Navigation de robot Mindstorm© dans un monde incertain

Bastien Thouverez Lucas Foulon

Février 2015

Résumé : Ce projet vise à faire naviguer un robot LEGO Mindstorm© dans un environnement inconnu, de lui faire trouver une position particulière, de lui faire mémoriser la partie découverte du monde et d'exploiter ces informations au sein de l'algorithme A* afin de lui faire retrouver sa position initiale.

Mots-clés : Monde incertain, navigation, modélisation, algorithme A*, LEGO Mindstorm©, programmation Java

1 Introduction

1.1 Contexte

Au cours de leur première année de Master informatique de l'UCBL, dans le cadre de l'UE MIF14 Bases d'Intelligence Artificielle, les étudiants travaillent en cours et en TD sur l'algorithme A-star (A*). Cet algorithme de recherche de chemins dans un graphe est un des algorithmes de planification le plus simple à mettre en place. Il a été créé pour que la première solution trouvée soit l'une des meilleures. Il est correct mais pas forcément optimal car il ne donne pas forcément la meilleure solution. Son atout est sa capacité à s'exécuter plus rapidement que d'autres algorithmes (comme par exemple l'algorithme de Dijkstra, qui est un cas spécial de A*). Ensuite les étudiants doivent mettre en pratique cet algorithme en l'insérant dans des robots Lego Mindstorm© lors des séances de TP.

Chaque robot est composé d'une brique programmable dotée d'un écran digital, de quelques boutons de navigation, et de multiples ports dont un USB 2.0 afin de relier la brique à un ordinateur, et de sept ports pour ajouter toutes sortes de composants au robot (moteur, capteur infrarouge, de couleur, de pression, etc.). Le robot est ensuite inséré dans un labyrinthe en 3 dimensions (cf. Figure 1), construit sur la base d'un repère orthonormé. Ce labyrinthe est donc composé de cases carrées ayant toutes la même taille et séparées par des murs, ainsi que d'une ou plusieurs sorties.

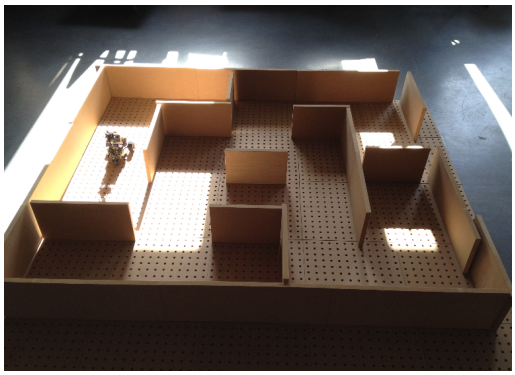


FIGURE 1 – Le labyrinthe

L'objectif des TPs MIF14 est de faire trouver la sortie au robot à l'aide d'un algorithme de recherche non informé, c'est-à-dire que le robot n'a pas de connaissances spécifiques de son environnement et utilise une stratégie très simple (par exemple essayer de sortir du labyrinthe en allant toujours à gauche dès qu'il le peut), puis de le faire revenir à sa position initiale. Pour cela, le robot utilise l'algorithme A*, en se basant sur le chemin qu'il aura parcouru pendant sa découverte du labyrinthe pour trouver une sortie, sachant qu'il n'a pas forcément parcouru la totalité du labyrinthe pour la trouver.

Le robot doit donc, dans un premier temps, modéliser son monde en construisant le repère orthonormé dont l'origine sera la position initiale du robot, et l'axe des abscisses (ou le Nord) sera représenté par la direction initiale du robot (cf. Figure 2).

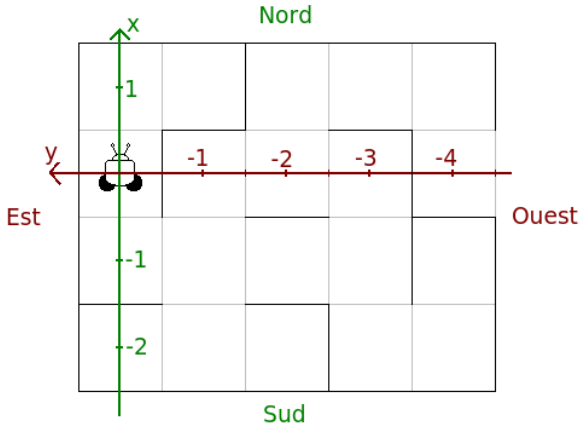


FIGURE 2 – Représentation actuelle de l'environnement

Pour réaliser ce projet, les étudiants disposent d'un code de base fourni. À l'aide de l'API leJOS NXJ [1], l'objectif est de coder en Java afin d'implémenter un algorithme pour

1. http ://www.lejos.org/nxj.php

sortir du labyrinthe (algorithme de recherche non informé ou autre, comme un algorithme de recherche en profondeur par exemple), puis l'algorithme A*. Il est ensuite demandé aux étudiants d'améliorer l'algorithme A*, qui n'est pas spécifique à ce problème en particulier, en lui ajoutant une ou plusieurs heuristiques. Une heuristique est une connaissance propre à un problème donné permettant de guider la recherche de solutions. Elle permet de spécialiser l'algorithme, pour permettre une optimisation du temps de recherche. Par exemple, pour son retour à sa position initiale, si sur son parcours le robot a le choix entre deux cases, une heuristique pourrait être de choisir la case qui, à vol d'oiseau, est la plus proche de la position initiale.

Dans le cadre de notre PRIM, la problématique se rapproche de celle étudiée dans l'UE MIF14, à une différence près : notre robot va désormais naviguer dans un monde incertain, c'est-à-dire que ce monde ne sera plus basé sur un repère orthonormé, mais sera quelconque. Avant notre robot savait qu'après avoir parcouru une certaine distance, il se trouvait dans une nouvelle case du repère. Ce n'est plus le cas ici, le robot n'a plus de notion de cases du repère et il devra être capable de découvrir son monde par lui-même. Il s'agira alors de réussir à modéliser ce monde non normé et de le mémoriser afin de permettre au robot de mettre en oeuvre l'algorithme A*.

1.2 Le robot et son environnement

Notre robot Lego Mindstorm© (cf. Figure 3) est équipé :
- d'une brique programmable ;
- de deux moteurs pour les roues à l'avant ;
- d'une roue arrière articulée ;
- d'un capteur ultrason.

Le labyrinthe dans lequel il évolue est plat, composé de murs perpendiculaires au sol formant des angles de 90° entre eux, il n'y a donc pas de murs obliques. Il est composé d'une ou plusieurs sorties modélisées par une distance infinie. Sa taille au départ est inconnue.

FIGURE 3 – Lego Mindstorm©

2 État de l'art

La première phase de ce projet consistait à faire un état de l'art relatif à notre sujet. En se basant sur les mots-clés associés à notre problématique, nous avons parcouru et étudié de nombreux documents, thèses et publications traitant de la robotique en général, de la navigation d'un robot et de la modélisation de l'environnement dans lequel il se trouve. La robotique est un sujet très vaste et en plein essor. De nombreuses avancées ont été faites et mises en œuvre dans des produits désormais disponibles à la vente, comme les aspirateurs robots ou encore les voitures autonomes. Cette popularité fait que nous avons trouvé une grosse quantité d'informations qu'il nous a fallu synthétiser. Nous présentons ci-dessous une synthèse des travaux liés à notre problématique en étudiant la navigation du robot et la découverte de son environnement.

Nous avons identifié deux types de navigation d'un robot [3] :

2.1 Navigation réactive

Dans cette configuration, le robot n'utilise que ses informations proprioceptives. Ces dernières sont les informations propres au robot et à sa configuration matérielle sans prendre en compte son environnement. Par exemple notre robot connaît le diamètre de ses roues et utilise un odomètre pour calculer des distances. Ces informations permettent de situer la position du robot dans l'espace mais se dégradent au cours du temps et deviennent de moins en moins précises. Par exemple, les rotations du robot ne sont pas précises au degré près, et après avoir enchaîné plusieurs rotations, cette imprécision rend les données moins fiables. On retrouve cette configuration pour la navigation vers un but [3] ou pour la navigation par évitement d'obstacle [5, 2].

2.2 Navigation par carte

Dans ce cas, le robot utilise ses informations proprioceptives pour construire une carte de son environnement : il peut par exemple calculer la distance à laquelle il se trouve d'un mur en mesurant le temps de trajet aller-retour d'une onde envoyée par son capteur ultrason. Ces informations, que le robot acquiert au cours de sa navigation, sont les informations dîtes extéroceptives. La navigation par carte se décompose en trois phases :

La cartographie : cette phase consiste à modéliser l'environnement sous une forme exploitable selon le problème donné. On distingue deux types de cartes : la carte topologique qui découpe l'environnement en zones et en liens entre des zones (cf. partie centrale de la figure 4), et la carte métrique qui cherche à représenter fidèlement l'environnement en calculant les distances entre les obstacles (cf. partie droite de la figure 4).

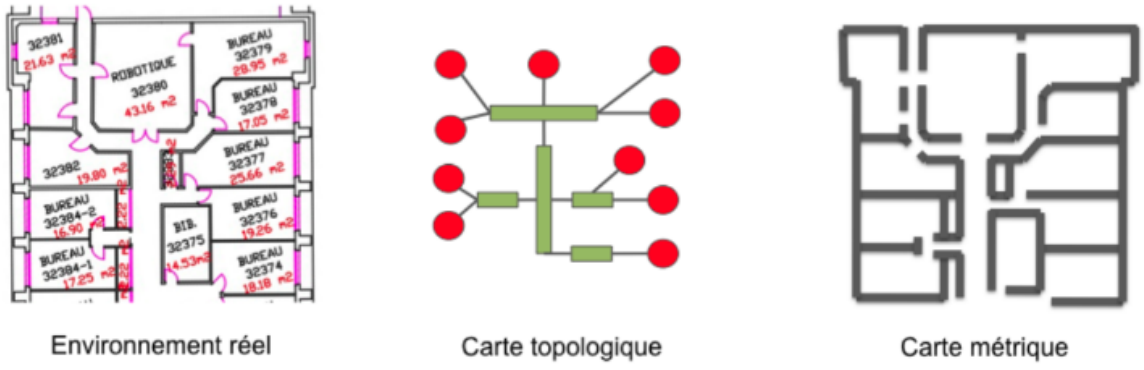


FIGURE 4 – Différentes représentations par carte [3]

La localisation : il s'agit ensuite au robot de se situer dans l'environnement qu'il a modélisé.

La planification : une fois qu'il connaît son environnement et qu'il sait où il se trouve, le robot peut entreprendre de rejoindre le but fixé par la problématique.

2.2.1 Les amers

Le terme "amer" est principalement utilisé pour la navigation maritime. Il représente un point de repère fixe et identifiable sans ambiguïté. Les amers facilitent la navigation et permettent de soulever les imprécisions crées par le mouvement du robot. Ils garantissent alors au mieux la cohérence de la carte de l'environnement. [3, 5, 1]

Pour cela, un amer doit être un point identifiable à de nombreux endroits différents, il doit être occulté le moins possible par d'autres objets, pour empêcher le plus possible qu'il soit confondu. Un amer est choisi suivant sa position mais aussi suivant sa taille. Plus il sera haut et plus il sera identifiable. Mais sa forme compte pour beaucoup, un objet qui s'avère être très haut mais pour lequel son extrémité sera horizontalement très étendue ne formerait pas un bon amer. La précision d'un amer dépendra donc aussi de l'étroitesse de son sommet.

2.2.2 Grille d'occupation

La grille d'occupation permet de partitionner l'environnement en un ensemble de cellules distinctes. Plus coûteux car beaucoup moins limité que le nombre d'amers, chaque cellule peut être rattachée à de nombreux attributs pour représenter les proprietés de la cellule. Le plus souvent, ils permettent d'indiquer le degré d'encombrement d'une cellule. [6, 4]
Par exemple, il est possible de représenter ce degré via un nombre compris entre 0 et 1, avec 0 un vide dont on est totalement sûr, et 1 une cellule occupée. Par défaut, toutes les cellules seraient alors initialisées à 0,5.

Le principal atout d'une grille d'occupation est sa capacité à représenter l'environnement de manière très dense suivant le pas de discréditation de la grille. La grille peut alors être utilisée dans des milieux de formes quelconques avec une estimation probabiliste ou statistique par rapport à la certitude des informations recueillies. De plus, la grille est facilement interprétable par l'homme.

Mais son point faible réside dans la quantité d'information récoltée lors de la découverte de l'environnement, difficilement compactable. Dans les environnements peu encombrés, par exemple, la grille s'adapte peu et enregistre beaucoup de données qui pourraient s'avérer inutiles.

2.2.3 Synthèse de l'état de l'art

Beaucoup de documents traitaient de la navigation et de la modélisation d'environnement souvent naturels et généralement accidentés. Pour notre problématique précise qui traite d'un terrain plat avec des murs droits, nous avons donc choisi d'exploiter les principes des approches présentées ci-dessus ainsi que nos acquis de MIF14 pour créer notre propre modélisation de l'environnement.

En particulier, les amers et la grille d'occupation nous ont beaucoup inspiré pour la modelisation. Nous verrons par la suite qu'il existe deux types de nœuds, et qu'ils sont crées lorsque l'on rencontre des obstacles.

3 Modélisation

3.1 Premières idées

Le choix de la navigation par carte était le plus approprié à notre problématique. Il nous fallait construire une carte, et plus précisément un graphe qui nous permettrait de mettre en œuvre l'algorithme A*. Les sommets de notre graphe seront des points du labyrinthe (case ou mur). Grâce au calcul des distances à l'aide de son odomètre et son capteur ultrason, le robot pourra calculer les distances séparant les sommets, ainsi nous pourrons mettre en forme un tel graphe. Le problème qui s'est vite posé était celui de synthétiser les informations relatives à la navigation du robot. Par exemple pendant sa navigation, le robot pose un sommet du graphe, et plus tard, il repasse très proche de ce sommet : il nous fallait savoir si nous devions synthétiser les informations à la fin du parcours du robot ou faire en sorte que le robot reconnaisse à la volée les sommets déjà parcourus. Ce problème s'applique aussi aux chemins empruntables de l'environnement modélisés par les arêtes du graphe.

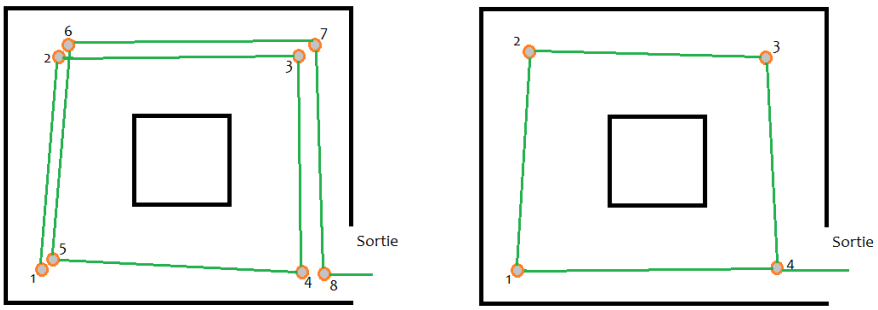


FIGURE 5 – Illustration du problème

Sur la figure 5, à gauche apparaît le résultat de la navigation du robot et à droite la modélisation optimale de l'environnement après synthèse des informations.

3.2 Notre modélisation

Nous avons choisi d'utiliser une carte topologique modélisée par un graphe non orienté pondéré sur ses arcs. Dans un premier temps, nous avons travaillé sur un labyrinthe droit, c'est-à-dire avec des murs toujours parallèles ou perpendiculaires entre eux. Étant donné ce fait, nous pouvons construire le graphe sur la base d'un repère orthonormé en utilisant comme unité, l'unité de mesure du robot lorsqu'il calcule les distances avec le capteur ultrason.

L'idée est de poser un sommet du graphe dès lors que le robot se trouve en face d'un mur, de modéliser le mur en posant un autre sommet du graphe, de poser des sommets aux intersections d'arêtes, puis alors de relier ces sommets par des arêtes lorsqu'un chemin existe entre deux sommets.

La figure 6 montre une représentation de l'environnement que pourrait avoir notre robot. Les traits noirs désignent les murs d'un labyrinthe.

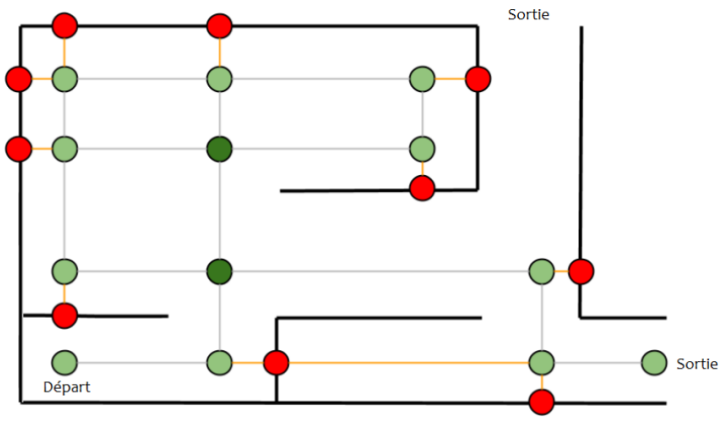


FIGURE 6 – Modélisation de l'environnement après navigation du robot

Lors de sa navigation, s'il détecte un mur, notre robot pose un sommet du graphe à sa position qui définit un point empruntable (cf. sommets vert clair) dans l'environnement, et un autre sommet en face de lui pour définir un mur (cf. sommets rouges), puis il tourne et repart jusqu'à trouver un autre mur et poser un autre sommet. Tous ces sommets sont reliés entre eux par des arêtes qui définissent un chemin libre de passage entre deux sommets empruntables (cf. arêtes grises), ou entre un sommet et un mur (cf. arêtes oranges).

Le second problème qui s'est posé était celui de détecter le croisement d'arêtes du graphe et d'y poser un sommet (cf. sommets vert foncé).

Pour mettre en place cette modélisation, nous avons choisi de développer un programme divisé en différentes parties détaillées ci-après. Un diagramme de classes est disponible en annexes.

3.2.1 Un robot

Notre robot est représenté par une classe Java qui hérite de la classe *DifferentialPilot* de l'API leJOS. Cette classe permet de contrôler notre robot de façon abstraite par rapport à sa configuration : par exemple, sans cette classe, pour faire avancer le robot, il faudrait dire aux deux moteurs de fonctionner en même temps à la même vitesse, ou pour tourner sur place, il faudrait dire à l'un des moteurs de tourner dans un sens, et à l'autre de tourner dans l'autre sens. Après être construit en spécifiant notre configuration du robot, l'objet *DifferentialPilot* est doté d'une fonction forward() et d'une autre méthode rotate() qui vont gérer pour l'utilisateur la synchronisation des moteurs pour effectuer ces actions. Notre robot a en plus pour attributs un *UltrasonicSensor* qui permet de manipuler son capteur ultrason, ainsi qu'un entier *direction* qui représente la direction en radian vers laquelle le robot se dirige.

3.2.2 Structure de graphe

C'est la structure principale que le robot utilise lors de sa navigation : il construit des Nœud à la volée et les relient par des Arête.

Chaque nœud et arête contiennent un numéro unique permettant leur identification. Un nœud est doté d'un type (mur ou case), de coordonnées x et y (cf. structure trigonométrique

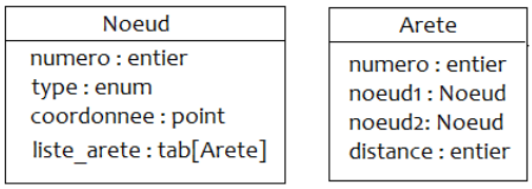


FIGURE 7 – Principaux attributs des classes du graphe

dans le paragraphe suivant), et d'une liste d'arêtes pour identifier ses voisins. Une arête contient les deux nœuds qu'elle relie, et la distance qui sépare ces nœuds. Le graphe contient simplement une liste de nœuds et une liste d'arêtes. Enfin c'est dans la classe Astar que s'opère le calcul de l'itinéraire retour du robot pour retrouver sa position initiale. Le robot connaît sa position courante quand il se trouve à une sortie du labyrinthe, ainsi que sa position initiale, au départ de sa navigation. Ces deux paramètres sont envoyés à une fonction findTheHomeWay qui, avec les informations mémorisées par le robot pendant sa navigation, calcule un plus court chemin de retour grâce à l'algorithme A*, puis renvoie la série de nœuds que doit emprunter le robot pour suivre ce court chemin de retour.

Cette structure nous permet alors de minimiser le nombre d'informations recueillies. Le robot créé un noeud seulement lorsqu'il change de direction. Une arête suffit lorsque le robot tient une trajectoire rectiligne.

Avant d'avancer, le robot se situe sur le dernier nœud qu'il a construit, et calcule la trajectoire qu'il va emprunter, c'est-à-dire qu'il va calculer une estimation de la future arête et du futur nœud qui vont être crées. Cela permet, pour la suite, en anticipant la trajectoire, d'éviter au robot de tourner en rond en fusionnant les segments. Il pourra, par exemple, s'arrêter s'il s'apprête à emprunter un chemin qu'il a déjà parcouru, ou encore fusionner les intersections des segments.

Le robot peut alors synthétiser les informations à la volée et mettre à jour le graphe au fur et à mesure qu'il circule dans son monde incertain.

3.2.3 Structure trigonométrique

Cette structure est composée de Point, de Droite et de Segment. L'idée est que chaque Nœud est associé à un point pour représenter ses coordonnées dans l'espace. Ainsi, avec deux points, nous pouvons obtenir une équation de droite associée à l'arête mais le robot ne traverse jamais toute l'équation de la droite. Ainsi nous pouvons créer un segment en définissant un intervalle sur cette droite. Cette structure nous permet donc de détecter les croisements entre chemins et d'apprécier la position de ces points de croisement.Elle nous permet aussi d'estimer les futures positions et trajectoires du robot. Ces fonctions sont détaillées ci-dessous.

Détermination de la future position. À partir de la dernière position du robot $P_{initial}$, de coordonnée $[x_{pInitial}, y_{pInitial}]$, comme étant la dernière position du robot, on peut estimer sa trajectoire et sa prochaine position à l'aide du sonar, noté $mesure_{sonar}$, et l'angle du robot dans le repère, noté α exprimé en radian (cf. cercle trigo en annexes) selon la formule suivante : $P_{final} = [x_{pInitial} + \sin(\alpha) \cdot mesure_{sonar}, y_{pInitial} + \cos(\alpha) \cdot mesure_{sonar}] = [x_{pFinal}, y_{pFinal}]$.

Cela permet au robot de connaître sa future position. Avec ces deux points, il pourra aussi anticiper la trajectoire qu'il va emprunter et peut-être trouver des croisements.

Détermination du chemin emprunté. Avec les positions de deux points, $P_1 = [x_1, y_1]$ et $P_2 = [x_2, y_2]$, on peut formuler une équation de droite, en soulevant deux exceptions, les équations de droites parallèles à l'axe des abscisses, et celle parallèles à l'axe des ordonnées : $a \cdot x + b \cdot y + c = 0$.

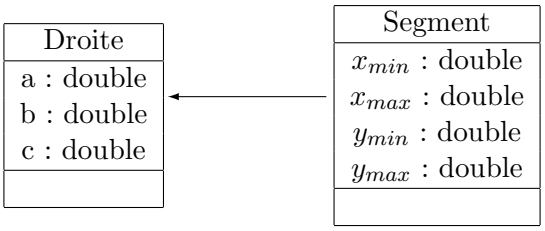
Dans le cas général, on pose : $b = -1$, $a = \frac{y_1 - y_2}{x_1 - x_2}$, et $c = y_1 - a \cdot x_1 = y_2 - a \cdot x_2$.

Ce qui nous donne une équation du type : $y = a \cdot x + c = \frac{y_1 - y_2}{x_1 - x_2} \cdot x + y_1 - a \cdot x_1$ (ou $y = \frac{y_1 - y_2}{x_1 - x_2} \cdot x + y_2 - a \cdot x_2$).

Pour le cas où la droite est parallèle à l'axe des abscisses tel que $y_1 = y_2$: $b = -1$, $a = 0$, et $c = y_1 = y_2$, et donc équation de droite de la forme $y = y_1 = y_2$.

Et le cas où la droite est parallèle à l'axe des ordonnées tel que $x_1 = x_2$: $b = 0$, $a = -1$, et $c = x_1 = x_2$, et donc équation de droite de la forme $x = x_1 = x_2$.

Bien entendu, le robot n'emprunte pas toute la droite d'équation $a \cdot x + b \cdot y + c = 0$, c'est pour cela que nous avons défini une classe *Segment* ***extends*** *Droite*



Cette classe Segment hérite de la classe Droite. Elle y ajoute deux points de l'espace implémentés avec quatre *double* pour modéliser l'intervalle du segment sur la droite. Avec cette structure de segment, nous pouvons dovrénavant calculer leur intersection pour poser un nœud du graphe.

Intersection de deux segments. On pose alors deux segments. Le premier est le chemin que le robot s'apprête à parcourir, et le second est un segment construit à partir d'une arête quelconque de la liste d'arêtes.

L'intersection de deux segments peut s'écrire de cette façon :

$$\begin{cases} y = a_1 \cdot x + b_1 & | \quad x \in [x_{1_{min}}, x_{1_{max}}] \text{ et } y \in [y_{1_{min}}, y_{1_{max}}] \\ y = a_2 \cdot x + b_2 & | \quad x \in [x_{2_{min}}, x_{2_{max}}] \text{ et } y \in [y_{2_{min}}, y_{2_{max}}] \end{cases}$$

Les coordonnées x_{min}, x_{max}, y_{min}, et y_{max} sont les coordonnées des Noeuds qui ont été utilisés pour construire les droites. Ce sont les intervalles des Segments parcouru ou que le robot s'apprête à parcourir.

Dans un premier temps, on compare les cœfficients directeurs. Si $a_1 = a_2$, alors les segments sont parallèles. On arrive dans un cas particulier, soit les segments ne peuvent jamais se croiser, soit les intervalles ont des valeurs en commun, alors les arêtes se superposent.

Dans un second temps, on compare donc les intervalles, c'est-à-dire si $[x_{1_{min}}, x_{1_{max}}] \cap [x_{2_{min}}, x_{2_{max}}] = \emptyset$ ou si $[y_{2_{min}}, y_{1_{max}}] \cap [y_{2_{min}}, y_{2_{max}}] = \emptyset$, on ne va pas plus loin car les segments ne se croisent pas dans les intervalles. Sinon on continue à appliquer l'algorithme. L'algorithme se trouve dans les annexes.

3.2.4 Une communication Android

Sachant que notre robot était capable de communiquer par Bluetooth, nous avons souhaité développer une application pour avoir un rendu en direct de la navigation du robot sur mobile ou tablette, et ainsi pouvoir sauvegarder ces données. Pour cela, nous sommes partis de l'application libre Android nxt-remote-control [2] qui permet de faire naviguer un robot, à laquelle nous avons ajouté ce dont nous avions besoin, à savoir afficher les informations prises par le robot pendant sa navigation.

La figure 8 montre une capture d'écran de l'application après communication des informations par le robot sur la construction du graphe de l'environnement. Une option accessible à partir de cet écran permet de sauvegarder ces données dans le téléphone et de pouvoir charger un fichier sauvegardé.

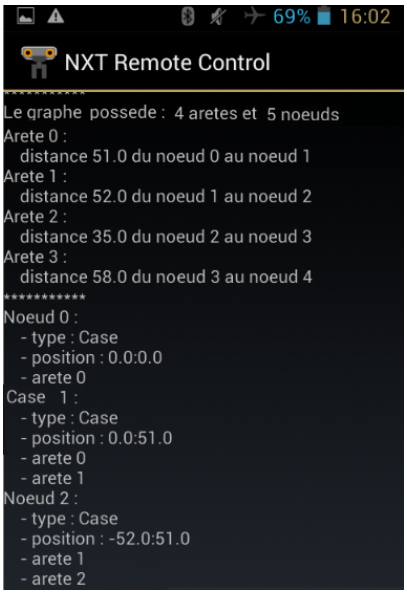


FIGURE 8 – Screenshot de l'application

3.3 Les limites de notre modélisation

La configuration de notre robot nous a imposé des limites dans la réponse à notre problématique. Dans un premier temps, la sortie du labyrinthe est modélisée par une distance infinie : en pratique, le capteur ultrason du robot possède une distance maximale D de détection d'objets, et lorsqu'il arrive en face d'une sortie, il ne détecte plus de murs à une distance inférieure à D, il sait donc qu'il a terminé sa navigation. Le problème est que si un couloir plus long que cette distance D se trouve dans le labyrinthe, en arrivant dedans, le robot ne détectera pas de mur car il sera trop loin pour son capteur, et il pensera être à la sortie alors qu'il n'en est rien. Une deuxième limite est celle de la mémoire du robot. Lors d'une longue navigation, la quantité d'informations à mémoriser est trop importante, et le traitement de ces informations pour mettre en oeuvre A* peut faire exploser la mémoire si il y a beaucoup de nœuds à explorer.

4 Conclusion

Nous avons assez bien respecté le planning prévu. Notre phase de recherche a peut-être été trop courte par rapport à la quantité importante d'informations trouvées sur le sujet, mais la rallonger ne nous aurait pas laissé assez de temps pour développer nos idées. À l'heure actuelle, toutes les fonctionnalités présentées en théorie ne sont pas encore implémentées.

Notre idée de développer une application Android à côté nous a semblé judicieuse tout au long du projet. Elle nous a beaucoup servi pour débugger notre programme, notamment lors de la détection de croisements entre chemins, mais passer du temps sur le développement Android était du temps que nous ne passions pas à développer la navigation du robot. Nous souhaitions, en plus de nous aider à débugger, avoir un rendu en direct de la construction du graphe sur mobile, mais le temps nous manquait pour développer une telle application.

Au départ du projet, nos contraintes ne spécifiaient pas que les murs formaient des angles droits entre eux, la présence de murs obliques pouvait être possible, seulement lors du développement de la navigation du robot, nous avons choisi de nous acquitter de cette

2. https ://code.google.com/p/nxt-remote-control/

contrainte : notre robot est prêt à prendre en compte des murs obliques avec son attribut direction, mais lors de sa navigation, il n'effectue que des rotations d'angle multiple de 90°.

Ce projet était très intéressant, et nous avions énormément d'idées, mais trop peu de temps. Ainsi, nous avons dû nous consacrer à une modélisation spécifique alors que nous aurions pu travailler sur beaucoup d'autres.

Références

[1] Stephane Betge-Brezetz. Modelisation incrementale et localisation par amers pour la navigation d'un robot mobile autonome en environnement naturel, 1996.

[2] Viviane Cadenat. Contribution a la navigation d'un robot mobile par commande referencee multi-capteurs, 2011.

[3] David Filliat. *Robotique mobile*. ENSTA PariTech, 2013.

[4] Haythem Ghazouani. Navigation visuelle de robots mobiles dans un environnement d'intérieur, 2012.

[5] Olivier Lefebvre. Navigation autonome sans collision pour robots mobiles nonholonomes, 2006.

[6] Julien Moras. Grilles de perception evidentielles pour la navigation robotique en milieu urbain, 2013.

5 Annexes

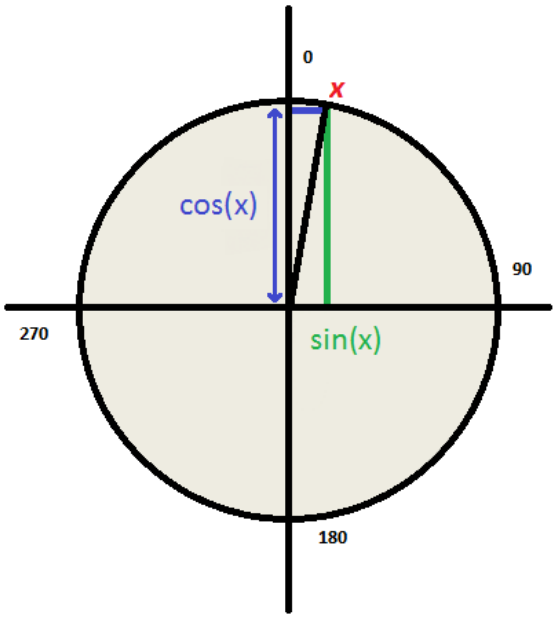


FIGURE 9 – Cercle trigonométrique adapté à notre modélidation

Algorithme 1 RECHERCHE INTERSECTION

Entrée(s) s_{liste} un segment de la liste, s_{final} le prochain segment parcouru par robot p_1, p_2 des Points

si $a_{s_{final}} == a_{s_{liste}}$ **alors**
  si $[x_{min_{sFinal}}, x_{max_{sFinal}}] \cap [x_{min_{sListe}}, x_{max_{sListe}}] \neq \emptyset$ **alors**
    //Les segments se superposent partiellement ou entièrement
    si $x_{min_{sFinal}} \in [x_{min_{sListe}}, x_{max_{sListe}}]$ **alors**
      $x_{p_1} = x_{min_{sFinal}}$
      $x_{p_2} = x_{max_{sListe}}$
    sinon
      $x_{p_1} = x_{max_{sFinal}}$
      $x_{p_2} = x_{min_{sListe}}$
    fin si
    si $y_{min_{sFinal}} \in [y_{min_{sListe}}, y_{max_{sListe}}]$ **alors**
      $y_{p_1} = y_{min_{sFinal}}$
      $y_{p_2} = y_{max_{sListe}}$
    sinon
      $y_{p_1} = y_{max_{sFinal}}$
      $y_{p_2} = y_{min_{sListe}}$
    fin si
  sinon
    $p_1 = p_2 = null$
  fin si
sinon si $[x_{min_{sFinal}}, x_{max_{sFinal}}] \cap [x_{min_{sListe}}, x_{max_{sListe}}] \neq \emptyset$
and $[y_{min_{sFinal}}, y_{max_{sFinal}}] \cap [y_{min_{sListe}}, y_{max_{sListe}}] \neq \emptyset$ **alors**
  si $b_{s_{final}} == 0$ **alors**
    $x_{p_1} = c_{s_{final}}$
    $y_{p_1} = a_{s_{liste}} \cdot x_{p_1} + c_{s_{liste}}$
  sinon si $b_{s_{liste}} == 0$ **alors**
    $x_{p_1} = c_{s_{liste}}$
    $y_{p_1} = a_{s_{final}} \cdot x_{p_1} + c_{s_{final}}$
  sinon
    $x_{p_1} = \frac{c_{s_{liste}} - c_{s_{final}}}{a_{s_{final}} - a_{s_{liste}}}$
    $y_{p_1} = a_{s_{final}} \cdot x_{p_1} + c_{s_{final}}$
  fin si
  $p_2 = null$
sinon
  $p_1 = p_2 = null$
fin si

Sortie(s) p_1, p_2 les points d'intersections des deux segments

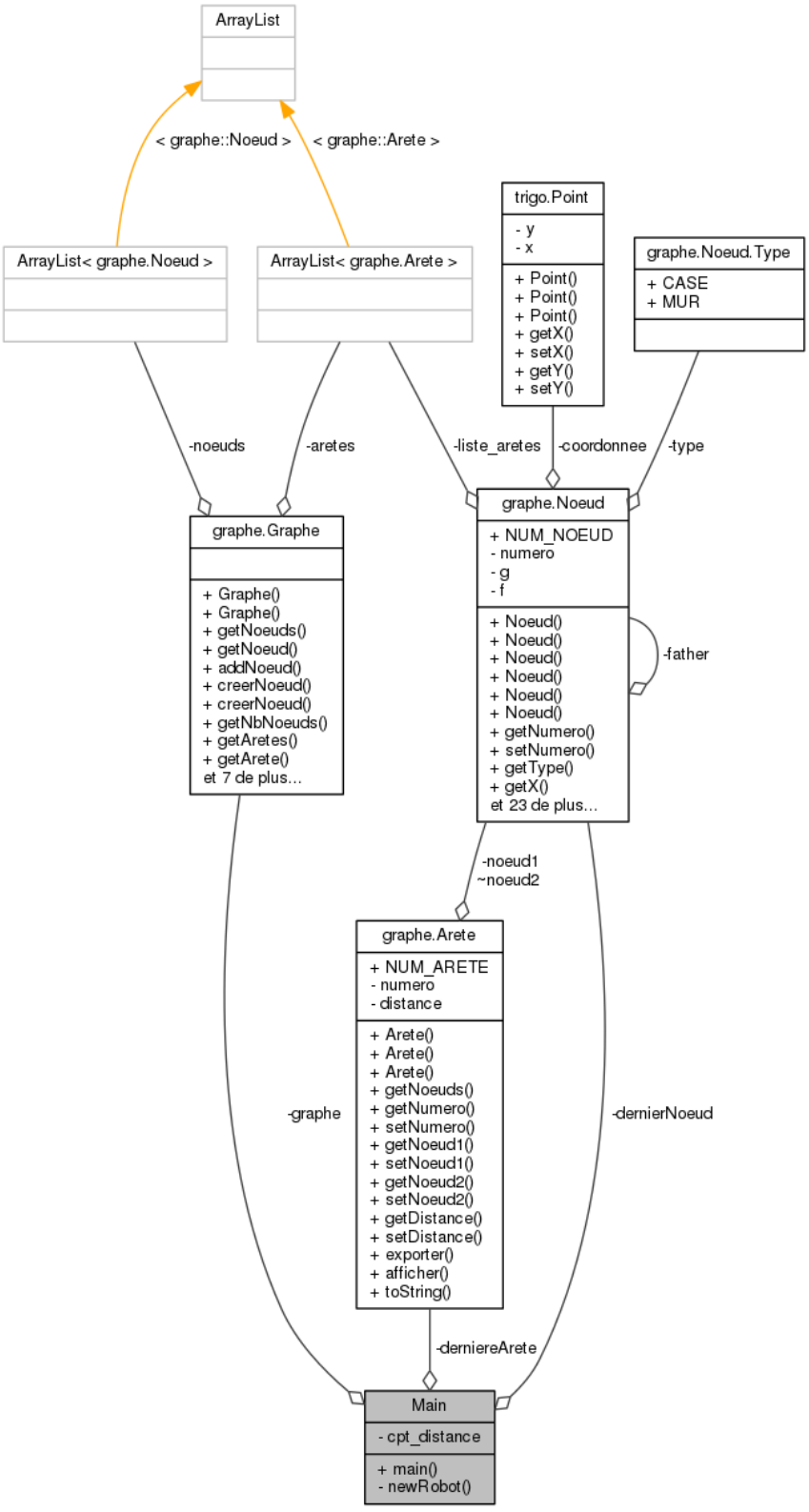


FIGURE 10 – Diagramme de classes de l'application