C., Laugier, C., Fraichard, T., and Bessière, P. (2006). Bayesian occupancy filtering for multitarget tracking : an automotive application. *The International Journal of Robotics Research*, 25(1) :19–30.

[Davies and Stone, 1995] Davies, M. and Stone, T. (1995). Folk psychology : The theory of mind debate.

[Demaine et al., 2000] Demaine, E. D., Demaine, M. L., and O'Rourke, J. (2000). PushPush and Push-1 are NP-hard in 2d. *arXiv preprint cs/0007021*.

[Demazeau, 2003] Demazeau, Y. (2003). Multi-Agent Systems Methodology, In Second Franco-Mexican School on Cooperative and Distributed Systems (LAFMI 2003).

[Djerroud and Cherif, 2016] Djerroud, H. and Cherif, A. A. (2016). Towards a computational model of consciousness : Global abstraction workspace. *International Journal of Cognitive and Language Sciences*, 11(1) :37–42.

[Djerroud and Cherif, 2018] Djerroud, H. and Cherif, A. A. (2018). Visualization tool for jade platform (jex). In *Proceedings of the Future Technologies Conference*, pages 481–489. Springer.

[Djerroud and Cherif, 2019] Djerroud, H. and Cherif, A. A. (2019). Environment engine for situated mas. In *ICAART (1)*, pages 129–137.

[Drogoul et al., 2013] Drogoul, A., Amouroux, E., Caillou, P., Gaudou, B., Grignard, A., Marilleau, N., Taillandier, P., Vavasseur, M., Vo, D.-A., and Zucker, J.-D. (2013). Gama : multi-level and complex environment for agent-based models and simulations. In *12th International Conference on Autonomous agents and multi-agent systems*, pages 2–p. Ifaamas.

[Duchoň et al., 2014] Duchoň, F., Babinec, A., Kajan, M., Beňo, P., Florek, M., Fico, T., and Jurišica, L. (2014). Path planning with modified a star algorithm for a mobile robot. *Procedia Engineering*, 96 :59–69.

[Elfes, 1989a] Elfes, A. (1989a). A tesselated probabilistic representation for spatial robot perception and navigation.

[Elfes, 1989b] Elfes, A. (1989b). Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6) :46–57.

[Elfes, 1991] Elfes, A. (1991). Occupancy grids : A probabilistic framework for robot perception and navigation.

[Faust et al., 2018] Faust, A., Oslund, K., Ramirez, O., Francis, A., Tapia, L., Fiser, M., and Davidson, J. (2018). Prm-rl : Long-range robotic navigation tasks by combining reinforcement learning and sampling-based planning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5113–5120. IEEE.

[Ferber, 1997] Ferber, J. (1997). Les systèmes multi-agents : un aperçu général. *Techniques et sciences informatiques*, 16(8).

[Ferber and Weiss, 1999] Ferber, J. and Weiss, G. (1999). *Multi-agent systems : an introduction to distributed artificial intelligence*, volume 1. Addison-Wesley Reading.

[Fipa, 2002] Fipa, A. (2002). Fipa acl message structure specification. *Foundation for Intelligent Physical Agents, http ://www. fipa. org/specs/fipa00061/SC00061G. html (30.6. 2004).*

[Friedlander and Franklin, 2008] Friedlander, D. and Franklin, S. (2008). LIDA and a Theory of Mind. *Frontiers in Artificial Intelligence and Applications*, 171 :137.

[Gat et al., 1998] Gat, E., Bonnasso, R. P., Murphy, R., et al. (1998). On three-layer architectures. *Artificial intelligence and mobile robots*, 195 :210.

[Gate, 2009] Gate, G. (2009). *Reliable perception of highly changing environments : implementations for car-to-pedestrian collision avoidance systems.* PhD thesis.

[Gaussier et al., 2000] Gaussier, P., Joulain, C., Banquet, J.-P., Leprêtre, S., and Revel, A. (2000). The visual homing problem : An example of robotics/biology cross fertilization. *Robotics and autonomous systems*, 30(1-2) :155–180.

[Ghallab, 2015] Ghallab, M. (2015). L'anticipation en robotique.

[Gourichon et al., 2002] Gourichon, S., Meyer, J.-A., and Pirim, P. (2002). Using coloured snapshots for short-range guidance in mobile robots. *International Journal of Robotics and Automation*, 17(4) :154–162.

[Guizzo and Ackerman, 2017] Guizzo, E. and Ackerman, E. (2017). The turtlebot3 teacher [resources_hands on]. *IEEE Spectrum*, 54(8) :19–20.

[Gutknecht and Ferber, 2000] Gutknecht, O. and Ferber, J. (2000). Madkit : a generic multi-agent platform. In *Proceedings of the fourth international conference on Autonomous agents*, pages 78–79. ACM.

[Himmelsbach et al., 2008] Himmelsbach, M., Mueller, A., Lüttel, T., and Wünsche, H.-J. (2008). Lidar-based 3d object perception. In *Proceedings of 1st international workshop on cognition for technical systems*, volume 1.

[Horiuchi and Noborio, 2001] Horiuchi, Y. and Noborio, H. (2001). Evaluation of path length made in sensor-based path-planning with the alternative following. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*, volume 2, pages 1728–1735. IEEE.

[Howden et al., 2001] Howden, N., Rönnquist, R., Hodgson, A., and Lucas, A. (2001). JACK intelligent agents-summary of an agent infrastructure. In *5th International conference on autonomous agents*.

[Hsu et al., 1999] Hsu, D., Latombe, J.-C., and Motwani, R. (1999). Path planning in expansive configuration spaces. *International Journal of Computational Geometry & Applications*, 9(04n05) :495–512.

[Ingrand, 2003] Ingrand, F. (2003). Architectures logicielles pour la robotique autonome. *JNRR*, 3 :8–10.

[Jones et al., 2014] Jones, H., Sabouret, N., and Youssef, A. B. (2014). Strategic Intentions based on an Affective Model and a simple Theory of Mind. In *Workshop Affects, Compagnons Artificiels et Interaction*.

[Kakiuchi et al., 2010] Kakiuchi, Y., Ueda, R., Kobayashi, K., Okada, K., and Inaba, M. (2010). Working with movable obstacles using on-line environment perception reconstruction using active sensing and color range sensor. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1696–1701. IEEE.

[Kam et al., 2015] Kam, H. R., Lee, S.-H., Park, T., and Kim, C.-H. (2015). Rviz : a toolkit for real domain data visualization. *Telecommunication Systems*, 60(2) :337–345.

[Kamon and Rivlin, 1997] Kamon, I. and Rivlin, E. (1997). Sensory-based motion planning with global proofs. *IEEE transactions on Robotics and Automation*, 13(6) :814–822.

[Katzfuss et al., 2016] Katzfuss, M., Stroud, J. R., and Wikle, C. K. (2016). Understanding the ensemble kalman filter. *The American Statistician*, 70(4) :350–357.

[Kavraki, 1994] Kavraki, L. (1994). *Random networks in configuration space for fast path planning*. Number 1535. stanford university.

[Kavraki et al., 1994] Kavraki, L., Svestka, P., and Overmars, M. H. (1994). *Probabilistic roadmaps for path planning in high-dimensional configuration spaces*, volume 1994. Unknown Publisher.

[Kavraki et al., 1998] Kavraki, L. E., Latombe, J.-C., Motwani, R., and Raghavan, P. (1998). Randomized query processing in robot path planning. *Journal of Computer and System Sciences*, 57(1) :50–60.

[Kavraki et al., 1996] Kavraki, L. E., Svestka, P., Latombe, J.-C., and Overmars, M. H. (1996). Probabilistic roadmaps for path planning in high-dimensional configuration spaces. *IEEE transactions on Robotics and Automation*, 12(4) :566–580.

[Kim et al., 2006] Kim, D., Sun, J., Oh, S. M., Rehg, J. M., and Bobick, A. F. (2006). Traversability classification using unsupervised on-line visual learning for outdoor robot navigation. In *Proceedings 2006 IEEE International Conference on Robotics and Automation, 2006. ICRA 2006.*, pages 518–525. IEEE.

[Koenig and Howard, 2004] Koenig, N. and Howard, A. (2004). Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE.

[Koenig and Likhachev, 2002] Koenig, S. and Likhachev, M. (2002). Dˆ* lite. *Aaai/iaai*, 15.

[Koenig et al., 2004] Koenig, S., Likhachev, M., and Furcy, D. (2004). Lifelong planning a. *Artificial Intelligence*, 155(1-2) :93–146.

[Kornhauser et al., 2007] Kornhauser, D., Rand, W., and Wilensky, U. (2007). Visualization tools for agent-based modeling in NetLogo. *Proceedings of Agent2007*, pages 15–17.

[Kruse et al., 2013] Kruse, T., Pandey, A. K., Alami, R., and Kirsch, A. (2013). Human-aware robot navigation : A survey. *Robotics and Autonomous Systems*, 61(12) :1726–1743.

[Kröse et al., 2001] Kröse, B. J., Vlassis, N., Bunschoten, R., and Motomura, Y. (2001). A probabilistic model for appearance-based robot localization. *Image and Vision Computing*, 19(6) :381–391.

[Kurup and Lebiere, 2012] Kurup, U. and Lebiere, C. (2012). What can cognitive architectures do for robotics ? *Biologically Inspired Cognitive Architectures*, 2 :88–99.

[Laird, 2009] Laird, J. E. (2009). Toward cognitive robotics. In *Unmanned Systems Technology XI*, volume 7332, page 73320Z. International Society for Optics and Photonics.

[Laird, 2012] Laird, J. E. (2012). *The Soar cognitive architecture*. MIT press.

[Laird et al., 2012] Laird, J. E., Kinkade, K. R., Mohan, S., and Xu, J. Z. (2012). Cognitive robotics using the soar cognitive architecture. In *Workshops at the twenty-sixth AAAI conference on artificial intelligence*.

[Lallée et al., 2012] Lallée, S., Pattacini, U., Lemaignan, S., Lenz, A., Melhuish, C., Natale, L., Skachek, S., Hamann, K., Steinwender, J., Sisbot, E. A., et al. (2012). Towards a platform-independent cooperative human robot interaction system : Iii an architecture for learning and executing actions and shared plans. *IEEE Transactions on Autonomous Mental Development*, 4(3) :239–253.

[Lambrinos et al., 2000] Lambrinos, D., Möller, R., Labhart, T., Pfeifer, R., and Wehner, R. (2000). A mobile robot employing insect strategies for navigation. *Robotics and Autonomous systems*, 30(1-2) :39–64.

[Lamiraux et al., 2004] Lamiraux, F., Bonnafous, D., and Lefebvre, O. (2004). Reactive path deformation for nonholonomic mobile robots. *IEEE transactions on robotics*, 20(6) :967–977.

[Langley et al., 2009] Langley, P., Laird, J. E., and Rogers, S. (2009). Cognitive architectures : Research issues and challenges. *Cognitive Systems Research*, 10(2) :141–160.

[Langley et al., 1991] Langley, P., McKusick, K. B., Allen, J. A., Iba, W. F., and Thompson, K. (1991). A design for the ICARUS architecture. *ACM SIGART Bulletin*, 2(4) :104–109.

[Latombe, 2012] Latombe, J.-C. (2012). *Robot motion planning*, volume 124. Springer Science & Business Media.

[Laumond, 2014] Laumond, J.-P. (2014). *Simplexité et Robotique*.

[LaValle, 1998] LaValle, S. M. (1998). Rapidly-exploring random trees : A new tool for path planning.

[LaValle, 2006] LaValle, S. M. (2006). *Planning algorithms*. Cambridge university press.

[LaValle and Kuffner Jr, 2000] LaValle, S. M. and Kuffner Jr, J. J. (2000). Rapidly-exploring random trees : Progress and prospects.

[Lemaignan et al., 2011] Lemaignan, S., Ros, R., Alami, R., and Beetz, M. (2011). What are you talking about ? grounding dialogue in a perspective-aware robotic architecture. In *2011 RO-MAN*, pages 107–112. IEEE.

[Levihn, 2011] Levihn, M. (2011). *Navigation among movable obstacles in unknown environments*. PhD Thesis, Georgia Institute of Technology.

[Levihn et al., 2013a] Levihn, M., Kaelbling, L. P., Lozano-Pérez, T., and Stilman, M. (2013a). Foresight and reconsideration in hierarchical planning and execution. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 224–231. IEEE.

[Levihn et al., 2013b] Levihn, M., Scholz, J., and Stilman, M. (2013b). Hierarchical decision theoretic planning for navigation among movable obstacles. In *Algorithmic Foundations of Robotics X*, pages 19–35. Springer.

[Levihn et al., 2013c] Levihn, M., Scholz, J., and Stilman, M. (2013c). Planning with movable obstacles in continuous environments with uncertain dynamics. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 3832–3838. IEEE.

[Levihn et al., 2014] Levihn, M., Stilman, M., and Christensen, H. (2014). Locally optimal navigation among movable obstacles in unknown environments. In *2014 IEEE-RAS International Conference on Humanoid Robots*, pages 86–91. IEEE.

[Levine, 2018] Levine, W. S. (2018). *The control handbook : control system applications*. CRC press.

[Levinson and Thrun, 2010] Levinson, J. and Thrun, S. (2010). Robust vehicle localization in urban environments using probabilistic maps. In *2010 IEEE International Conference on Robotics and Automation*, pages 4372–4378. IEEE.

[Levitt, 1990] Levitt, T. S. (1990). Qualitative navigation for mobile robots. *Int. J. Artificial Intelligence*, 44 :305–360.

[Likhachev et al., 2003] Likhachev, M., Gordon, G. J., and Thrun, S. (2003). Ara* : Anytime a* with provable bounds on sub-optimality. *Advances in neural information processing systems*, 16 :767–774.

[Lin et al., 2014] Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. (2014). Microsoft coco : Common objects in context. In *European conference on computer vision*, pages 740–755. Springer.

[Luke et al., 2005] Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., and Balan, G. (2005). Mason : A multiagent simulation environment. *Simulation*, 81(7) :517–527.

[Lumelsky and Stepanov, 1986] Lumelsky, V. and Stepanov, A. (1986). Dynamic path planning for a mobile automaton with limited information on the environment. *IEEE transactions on Automatic control*, 31(11) :1058–1063.

[Macal and North, 2008] Macal, C. M. and North, M. J. (2008). Agent-based modeling and simulation : ABMS examples. In *2008 Winter Simulation Conference*, pages 101–112. IEEE.

[Maes, 1989] Maes, P. (1989). The dynamics of action selection.

[McGuire et al., 2019] McGuire, K. N., de Croon, G., and Tuyls, K. (2019). A comparative study of bug algorithms for robot navigation. *Robotics and Autonomous Systems*, 121 :103261.

[Meng et al., 2018] Meng, Z., Sun, H., Teo, K. B., and Ang, M. H. (2018). Active path clearing navigation through environment reconfiguration in presence of movable obstacles. In *2018 IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, pages 156–163. IEEE.

[Moghaddam and Masehian, 2016] Moghaddam, S. K. and Masehian, E. (2016). Planning robot navigation among movable obstacles (namo) through a recursive approach. *Journal of Intelligent & Robotic Systems*, 83(3-4) :603–634.

[Montemerlo et al., 2002] Montemerlo, M., Thrun, S., Koller, D., Wegbreit, B., et al. (2002). Fastslam : A factored solution to the simultaneous localization and mapping problem. *Aaai/iaai*, 593598.

[Mueggler et al., 2014] Mueggler, E., Faessler, M., Fontana, F., and Scaramuzza, D. (2014). Aerial-guided navigation of a ground robot among movable obstacles. In *2014 IEEE International Symposium on Safety, Security, and Rescue Robotics (2014)*, pages 1–8. IEEE.

[Murphy, 2000] Murphy, R. (2000). *Introduction to AI robotics*. MIT press.

[Muscettola et al., 2002] Muscettola, N., Dorais, G. A., Fry, C., Levinson, R., Plaunt, C., and Clancy, D. (2002). Idea : Planning at the core of autonomous reactive agents.

[Ng and Bräunl, 2007] Ng, J. and Bräunl, T. (2007). Performance comparison of bug navigation algorithms. *Journal of Intelligent and Robotic Systems*, 50(1) :73–84.

[Nieuwenhuisen et al., 2008] Nieuwenhuisen, D., van der Stappen, A. F., and Overmars, M. H. (2008). An effective framework for path planning amidst movable obstacles. In *Algorithmic Foundation of Robotics VII*, pages 87–102. Springer.

[Nilsson, 1969] Nilsson, N. J. (1969). A mobile automaton : An application of artificial intelligence techniques. Technical report, SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INLIGENCE CENTER.

[Nistér et al., 2004] Nistér, D., Naroditsky, O., and Bergen, J. (2004). Visual odometry. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, volume 1, pages I–I. Ieee.

[Noborio et al., 2000] Noborio, H., Fujimura, K., and Horiuchi, Y. (2000). A comparative study of sensor-based path-planning algorithms in an unknown maze. In *Proceedings. 2000 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2000)(Cat. No. 00CH37113)*, volume 2, pages 909–916. IEEE.

[Nwana et al., 1999] Nwana, H. S., Ndumu, D. T., Lee, L. C., and Collis, J. C. (1999). ZEUS : a toolkit for building distributed multiagent systems. *Applied Artificial Intelligence*, 13(1-2) :129–185.

[Odell et al., 2003] Odell, J., Parunak, H. V. D., and Fleischer, M. (2003). Modeling agents and their environment : The communication environment. *Journal of Object Technology*, 2(3) :39–52.

[Okada et al., 2004] Okada, K., Haneda, A., Nakai, H., Inaba, M., and Inoue, H. (2004). Environment manipulation planner for humanoid robots using task graph that generates action sequence. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 2, pages 1174–1179. IEEE.

[Omicini, 2000] Omicini, A. (2000). SODA : Societies and infrastructures in the analysis and design of agent-based systems. In *International Workshop on Agent-Oriented Software Engineering*, pages 185–193. Springer.

[Ota, 2004] Ota, J. (2004). Rearrangement of multiple movable objects-integration of global and local planning methodology. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, volume 2, pages 1962–1967. IEEE.

[Padgham and Winikoff, 2002] Padgham, L. and Winikoff, M. (2002). Prometheus : A methodology for developing intelligent agents. In *International Workshop on Agent-Oriented Software Engineering*, pages 174–185. Springer.

[Paletta et al., 2001] Paletta, L., Frintrop, S., and Hertzberg, J. (2001). Robust localization using context in omnidirectional imaging. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*, volume 2, pages 2072–2077. IEEE.

[Parunak, 1997] Parunak, H. V. D. (1997). " Go to the ant" : Engineering principles from natural multi-agent systems. *Annals of Operations Research*, 75 :69–101.

[Pellier et al., 2018] Pellier, D., Adam, C., Johal, W., Fiorino, H., and Pesty, S. (2018). Une architecture cognitive et affective orienté interaction. *arXiv preprint arXiv :1810.10909*.

[PIPAME, 2012] PIPAME (2012). Le développement industriel futur de la robotique personnelle et de service en France.

[Quigley et al., 2009] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros : an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan.

[Quinlan and Khatib, 1993] Quinlan, S. and Khatib, O. (1993). Elastic bands : Connecting path planning and control. In *[1993] Proceedings IEEE International Conference on Robotics and Automation*, pages 802–807. IEEE.

[Ramamurthy et al., 2006] Ramamurthy, U., Baars, B. J., SK, D., and Franklin, S. (2006). LIDA : A working model of cognition.

[Rao and Georgeff, 1991] Rao, A. S. and Georgeff, M. P. (1991). Modeling rational agents within a BDI-architecture. *KR*, 91 :473–484.

[Rao et al., 1992] Rao, A. S., Georgeff, M. P., and Sonenberg, E. A. (1992). Social plans : A preliminary report. *Decentralized AI*, 3 :57–76.

[Rasouli and Tsotsos, 2017] Rasouli, A. and Tsotsos, J. K. (2017). The effect of color space selection on detectability and discriminability of colored objects. *arXiv preprint arXiv :1702.05421*.

[Réguigne-Khamassi and Doncieux, 2016] Réguigne-Khamassi, M. and Doncieux, S. (2016). Nouvelles approches en robotique cognitive. *Intellectica*, 65(1) :7–25.

[Remazeilles, 2004] Remazeilles, A. (2004). *Navigation à partir d'une mémoire d'images*. PhD Thesis, Université Rennes 1.

[Remazeilles and Chaumette, 2007] Remazeilles, A. and Chaumette, F. (2007). Image-based robot navigation from an image memory. *Robotics and Autonomous Systems*, 55(4) :345–356.

[Renault et al., 2019] Renault, B., Saraydaryan, J., and Simonin, O. (2019). Towards s-namo : Socially-aware navigation among movable obstacles. In *Robot World Cup*, pages 241–254. Springer.

[Rohmer et al., 2013] Rohmer, E., Singh, S. P., and Freese, M. (2013). V-rep : A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326. IEEE.

[Rosenblatt, 1997] Rosenblatt, J. K. (1997). Damn : A distributed architecture for mobile navigation. *Journal of Experimental & Theoretical Artificial Intelligence*, 9(2-3) :339–360.

[Russell and Norvig, 2016] Russell, S. J. and Norvig, P. (2016). *Artificial intelligence : a modern approach*. Malaysia ; Pearson Education Limited,.

[Sabah, 1990] Sabah, G. (1990). CARAMEL : a flexible model for interaction between the cognitive processes underlying natural language understanding. In *Proceedings of the 13th conference on Computational linguistics-Volume 3*, pages 446–448. Association for Computational Linguistics.

[Sabah and Briffault, 1993] Sabah, G. and Briffault, X. (1993). CARAMEL : A step towards reflection in natural language understanding systems. In *Proceedings of 1993 IEEE Conference on Tools with Al (TAI-93)*, pages 258–265. IEEE.

[Sankaranarayanan and Vidyasagar, 1990] Sankaranarayanan, A. and Vidyasagar, M. (1990). A new path planning algorithm for moving a point object amidst unknown obstacles in a plane. In *Proceedings., IEEE International Conference on Robotics and Automation*, pages 1930–1936. IEEE.

[Scholz et al., 2016] Scholz, J., Jindal, N., Levihn, M., Isbell, C. L., and Christensen, H. I. (2016). Navigation among movable obstacles with learned dynamic constraints. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3706–3713. IEEE.

[Schwartz and Sharir, 1983] Schwartz, J. T. and Sharir, M. (1983). On the "piano movers" problem. ii. general techniques for computing topological properties of real algebraic manifolds. *Advances in applied Mathematics*, 4(3) :298–351.

[Siegwart et al., 2011] Siegwart, R., Nourbakhsh, I. R., and Scaramuzza, D. (2011). *Introduction to autonomous mobile robots*. MIT press.

[Sim and Dudek, 1999] Sim, R. and Dudek, G. (1999). Learning visual landmarks for pose estimation. In *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No. 99CH36288C)*, volume 3, pages 1972–1978. IEEE.

[Stachniss et al.,] Stachniss, C., Leonard, J. J., and Thrun, S. Simultaneous localization and mapping. In *Springer Handbook of Robotics*.

[Stentz, 1997] Stentz, A. (1997). Optimal and efficient path planning for partially known environments. In *Intelligent unmanned ground vehicles*, pages 203–220. Springer.

[Steux and El Hamzaoui, 2010] Steux, B. and El Hamzaoui, O. (2010). tinyslam : A slam algorithm in less than 200 lines c-language program. In *2010 11th International Conference on Control Automation Robotics & Vision*, pages 1975–1979. IEEE.

[Stilman and Kuffner, 2008] Stilman, M. and Kuffner, J. (2008). Planning among movable obstacles with artificial constraints. *The International Journal of Robotics Research*, 27(11-12) :1295–1307.

[Stilman and Kuffner, 2005] Stilman, M. and Kuffner, J. J. (2005). Navigation among movable obstacles : Real-time reasoning in complex environments. *International Journal of Humanoid Robotics*, 2(04) :479–503.

[Stilman et al., 2007] Stilman, M., Nishiwaki, K., Kagami, S., and Kuffner, J. J. (2007). Planning and executing navigation among movable obstacles. *Advanced Robotics*, 21(14) :1617–1634.

[Suguri, 1999] Suguri, H. (1999). A standardization effort for agent technologies : The foundation for intelligent physical agents and its activities. In *hicss*, page 8061. IEEE.

[Sun et al., 2017] Sun, H., Meng, Z., and Ang, M. H. (2017). Semantic mapping and semantics-boosted navigation with path creation on a mobile robot. In *2017 IEEE International Conference on Cybernetics and Intelligent Systems (CIS) and IEEE Conference on Robotics, Automation and Mechatronics (RAM)*, pages 207–212. IEEE.

[Sun, 2001] Sun, R. (2001). *Duality of the mind : A bottom-up approach toward cognition.* Psychology Press.

[Sun, 2007] Sun, R. (2007). The motivational and metacognitive control in clarion. *Modeling integrated cognitive systems*, pages 63–75.

[Sun, 2016] Sun, R. (2016). *Anatomy of the mind : exploring psychological mechanisms and processes with the Clarion cognitive architecture.* Oxford University Press.

[Sycara et al., 2003] Sycara, K., Paolucci, M., Van Velsen, M., and Giampapa, J. (2003). The retsina mas infrastructure. *Autonomous agents and multi-agent systems*, 7(1-2) :29–48.

[Taillandier et al., 2012] Taillandier, P., Therond, O., Gaudou, B., et al. (2012). Une architecture d'agent bdi basée sur la théorie des fonctions de croyance : application à la simulation du comportement des agriculteurs. *Journées Francophones sur les Systèmes Multi-Agents 2012*, pages 107–116.

[Takahashi and Schilling, 1989] Takahashi, O. and Schilling, R. J. (1989). Motion planning in a plane using generalized Voronoi diagrams. *IEEE Transactions on robotics and automation*, 5(2) :143–150.

[Tessellations, 2000] Tessellations, S. (2000). *Concepts and Applications of Voronoi Diagrams.* Wiley.

[Thórisson and Helgasson, 2012] Thórisson, K. and Helgasson, H. (2012). Cognitive architectures and autonomy : A comparative review. *Journal of Artificial General Intelligence*, 3(2) :1–30.

[Thrun, 2002] Thrun, S. (2002). Particle filters in robotics. In *Proceedings of the Eighteenth conference on Uncertainty in artificial intelligence*, pages 511–518. Morgan Kaufmann Publishers Inc.

[Tisue and Wilensky, 2004] Tisue, S. and Wilensky, U. (2004). Netlogo : A simple environment for modeling complexity. In *International conference on complex systems*, volume 21, pages 16–21. Boston, MA.

[Trafton et al., 2013] Trafton, J. G., Hiatt, L. M., Harrison, A. M., Tamborello II, F. P., Khemlani, S. S., and Schultz, A. C. (2013). Act-r/e : An embodied cognitive architecture for human-robot interaction. *Journal of Human-Robot Interaction*, 2(1) :30–55.

[Trullier and Meyer, 1997] Trullier, O. and Meyer, J.-A. (1997). Biomimetic navigation models and strategies in animats. *AI communications*, 10(2) :79–92.

[Trullier et al., 1997] Trullier, O., Wiener, S. I., Berthoz, A., and Meyer, J.-A. (1997). Biologically based artificial navigation systems : Review and prospects. *Progress in neurobiology*, 51(5) :483–544.

[Van Den Berg et al., 2009] Van Den Berg, J., Stilman, M., Kuffner, J., Lin, M., and Manocha, D. (2009). Path planning among movable obstacles : a probabilistically complete approach. In *Algorithmic Foundation of Robotics VIII*, pages 599–614. Springer.

[Vernon et al., 2007a] Vernon, D., Metta, G., and Sandini, G. (2007a). The icub cognitive architecture : Interactive development in a humanoid robot. In *2007 IEEE 6th International Conference on Development and Learning*, pages 122–127. Ieee.

[Vernon et al., 2007b] Vernon, D., Metta, G., and Sandini, G. (2007b). A survey of artificial cognitive systems : Implications for the autonomous development of mental capabilities in computational agents. *IEEE transactions on evolutionary computation*, 11(2) :151–180.

[Warnier, 2012] Warnier, M. (2012). *Gestion des croyances de l'homme et du robot et architecture pour la planification et le contrôle de la tâche collaborative homme-robot*. PhD Thesis, INSA de Toulouse.

[Weyns and Michel, 2015] Weyns, D. and Michel, F. (2015). Agent environments for multi-agent systems–a research roadmap. In *Agent Environments for Multi-Agent Systems IV*, pages 3–21. Springer.

[Weyns et al., 2004] Weyns, D., Parunak, H. V. D., Michel, F., Holvoet, T., and Ferber, J. (2004). Environments for multiagent systems state-of-the-art and research challenges. In *International Workshop on Environments for Multi-Agent Systems*, pages 1–47. Springer.

[Weyns et al., 2015] Weyns, D., Parunak, V. D., Boissier, O., Michel, F., Schumacher, M., and Ricci, A. (2015). Agent environments for multi-agent systems. In *Agent Environments for Multi-Agent Systems IV : proceedings of the 4th International Workshop Environments for MAS (E4MAS) 2014*. 6 mai 2014.

[Wilfong, 1991] Wilfong, G. (1991). Motion planning in the presence of movable obstacles. *Annals of Mathematics and Artificial Intelligence*, 3(1) :131–150.

[Wolf et al., 2002] Wolf, J., Burgard, W., and Burkhardt, H. (2002). Robust vision-based localization for mobile robots using an image retrieval system based on invariant features. In *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No. 02CH37292)*, volume 1, pages 359–365. IEEE.

[Wooldridge, 2009] Wooldridge, M. (2009). *An introduction to multiagent systems*. John Wiley & Sons.

[Wooldridge and Jennings, 1995] Wooldridge, M. and Jennings, N. R. (1995). Intelligent agents : Theory and practice. *The knowledge engineering review*, 10(2) :115–152.

[Wu et al., 2010] Wu, H.-n., Levihn, M., and Stilman, M. (2010). Navigation among movable obstacles in unknown environments. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 1433–1438. IEEE.