École Polytechnique de l'Université de Tours
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. +33 (0)2 47 36 14 14
www.polytech.univ-tours.fr

Département Informatique
4e année
2012 - 2013

Projet Robotique

Résolution d'un labyrinthe à l'aide d'un robot Lego Mindstorms

Encadrants
Pierre Gaucher
pierre.gaucher@univ-tours.fr
Jean-louis Bouquard
bouquard@univ-tours.fr

Étudiants
Fabien Buda
fabien.buda@etu.univ-tours.fr
Alexandre Lefillastre
alexandre.lefillastre@etu.univ-tours.fr

Université François-Rabelais, Tours

DI4 2012 - 2013

Version du 15 janvier 2013

Table des matières

1 Introduction **6**

2 Présentation de la plateforme Lego Mindstorms **7**
- 2.1 Lego NXT . . . 7
- 2.2 Les modules d'entrées et sorties . . . 8
  - 2.2.1 Modules de sortie . . . 8
  - 2.2.2 Modules d'entrée . . . 8
- 2.3 Programmation . . . 9

3 Notre robot **10**
- 3.1 Description du robot . . . 10
  - 3.1.1 Modules de sortie . . . 10
  - 3.1.2 Modules d'entrée . . . 10
- 3.2 Programmation et environnement de développement . . . 11

4 Structure de données et algorithmes employés **13**
- 4.1 Structure de données . . . 13
- 4.2 Algorithmes utilisés . . . 14
  - 4.2.1 Explorer l'environnement . . . 14
  - 4.2.2 Parcours en profondeur . . . 15
  - 4.2.3 Déplacements du robot . . . 17
  - 4.2.4 Recherche de case . . . 18
  - 4.2.5 Affichage du plan à l'écran . . . 18

5 Résultat **20**

6 Difficultés rencontrées **21**
- 6.1 Aléas divers . . . 21
- 6.2 Fiabilité du robot . . . 21

7 Évolutions envisagées **23**
- 7.1 Utiliser un calcul du plus court chemin . . . 23
- 7.2 Modifications sur le robot . . . 23
- 7.3 Programme de calibrage automatique . . . 24
- 7.4 Changer la structure de données . . . 24
- 7.5 Utiliser les communications Bluetooth . . . 24

8 Conclusion **26**

Table des figures

2.1 La brique Lego NXT . . . 7
2.2 Différents modules d'entrées et sorties . . . 8

3.1 L'IDE Eclipse et la console leJOS NXJ . . . 11

5.1 Exemple d'un labyrinthe . . . 20

6.1 Effets du manque de précision dans les mouvements . . . 22

7.1 Utilisation d'un calcul du plus court chemin . . . 23
7.2 Idée d'un programme de calibration : première étape . . . 24
7.3 Idée d'un programme de calibration : deuxième étape . . . 24

Listings

4.1 Fonctions pour explorer l'environnement 14
4.2 Algorithme de parcours en profondeur 15
4.3 Fonction pour gérer les déplacements 17
4.4 Fonction pour rechercher une case 18
4.5 Fonction pour gérer l'affichage du plan 18

Introduction

La résolution d'un labyrinthe à l'aide d'un Lego Mindstorms s'inscrit dans la continuité d'un projet de 3ème année. En effet, l'année dernière, Benjamin Allée avait développé un logiciel qui permettait de sortir d'un labyrinthe à l'aide de l'algorithme de parcours en profondeur. L'utilisateur positionnait graphiquement sa souris sur la case de départ et le logiciel gérait la résolution du labyrinthe, ainsi que l'affichage graphique permettant à l'utilisateur de suivre l'évolution du déplacement du robot.

Dans ce projet, nous allons mettre en pratique la résolution d'un labyrinthe : un robot de type Lego Mindstorms sera positionné dans un vrai labyrinthe. Il devra alors parcourir ce dernier afin de trouver la sortie, ou parcourir l'intégralité du labyrinthe s'il n'y a pas d'issue.

Le type de robot nous a été imposé, cependant les autres aspects, tels que la forme du robot construit, le mode de déplacement, et le langage de programmation utilisé étaient laissés à notre choix.

Dans ce rapport, nous présenterons le robot utilisé, l'algorithme et la structure de données utilisée pour parcourir le labyrinthe, les difficultés que nous avons rencontrées, et les évolutions que nous avons envisagées mais que nous n'avons pas été en mesure de mettre en place.

Présentation de la plateforme Lego Mindstorms

En 1998, la société Lego, sur la base de ses jeux pour enfants, crée la plateforme Lego Mindstorms. Elle se compose d'une "brique", qui contient le programme du robot et gère les moteurs et les différents capteurs. L'assemblage d'un robot se fait au moyen des composants de la gamme *Lego Technic*, permettant de créer tout type de robots (bras articulé, robot humanoïde ...). Enfin, la programmation peut être réalisée avec le logiciel fourni, prévu pour être facilement utilisable par les enfants.

Pour notre projet, nous emploierons la version actuelle de la brique, la version NXT. Une nouvelle version, la brique EV3 a été dévoilée par Lego le 09 Janvier 2013 pour une sortie dans le courant de l'année, avec notamment un matériel plus récent (plus de rapidité, plus de mémoire) et une connectivité élargie permettant par exemple le contrôle depuis les plateformes Android et iOS.

2.1 Lego NXT

FIGURE 2.1 – La brique Lego NXT

La brique NXT propose les entrées/sorties suivantes :

- **Sorties moteurs** : 3 ports de sortie notés A, B, C permettent de connecter et contrôler des moteurs.
- **Entrée accessoires** : 4 ports d'entrées notés 1, 2, 3 et 4 permettent le branchement de différents types de capteurs.
- **Port USB** : Un port USB permet le branchement à un ordinateur pour programmer la brique.
- **Haut-parleur** : Un haut-parleur permet d'émettre des sons.
- **Boutons** : Quatre boutons sont visibles sur la façade :
  - Le bouton *orange*, permet de confirmer un choix dans les menus.
  - Le bouton *gris foncé*, permet un retour arrière dans les menus.
  - Les boutons flèche *gauche* et *droite* permettent de naviguer dans les menus.

Les boutons peuvent être utilisés dans les programmes pour permettre une interaction avec l'utilisateur. Nous avons aussi remarqué qu'avec leJOS, un appui simultané sur les boutons *orange* et *gris foncé* provoquent un redémarrage de la brique, ce qui permet de sortir du programme si celui-ci ne peut pas se terminer de lui même.
- **Écran LCD** : Celui-ci permet l'affichage du menu de la brique (lancement d'un programme enregistré sans l'intervention d'un ordinateur), et permet aux programmes l'affichage de textes, ou d'images.
- **Bluetooth** : La brique dispose d'une connectivité sans fil Bluetooth permettant l'ajout de programmes et de fichiers sans passer par le câble USB. Sur les briques utilisant leJOS, le Bluetooth peut être utilisé pour faire communiquer plusieurs briques, ou communiquer avec un ordinateur pour proposer un contrôle à distance ou pour utiliser la puissance de calcul de l'ordinateur.

L'alimentation électrique de la brique NXT est assurée par des piles, ou une batterie disponible en option.

2.2 Les modules d'entrées et sorties

Figure 2.2 – Différents modules d'entrées et sorties

2.2.1 Modules de sortie

A ce jour, il n'existe qu'un seul module de sortie compatible : les servomoteurs fournis avec la brique NXT. Ces moteurs permettent les déplacements du robot, fonctionnent indépendamment les uns des autres avec des vitesses ajustables, disposent de contrôles pour autoriser ou bloquer la rotation des roues lorsque le moteur ne tourne pas, et sont équipées d'un *tachymètre* qui permet d'effectuer des mouvements précis en choisissant l'angle des rotations effectuées par les moteurs.

2.2.2 Modules d'entrée

Le Mindstorms NXT dispose de nombreux capteurs compatibles, car Lego permet à des sociétés tierces de créer des capteurs pour les Mindstorms. La liste ci dessous ne présente que les modules officiels.

- **Capteur de pression** : Permet de détecter la pression et le relâchement sur le capteur. Il peut notamment être utilisé pour détecter une collision avec un obstacle.
- **Capteur de pression sonore** : Permet de mesurer le niveau sonore en pourcentage. Par exemple, un niveau sonore de 5% correspond à une salle silencieuse. Entre 5 et 10 %, cela correspond à une conversation à une certaine distance. Entre 10 et 30%, cela équivaut à une conversation normale près du capteur ou à l'écoute de musique à puissance normale. Enfin, au dessus, le capteur reçoit l'équivalent du bruit dans une salle de concert.
- **Capteur de luminosité** : Permet de distinguer l'ombre et la lumière. Il permet aussi de mesurer l'intensité lumineuse d'une pièce ainsi que celle de surfaces colorées.
- **Capteur de couleurs** : Le capteur de couleurs permet la détection de six couleurs basiques différentes, il permet aussi l'émission d'une lumière rouge, verte ou bleue.
- **Capteur ultra-sonore** : Permet de mesurer la distance entre le robot et un obstacle en utilisant le principe de l'écho-localisation des chauve souris : le capteur émet un ultrason et mesure le temps avant de recevoir un écho. La distance mesurée est théoriquement comprise entre 0 et 255 cm avec une précision de plus ou moins 3 cm. Dans la pratique, en dessous de 8-10 cm les mesures sont fausses, et la portée maximale est aux alentours de 170 cm. A noter que si l'obstacle est trop proche pour que l'écho puisse revenir au capteur, le capteur renvoie une distance de 255 cm. Enfin, l'onde sonore se propage en forme de cone, donc l'obstacle détecté n'est pas toujours directement en face du capteur.

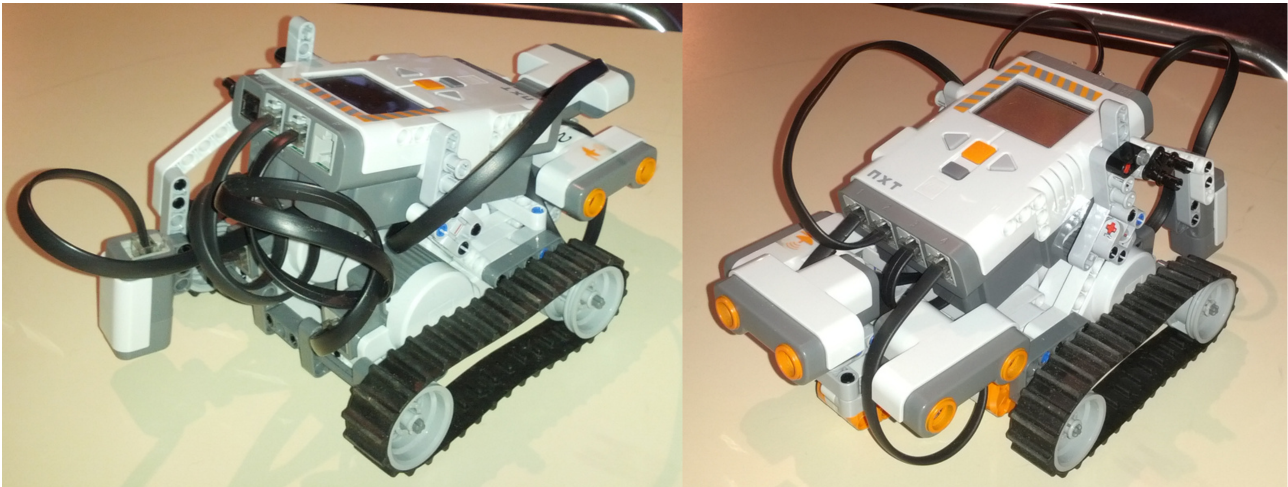
2.3 Programmation

Depuis leur création, les systèmes Mindstorms sont programmables via le logiciel fourni par la société Lego. Une interface visuelle permet de définir très rapidement les actions à réaliser par le robot. Mais, cet outil n'est pas pratique pour produire un programme complexe, car il permet seulement de réaliser des structures algorithmiques simples conditionnées directement par les valeurs envoyées par les capteurs, et ne permet pas d'utiliser la mémoire pour stocker des informations (la structure de données du labyrinthe par exemple).

Cependant, grâce à la facilité de modification ("Hacking") de la brique, de nombreux langages sont apparus pour pouvoir développer des applications. On peut citer, par exemple, le langage NXC (Not exactly C), proche du langage C, ainsi que leJOS qui est semblable au Java. Dans certains cas, une mise à jour de la brique est nécessaire.

Plus d'informations sur les facilités de programmation, sont disponibles à la page suivante : http://en.wikipedia.org/wiki/Lego_Mindstorms_NXT#Programming.

Notre robot



3.1 Description du robot

Le robot que nous avons construit est une version légèrement modifiée d'un des modèles de base indiqué dans le manuel de la boite Lego. Le modèle de base indiquait comment obtenir la structure du robot, mais ne comportait aucun capteur. Nous l'avons donc un peu modifié pour pouvoir accrocher les capteurs dont nous avions besoin.

3.1.1 Modules de sortie

Sur notre robot, nous avons employé deux moteurs : Le moteur de "gauche" est défini sur la sortie B et le moteur de "droite" sur la sortie C. Un troisième moteur est présent mais n'est pas utilisé lors des déplacements : Il permet de soutenir la structure du robot.

En ce qui concerne le déplacement, nous avons privilégié les chenilles : lors de tests, nous avions conclus que les roues ne permettaient pas vraiment de faire une rotation "sur place" (sans déplacer le robot). De plus, les chenilles semblaient mieux adhérer aux différents types de sols.

3.1.2 Modules d'entrée

En entrée, deux types de capteurs sont employés : ce sont des capteurs ultra-sonores et de couleur. Les capteurs ultra-sonores permettent de détecter la présence de murs. Ils sont au nombre de trois : le capteur ultra-sonore de gauche est relié sur l'entrée numéro 4. Celui de droite est positionné sur l'entrée numéro 1. Enfin le capteur présent à l'avant est disposé sur le port numéro 2.

Enfin, nous avons employé un capteur de couleur pour détecter la sortie du labyrinthe. A chaque déplacement sur une case (sauf en cas de sortie de la récursivité), le capteur essaye de détecter la présence d'un sol rouge. A ce moment là, le robot se bloque et lance une musique de victoire.

A noter que ce capteur est positionné à l'arrière du robot et qu'il est connecté à l'entrée numéro 3 du Mindstorms.

3.2 Programmation et environnement de développement

Le langage de programmation utilisé pour notre robot est leJOS, Java pour la plateforme NXT. La documentation disponible est assez abondante : le site internet dispose d'une JavaDoc (http://leJOS.sourceforge.net/p_technologies/nxt/nxj/api/index.html), ce qui permet de connaitre la définition des fonctions. De nombreux exemples de code sont aussi disponibles sur Internet, ce qui nous a permis de réaliser rapidement des opérations basiques telles que le déplacement ou l'utilisation des capteurs.

Par rapport à d'autres langages, la "brique" nécessite un flashage pour pouvoir être programmée en Java. Cela est possible via l'utilitaire NXJ Flash : on relie le câble USB de l'ordinateur vers le NXT et il suffit de suivre les instructions du logiciel. A noter qu'il faut impérativement un système Windows 32 bits, sans quoi il est impossible de réaliser cette opération.

Pour l'écriture du code, nous avons privilégié l'environnement de développement Eclipse (http://www.eclipse.org/). En effet, tous nos cours de programmation en Java se sont déroulés avec ce logiciel. De plus, leJOS est bien intégré dans l'IDE depuis la dernière version du langage. Mais il est aussi possible, d'utiliser NetBeans.

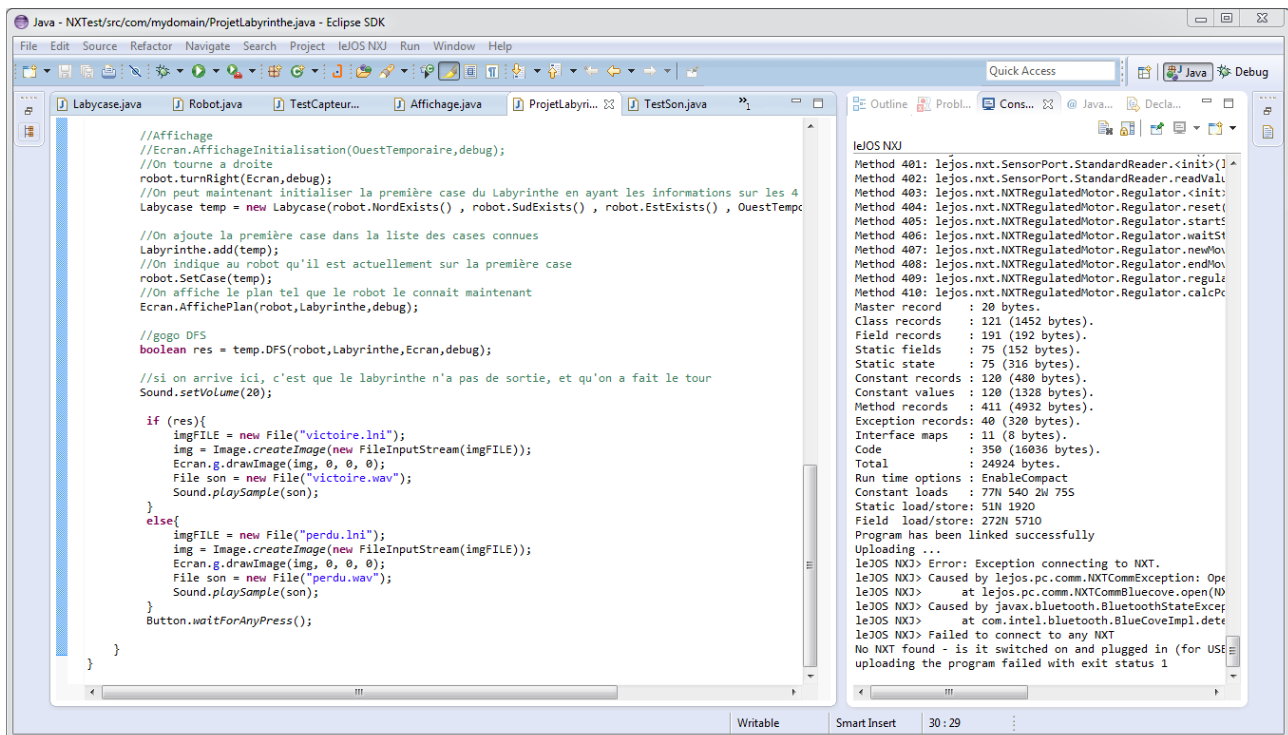


FIGURE 3.1 – L'IDE Eclipse et la console leJOS NXJ

La programmation, via l'IDE, se déroule exactement comme un projet Java ordinaire. Seule la cible change. La commande "Executer le programme" envoie le code par le câble USB ou par Bluetooth, ce qui évite de devoir constamment brancher/débrancher le robot (le câble est trop court pour le laisser branché pendant qu'il se déplace). Lors de l'exécution, un terminal présent dans Eclipse indique les fonctions employées. Signalons aussi que leJOS refuse de s'éxecuter sans l'utilsation du Java SE Development Kit 32 bits.

Enfin, le logiciel NXJ Browse permet d'envoyer des fichiers, comme par exemple des sons, des images ou même du code sur la brique NXT.

Structure de données et algorithmes employés

4.1 Structure de données

Pour représenter le labyrinthe, nous pensions initialement utiliser une matrice, qui nous semblait proche de l'image d'un plan de labyrinthe, mais nous nous sommes rapidement aperçus qu'une matrice rendait difficile la gestion des murs et des passages possibles. Nous avons donc finalement choisi d'utiliser une structure de graphe : chaque sommet du graphe correspond à une "case" du labyrinthe, et les arêtes entre les sommets représentent les passages disponibles d'une case à une autre. Cette structure s'est avérée être relativement simple à manipuler, et nous a permis d'utiliser facilement l'algorithme de parcours en profondeur.

Dans notre structure de données, chaque case contient les informations suivantes : un couple (x,y) d'entiers de coordonnées, un booléen pour le marquage des visites, et des pointeurs vers les cases voisines pour représenter les arêtes du graphe.

Les coordonnées permettent d'une part de faciliter la représentation du labyrinthe, mais permettent aussi de reconnaitre les cases déjà connues. En effet, si le robot découvre à un moment une case X située à l'Ouest de la case Y, mais ne la visite pas, puis se retrouve finalement à l'Ouest de la case X après un détour, les coordonnées permettent de reconnaitre la case X dans la liste des cases connues. Si cette reconnaissance n'est pas faite, le robot ne serait pas capable de reconstituer correctement le graphe correspondant au labyrinthe et serait condamné à tourner en rond dès qu'il tombe sur un cycle.

Le booléen pour le marquage des visites est nécessaire pour parcourir un graphe : il évite de passer plusieurs fois au même endroit. Dans notre programme, le marquage des cases est effectué quand le robot arrive sur une case pas encore visitée. Il observe son environnement avec les capteurs pour créer les arêtes vers les cases voisines et vérifier qu'il n'est pas sur la case de sortie, puis marque la case comme visitée.

Enfin, les 4 pointeurs vers les cases voisines permettent la représentation sous forme de graphe. Dans notre cas, les pointeurs correspondent chacun à un point cardinal pour permettre au robot de savoir dans quel sens se tourner pour se rendre sur une case voisine de la case en cours d'exploration. Comme notre robot ne possède pas de boussole, on décide arbitrairement que le Nord correspond à l'orientation initiale du robot lorsque le programme est démarré.

Les autres objets utilisés au sein de notre programme sont un objet de la classe "robot" et un objet "écran" qui gère l'affichage.

L'objet robot possède un pointeur vers la case où il se situe, et une orientation pour savoir où il se dirige. Il est aussi utilisé pour appeler les méthodes de déplacement. L'objet écran sert uniquement à appeler les fonctions liées à l'affichage. Il sert notamment à afficher à l'écran le plan qui permet aux utilisateurs de voir ce que le robot connait du labyrinthe.

4.2 Algorithmes utilisés

4.2.1 Explorer l'environnement

Cette première fonction à pour objectif d'utiliser les capteurs du robot pour récupérer des informations sur l'environnement. Les capteurs ultra-sonores sont utilisés avec les fonctions du type *robot.NordExists()* qui renvoient la valeur *true* si la case adjacente dans la direction indiquée est considérée comme accessible. Si les cases sont accessibles, on initialise les nouvelles cases, si elles n'existent pas déjà, puis on les relie à la case courante.

Après avoir étudié l'état des cases voisines, on utilise le capteur de couleur pour savoir si le robot est sur la case de sortie, et on retourne cette information sous forme d'un booléen.

Listing 4.1 – Fonctions pour explorer l'environnement

```
1  public boolean NordExists(){
2    if(  // Si la case Nord est accessible
3        ((devant.getDistance()>WallDistance) && (orientation==NORD)) ||
4        ((gauche.getDistance()>WallDistance) && (orientation==EST)) ||
5        ((droite.getDistance()>WallDistance) && (orientation==OUEST))
6        )    return true;
7    else return false;
8  }
9  //On procede de la meme maniere pour les trois autres points cardinaux
10 
11 public static boolean exploreEnvironment(Robot robot, ArrayList<Labycase>
       labyrinthe){
12 
13   //La structure Robot contient un pointeur sur la case en cours d'exploration, c'
       est le champ "CaseCourante"
14   //La liste labyrinthe contient la liste de toutes les cases connues par le robot
15 
16   int nx,ny;
17   ColorSensor cs = new ColorSensor(SensorPort.S3, SensorConstants.TYPE_LIGHT_ACTIVE
       );
18   Labycase temp;
19 
20   //On marque la case en cours comme etant visite
21   robot.CaseCourante.visite = true;
22 
23   if(  robot.NordExists() )
24     { //Si la case Nord est accessible, on verifie d'abord son existence
25       nx=robot.CaseCourante.x;
26       ny=robot.CaseCourante.y-1;
27       temp = getCase(nx,ny,labyrinthe);
28       if (temp==null) { //Si elle n'existe pas on la cree
29         temp = new Labycase(robot.CaseCourante,1,nx,ny);
30         labyrinthe.add(temp);
31       }
32       //On effectue ensuite la liaison des cases (lors de l'initialisation, les
           nouvelles cases sont automatiquement liees a leur predecesseur)
33       else temp.Sud = robot.CaseCourante;
34       robot.CaseCourante.Nord = temp;
35     }
36     //On repete l'operation avec les autres points cardinaux
37     if( robot.EstExists() )
38     {
39       nx=robot.CaseCourante.x+1;
40       ny=robot.CaseCourante.y;
```

```
41          temp = getCase(nx,ny,labyrinthe);
42          if (temp==null) {
43            temp = new Labycase(robot.CaseCourante,2,nx,ny);
44            labyrinthe.add(temp);
45          }
46          else temp.Ouest = robot.CaseCourante;
47          robot.CaseCourante.Est = temp;
48        }
49        if( robot.SudExists() )
50        {
51          nx=robot.CaseCourante.x;
52          ny=robot.CaseCourante.y+1;
53          temp = getCase(nx,ny,labyrinthe);
54          if (temp==null) {
55            temp = new Labycase(robot.CaseCourante,3,nx,ny);
56            labyrinthe.add(temp);
57          }
58          else temp.Nord = robot.CaseCourante;
59          robot.CaseCourante.Sud = temp;
60        }
61        if( robot.OuestExists() )
62        {
63          nx=robot.CaseCourante.x-1;
64          ny=robot.CaseCourante.y;
65          temp = getCase(nx,ny,labyrinthe);
66          if (temp==null) {
67            temp = new Labycase(robot.CaseCourante,4,nx,ny);
68            labyrinthe.add(temp);
69          }
70          else temp.Est = robot.CaseCourante;
71          robot.CaseCourante.Ouest = temp;
72        }
73
74        //Apres avoir etudie les cases adjacentes, on regarde si on est sur la case de
            sortie, et on termine l'exploration
75        return (cs.getColorID() == ColorSensor.Color.RED);
76    }
```

4.2.2 Parcours en profondeur

Pour parcourir le labyrinthe, nous utilisons le classique parcours en profondeur qui est adapté au cheminement d'un robot dans un labyrinthe : un parcours en largeur obligerait le robot à faire des aller-retours incessants pour explorer, alors que le parcours en profondeur permet d'explorer complètement une branche du labyrinthe avant de faire demi-tour pour retourner au dernier point visité qui possède des voisins non-explorés.

L'implémentation que nous avons faite explore les cases voisines dans un ordre arbitraire prédéfini (on privilégie les voisins du Nord, puis ceux du Sud, puis ceux de l'Ouest, et les voisins de l'Est sont explorés en dernier), mais pourrait être modifiée pour privilégier les déplacements en ligne droite. Cela aurait cependant pour effet de rendre le code moins lisible et de réduire sa maintenabilité.

Listing 4.2 – Algorithme de parcours en profondeur

```
1 public boolean DFS(Robot robot, ArrayList<Labycase> labyrinthe, Affichage Ecran){
2   //Le parametre Ecran permet de rafraichir le plan affiche sur l'ecran LCD du
      robot
```

```
3
4     //On commence par lancer l'exploration de la nouvelle case pour connaitre les
          voisins disponibles, et savoir si le robot a atteint la sortie
5     boolean fini = exploreEnvironment(robot, labyrinthe);
6     //On affiche le plan acutalise avec les nouveaux voisins
7     Ecran.AffichePlan(robot, labyrinthe);
8
9     if (fini) return true; //Si on est sur la sortie, on arrete le parcours
10
11    if(this.Nord!=null && !this.Nord.visite) //On a un voisin au Nord, qui n'est pas
          encore marque
12    {
13      robot.deplacer(this.Nord, Ecran);
14      fini = this.Nord.DFS(robot, labyrinthe, Ecran);
15      if (!fini) robot.deplacer(this, Ecran); //pour le retour en sortie du DFS quand
            on a fini d'explorer une branche
16    }
17    if (fini) return true; //Si la branche exploree au Nord contient la sortie, on
          arrete le parcours sans aller explorer les sommets restants
18
19    //On repete la meme operation avec les 3 autres voisins potentiels
20
21    if(this.Sud!=null && !this.Sud.visite) //On a un voisin au Sud, qui n'est pas
          encore marque
22    {
23      robot.deplacer(this.Sud, Ecran);
24      fini = this.Sud.DFS(robot, labyrinthe, Ecran);
25      if (!fini)robot.deplacer(this, Ecran); //pour le retour en sortie du DFS quand
            on a fini d'explorer une branche
26    }
27    if (fini) return true;
28    if(this.Est!=null && !this.Est.visite) //On a un voisin au Est, qui n'est pas
          encore marque
29    {
30      robot.deplacer(this.Est, Ecran);
31      fini = this.Est.DFS(robot, labyrinthe, Ecran);
32      if (!fini)robot.deplacer(this, Ecran); //pour le retour en sortie du DFS quand
            on a fini d'explorer une branche
33    }
34    if (fini) return true;
35    if(this.Ouest!=null && !this.Ouest.visite) //On a un voisin au Ouest, qui n'Ouest
          pas encore marque
36    {
37      robot.deplacer(this.Ouest, Ecran);
38      fini = this.Ouest.DFS(robot, labyrinthe, Ecran);
39      if (!fini)robot.deplacer(this, Ecran); //pour le retour en sortie du DFS quand
            on a fini d'explorer une branche
40    }
41    if (fini) return true;
42
43    //Si la fonction ne s'est pas arrete avant d'arriver ici, aucune des branches
          partant de la case courante ne contient la sortie, on fait donc remonter ce
          resultat
44    return false;
45  }
```

4.2.3 Déplacements du robot

Les déplacements du robot sont gérés par des fonctions afin d'obtenir des mouvements formatés : avancer d'une case, tourner à 90° ou faire un demi-tour. Pour réaliser ces mouvements, on demande aux moteurs d'avancer de manière synchrone, ou d'avancer chacun dans une direction contraire pour effectuer les rotations.

Lors du parcours du labyrinthe, le robot est déplacé avec une fonction intermédiaire qui permet de gérer l'orientation : en fonction de l'orientation actuelle du robot, et en sachant où le robot souhaite se rendre, la fonction effectue les rotations nécessaires puis fait avancer le robot.

Listing 4.3 – Fonction pour gérer les déplacements

```
public void deplacer(Labycase Objectif){
  Delay.msDelay(1000);

  if( this.CaseCourante.Nord == Objectif) //Si la case que l'on souhaite atteindre
      est au Nord de la case actuelle
    {
      switch(orientation){
        case NORD : break;
        case EST : this.turnLeft();break;
        case SUD : this.turnAround();break;
        case OUEST : this.turnRight();break;
      }
    }
  if( this.CaseCourante.Sud == Objectif) //Si la case est au Sud
    {
      switch(orientation){
        case NORD : this.turnAround();break;
        case EST : this.turnRight();break;
        case SUD : break;
        case OUEST : this.turnLeft();break;
      }
    }
  if( this.CaseCourante.Est == Objectif) //Si la case est a l'Est
    {
      switch(orientation){
        case NORD : this.turnRight();break;
        case EST : break;
        case SUD : this.turnLeft();break;
        case OUEST : this.turnAround();break;
      }
    }
  if( this.CaseCourante.Ouest == Objectif) // Si la case est a l'Ouest
    {
      switch(orientation){
        case NORD : this.turnLeft();break;
        case EST : this.turnAround();break;
        case SUD : this.turnRight();break;
        case OUEST : break;
      }
    }

  //Apres avoir effectue les rotations, on avance et on change la CaseCourante du
      robot
  this.CaseCourante = Objectif;
  moveForward();
}
```

4.2.4 Recherche de case

Cette fonction permet de chercher dans la liste des cases connues si une case correspond aux coordonnées entrées en paramètre. Comme indiqué précédemment, cette fonction permet de reconnaitre les cases déjà connues. Elle sert aussi pour l'affichage du plan à l'écran.

Listing 4.4 – Fonction pour rechercher une case

```
1  public static Labycase getCase(int x, int y, ArrayList<Labycase> labyrinthe){
2    //retourne la case qui correspond au coordonnees (x,y) ou null si la case n'
       existe pas
3
4    int t=0;
5    Labycase temp;
6    for(t=0;t<labyrinthe.size();t++){
7      temp = labyrinthe.get(t);
8      if ((temp.x == x) && (temp.y == y))
9        return temp;
10   }
11   return null;
12 }
```

4.2.5 Affichage du plan à l'écran

Face à l'absence de débuggeur pour permettre de lire le contenu de la mémoire du robot, nous avons rapidement été obligé d'intégrer une fonction pour permettre l'affichage des connaissances du robot.

Les cases dont on ne connait pas l'état sont représentées par une case noire, car il s'agit d'endroits inaccessibles au vu des connaissances actuelles du robot. Les cases connues mais pas encore visitées sont représentés avec un point d'interrogation, et les cases visitées sont représentées par une case blanche entourée de murs correspondants aux voisins inaccessibles.

Listing 4.5 – Fonction pour gérer l'affichage du plan

```
1  public void AffichePlan(Robot robot, ArrayList<Labycase> Labyrinthe){
2    int i,j,x,y;
3    Labycase Labytemp;
4    ArrayList<Labycase> Map = new ArrayList<Labycase>(1);
5
6    //On vas chercher les 7x5 cases adjacentes a la position actuelle du robot
7    for(j=0;j<5;j++){
8      for(i=0;i<7;i++){
9        Map.add(Labycase.getCase((i-3)+robot.CaseCourante.x, (j-2)+robot.CaseCourante
           .y, Labyrinthe));
10     }
11   }
12
13   LCD.clear();
14   g.setColor(0);
15
16   for(j=0;j<5;j++){
17     for(i=0;i<7;i++){
18       Labytemp = Map.get(((i+1)+7*j)-1);
19       x=8+i*12;
20       y=4+j*12;
21       if (Labytemp==null){ //Si la case n'existe pas, ou n'est pas connue
```

```
22          g.fillRect(x, y, 12, 12);
23          //On dessine un carre noir
24          }
25        else if (!Labytemp.visite){ //Si la case case existe, mais pas encore visite
            (on ne connait donc pas ses voisins)
26          g.drawLine(x+4, y+2, x+7, y+2);
27          g.fillRect(x+3, y+3, 2, 2);
28          g.fillRect(x+7, y+3, 2, 2);
29          g.drawLine(x+6, y+5, x+8, y+5);
30          g.drawLine(x+5, y+6, x+6, y+6);
31          g.fillRect(x+5, y+8, 2, 2);
32          //On dessine un point d'interrogation
33          }
34        else{ //Sinon, la case existe, et a deja ete visitee, on connait donc ses
            voisins
35          if(Labytemp.Nord==null)    g.drawLine(x, y, x+11, y);
36          if(Labytemp.Sud==null)    g.drawLine(x, y+11, x+11, y+11);
37          if(Labytemp.Ouest==null)  g.drawLine(x, y, x, y+11);
38          if(Labytemp.Est==null)    g.drawLine(x+11, y, x+11, y+11);
39          //On dessine, un carre blanc, et on trace des bords noirs sur les cotes qui
              possedent un mur
40          }
41        }
42      }
43    switch(robot.GetOrientation()){ //Enfin, sur la case centrale on dessine une
        fleche pour indiquer l'orientation actuelle du robot
44      case 1 :g.drawLine(46, 33, 49, 30);g.drawLine(50, 30, 53, 33);g.fillRect(49,
          31, 2, 7);break;
45      case 2 :g.drawLine(50, 30, 53, 33);g.drawLine(53, 34, 50, 37);g.fillRect(46,
          33, 7, 2);break;
46      case 3 :g.drawLine(46, 34, 49, 37);g.drawLine(50, 37, 53, 34);g.fillRect(49,
          30, 2, 7);break;
47      case 4 :g.drawLine(50, 30, 46, 33);g.drawLine(46, 34, 49, 37);g.fillRect(47,
          33, 7, 2);break;
48      }
49
50  }
```

Résultat

Le but du projet était de permettre à un robot posé dans un labyrinthe, sans aucune information sur ce dernier, de trouver la sortie.

Dans la pratique, notre robot est tout à fait capable de parcourir le labyrinthe à la recherche de la sortie, si l'on omet toutefois les quelques problèmes de trajectoires liés à la fiabilité technique du robot détaillés dans la suite (cf section 6.2). Il faut donc surveiller le robot pour s'assurer qu'il ne s'écarte pas trop du centre des cases et le repositionner de temps en temps.

Pendant que le robot parcourt le labyrinthe, un plan miniature affiché sur l'écran permet de voir les cases explorées par le robot avec leurs murs, celles connues mais pas encore explorées, et les zones inaccessibles d'après les connaissances du robot (cf section 4.2.5).

Si le labyrinthe ne comporte pas de sortie, le robot explore la totalité du labyrinthe avant de revenir à sa position de départ, puis affiche un message à l'écran indiquant "Labyrinthe sans issue" accompagné d'une tonalité sonore.

Si le labyrinthe comporte une sortie (matérialisée par une surface rouge disposée au sol), le robot s'arrête sur la case rouge, et affiche un message "Victoire !" accompagné d'une autre tonalité sonore.

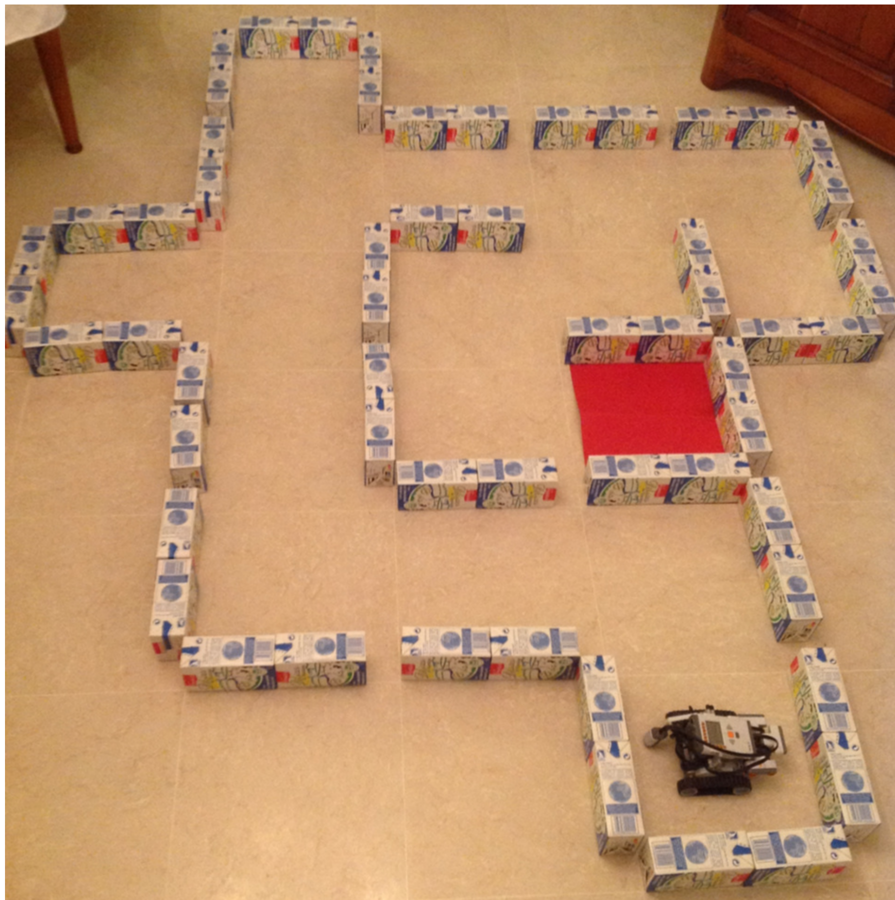


FIGURE 5.1 – Mise à l'épreuve de notre robot dans un labyrinthe improvisé

Difficultés rencontrées

6.1 Aléas divers

Au départ, nous voulions programmer le robot avec le langage NXC, un langage utilisant la structure du C pour permettre de manipuler le NXT. Cependant cette adaptation du langage C ne permettait pas certaines fonctionnalités essentielles telles que la manipulation d'une structure de données dynamique. Face à ce manque nous avons cherché d'autres alternatives, et nous avons finalement choisi leJOS, une adaptation du langage Java pour le NXT.

Notre projet était censé reprendre une partie du travail de Benjamin Allée qui avait l'année dernière programmé la résolution d'un labyrinthe. Mais en observant son code source, nous avons remarqué que le robot connaissait dès le départ sa position dans labyrinthe, ainsi que la taille du labyrinthe. Dans notre cas le robot ne connait pas sa position de départ, ni la taille du labyrinthe, il nous a donc fallu utiliser une autre structure de données pour prendre en compte ces différences.

6.2 Fiabilité du robot

Les robots étant des machines "réelles", elles sont moins fiables qu'un robot virtuel dans une simulation par ordinateur. Cependant notre robot s'est avéré particulièrement imprécis.

Le premier problème constaté vient des capteurs ultra-sonores : lorsque les obstacles sont trop proches (moins de 10 cm), les mesures sont fausses. Sur les obstacles plus éloignés les mesures manquent de précision. Pour palier à ce problème, les mesures sont considérées avec un seuil : au delà d'une certaine distance, on considère qu'il n'y a pas de mur et que la case suivante est accessible, et au contraire qu'il y a un mur si la distance mesurée est en dessous du seuil.

Le deuxième défaut des capteurs ultra-sonores est lié au fonctionnement des ondes : la mesure indique la distance de l'obstacle le plus proche dans un angle d'une trentaine de degrés. Selon la position du robot par rapport aux murs, les capteurs latéraux peuvent par exemple détecter les murs de la case de devant et en conclure qu'il y a des murs à côté du robot alors que les passages sont libres. Il faut donc autant que possible s'assurer que le robot soit le plus proche du centre de la case pour limiter les risques.

Les autres problèmes sont liés aux moteurs et à l'adhésion des roues au sol : dès qu'on change de type de sol, le calibrage des mouvements est à refaire. Au début nous pensions que les mouvements du robot seraient relativement fiables une fois les calibrages effectués, mais il s'est avéré que ce n'est pas le cas.

Lorsqu'on demande au robot d'avancer tout droit, on peut observer qu'il se décale vers la droite avec un angle variant de 4°à 10°. Le décalage pourrait être partiellement compensé en augmentant la vitesse du moteur de droite, mais nous avons découvert le caractère aléatoire du problème trop tard pour chercher des outils mathématiques permettant de compenser cette distorsion.

De même, pour les rotations d'un quart de tour, nous avons constaté un manque important de fiabilité. Nous avons demandé au robot d'effectuer plusieurs quarts de tour sans modifier la constante utilisée pour contrôler les moteurs et nous avons constaté des écarts dans l'angle de rotation allant jusqu'à 10°.

Nous avons aussi constaté que les roues sont mal situées par rapport au centre de gravité du robot, ce qui a pour effet de décaler légèrement le robot à chaque rotation. Cependant l'erreur occasionnée est négligeable par rapport aux problèmes évoqués ci-dessus.

La combinaison de ces problèmes sur les mouvements rend la trajectoire du robot très imprévisible : même si les distances parcourues en avançant nous ont semblées plutôt fiables, la légère déviation rapproche

lentement le robot des murs, tandis que les rotations ratées peuvent provoquer d'important décalages par rapport au centre de la case, ce qui conduit inévitablement le robot à percuter les murs.

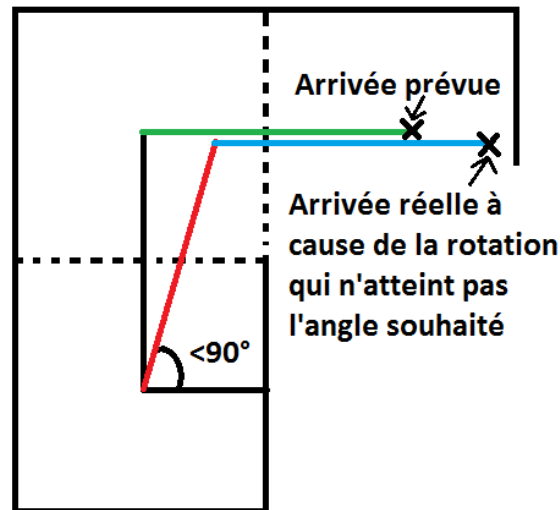


Figure 6.1 – Chaque mouvement inexact décale le robot

Pour essayer de résoudre ces problèmes, nous avons envisagé d'autres méthodes de déplacement : au lieu d'avancer de manière aveugle, le robot pourrait utiliser ses capteurs latéraux pour se rapprocher du centre des cases en modifiant la vitesse de ses moteurs, et s'il se retrouve face à un mur, il interromprait son déplacement pour reculer de manière à rejoindre le centre de la case.

Cependant, la manipulation des moteurs avec le langage leJOS permet difficilement d'utiliser en parallèle les capteurs pour effectuer de telles manœuvres sans interrompre complètement le mouvement. Si le robot stoppe son mouvement, les manœuvres restent très compliquées et risquent d'être longues à cause des temps de démarrage/stoppage des moteurs.
De plus, l'imprécision des capteurs risque de rendre les mouvements chaotiques.

Évolutions envisagées

7.1 Utiliser un calcul du plus court chemin

Dans notre implémentation du parcours en profondeur, lorsque la récursivité se termine, nous demandons au robot de retourner sur ses pas jusqu'à la dernière intersection qui possède un voisin non exploré afin de continuer le parcours. Cependant si le graphe comporte un cycle, lors du retour, le robot parcourra le cycle dans le sens inverse de son premier passage, alors qu'un chemin plus court peut exister.

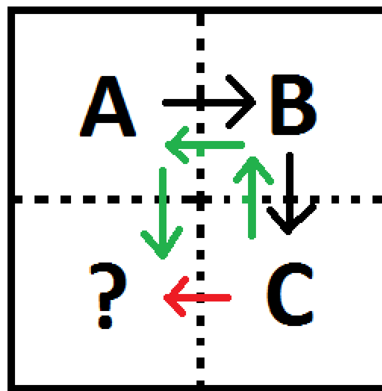


FIGURE 7.1 – Avec le plus court chemin (flèche rouge), le robot n'aurait pas besoin de repasser par A et B après avoir exploré C (flèches vertes)

En utilisant un calcul du plus court chemin, le robot pourrait se déplacer un peu plus rapidement, mais cette amélioration ne servirait que dans le cas cité plus haut, donc l'amélioration aurait un impact plutôt limité sur temps nécessaire pour le parcours du labyrinthe.

7.2 Modifications sur le robot

Le manque de fiabilité du robot Mindstorms provoquant de nombreux problèmes, l'une des premières évolutions envisageable pour améliorer le projet est d'utiliser un robot qui permettrait un meilleur contrôle des trajectoires. Le simple ajout d'un capteur du type gyroscope permettrait au robot de connaître l'angle de sa trajectoire, et il serait ainsi possible de le cadrer sur des angles précis : 0°, 90°, 180°et 270°par rapport à sa position initiale. Avec une telle modification le robot serait en mesure d'éviter de se décaler du centre de la case, et comme les distances de déplacement nous ont semblé plutôt fiables, il devrait être en mesure d'arriver sans encombre à la sortie du labyrinthe.

D'autres modifications pourraient être effectuées sur la structure du robot : nous avons assemblé le modèle de base présenté dans la notice de montage fourni par Lego, afin de ne pas perdre trop de temps sur l'aspect montage du robot, mais une construction utilisant des roues folles aurait peut-être amélioré la précision des rotations.

Enfin la perception de l'environnement par le robot pourrait être améliorée en changeant le type de capteurs : les capteurs ultra-sons avec leur zone de détection sont adaptés pour détecter des obstacles "ponctuels" (exemple un pied de chaise), mais dans notre cas avec des murs bien définis, cette zone de détection est plutôt source de parasites. Un capteur avec une zone moins large serait peut-être plus adapté.

Une autre évolution possible serait de monter un capteur sur un axe en rotation, afin que le robot dispose d'une sorte de radar qui lui fournirait une représentation 2D de son environnement au lieu d'une simple distance vers l'obstacle le plus proche. Une telle représentation permettrait de mieux détecter les passages, de repositionner plus rapidement le robot sur le centre des cases, ou d'envisager des types de labyrinthes plus complexes (cf section 7.4).

7.3 Programme de calibrage automatique

Une autre faiblesse de notre robot actuel vient du calibrage : les constantes utilisées pour le déplacement du robot (les temps d'allumage des moteurs pour faire avancer le robot d'une case, ou pour faire une rotation) et la distance de détection des murs sont inscrites "en dur" dans le code, et nécessitent une recompilation pour être modifiées. La modification de ces valeurs est laborieuse, parce qu'il faut tester de nombreuses fois les valeurs dès qu'on change de surface pour s'assurer qu'elles soient correctes.

En supposant que la précision des rotations soit gérée à l'aide d'un gyroscope, on peut imaginer un programme d'apprentissage pour que le robot découvre son environnement : à l'aide d'un labyrinthe de 1x2 cases, le robot pourrait trouver lui même les valeurs adaptées en commençant par se positionner au centre de la première case à l'aide de ses capteurs, puis en effectuant des allers-retours entre le centre des deux cases pour trouver le temps d'allumage des moteurs adéquat pour effectuer un déplacement parfait.

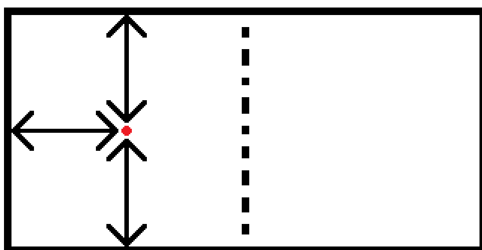


FIGURE 7.2 – Première étape : le robot se place au centre de la case à l'aide de petits mouvements, de manière à ce que les capteurs indiquent tous la même distance ; l'écart avec les murs pourra servir à déterminer la distance utilisée pour savoir s'il y a un mur ou non.

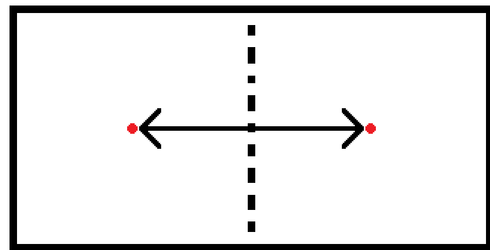


FIGURE 7.3 – Deuxième étape : le robot procède à des allers-retours entre les deux cases pour déterminer empiriquement le temps de fonctionnement des moteurs permettant de parcourir exactement la longueur d'une case. La bonne valeur est trouvée quand le robot atteint directement le centre de l'autre case sans devoir se réajuster.

7.4 Changer la structure de données

La structure de données que nous avons utilisée fonctionne uniquement avec les labyrinthes à base de cases carrées. Pour gérer des labyrinthes plus complexes avec par exemple des couloirs courbés ou des intersections qui contiennent plus que 4 directions possibles, il faudrait probablement avoir d'une part une sorte de plan, semblable à un plan qui serait tracé à main levé, et d'autre part une structure de graphe, les sommets représentant uniquement les intersections du labyrinthe. Le plan permettrait de calculer une trajectoire pour voyager entre deux intersections.

7.5 Utiliser les communications Bluetooth

La connexion Bluetooth permet à une brique NXT de communiquer avec un ordinateur ou avec d'autres briques NXT. La communication avec un ordinateur pourrait permettre un meilleur affichage des connaissances du labyrinthe car l'écran LCD de la brique est assez limité. Elle pourrait également permettre de

palier le manque de mémoire des briques. En effet la version actuelle du robot embarque une petite quantité de mémoire, et même si nous n'avons pas rencontré de problèmes avec, la mémoire risque de ne pas suffire si le robot est utilisé dans un labyrinthe particulièrement grand.

La communication avec d'autres briques pourrait être utilisée pour gérer un groupe de robots : à chaque intersection les robots se sépareraient pour explorer plus rapidement les branches du graphe, et s'ils tombent sur une impasse, ils rebrousseraient chemin puis rejoindraient les autres robots.

Conclusion

Ce projet de résolution d'un labyrinthe nous a tout d'abord permis de nous initier à la robotique, ce que nous n'avions jamais fait auparavant. De plus, la plateforme Lego NXT est un réel plaisir à utiliser et à programmer : elle propose différents types de capteurs ainsi que de nombreux langages de programmation. Les possibilités sont immenses.

Pour ce projet, nous avons donc eu le choix des technologies employées. Nous avons choisi des capteurs ultra sonores pour gérer les obstacles mais aussi un capteur de couleur pour détecter la sortie. Au départ, nos professeurs encadrants ont souhaité l'utilisation du langage C. Malheureusement, ce langage n'était pas adapté. Après discussion, la technologie Java fut employée. Nous avons réellement apprécié cette réelle autonomie.

Nous avons implémenté l'algorithme de parcours en profondeur pour pouvoir résoudre tout type de labyrinthe. Nous avons aussi géré les déplacements du robot en fonction de sa direction ainsi que l'utilisation des capteurs.

En pratique, notre robot réussit à trouver la sortie (si elle est disponible) mais on se heurte à des problèmes inhérents au robot utilisé : les déplacements et les rotations ne sont pas bien réalisés. Au fur et à mesure du parcours, le robot est décalé.

Nous pensons donc que la priorité dans le futur est de régler ce problème grâce à l'utilisation d'un capteur de type boussole ou gyroscope. Le robot pourra alors se déplacer de manière optimale.

Résolution d'un labyrinthe à l'aide d'un robot Lego Mindstorms

---

Département Informatique
4e année
2012 - 2013

Projet Robotique

Résumé : A l'aide du langage Java et de la technologie leJOS, nous avons implémenté l'algorithme de parcours en profondeur sur un robot de type Lego NXT pour pouvoir résoudre un labyrinthe. Des capteurs ultra-sonores ont été employés pour la détection des murs ainsi qu'un capteur de couleur pour détecter la sortie.

Mots clefs : Résolution de labyrinthe, Lego , NXT, leJOS, Java, parcours en profondeur

Abstract: This project aims to solve a maze with a Lego NXT. Depth first search algorithm and sensors (ultrasound sensors to detect walls and color sensor to detect goal) are employed to make this possible. Implementation is realised in Java with the leJOS technology.

Keywords: Maze solver, Lego , NXT, leJOS, Java, DFS, Depth First Search

---

Encadrants
Pierre Gaucher
pierre.gaucher@univ-tours.fr
Jean-louis Bouquard
bouquard@univ-tours.fr

Étudiants
Fabien Buda
fabien.buda@etu.univ-tours.fr
Alexandre Lefillastre
alexandre.lefillastre@etu.univ-tours.fr

Université François-Rabelais, Tours

DI4 2012 - 2013