

ENSIIE Strasbourg

Parc d'Innovation, Bâtiment de l'ISU

1 rue Jean-Dominique Cassini

67400 Illkirch

PROJET NXC :

RESOLVEUR

DE LABYRINTHE

Guillaume PARAN

Ismail LAGOUILLY

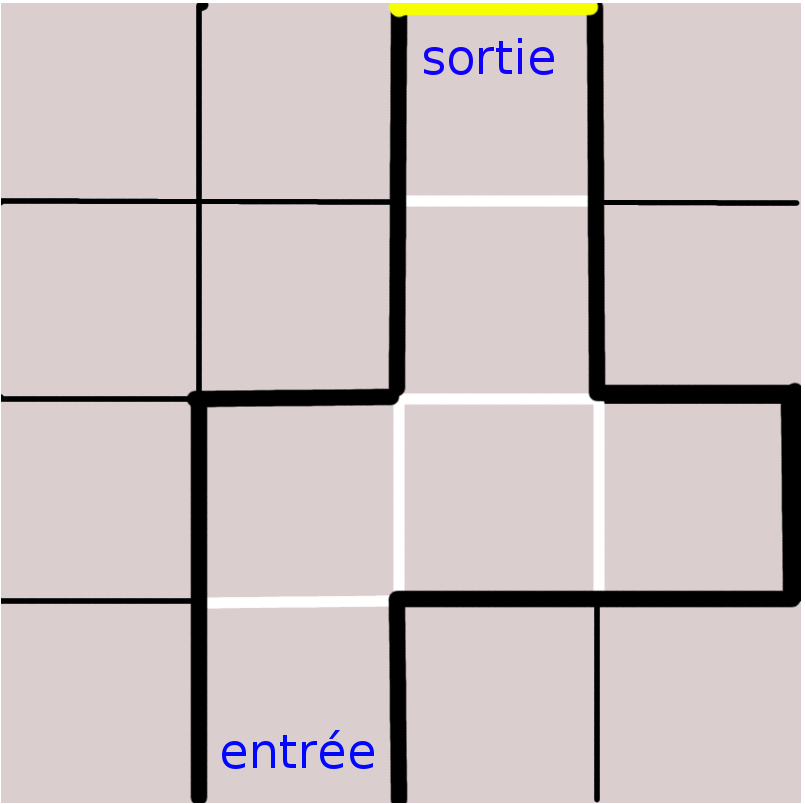
Quentin THENOZ

1. Introduction

# 1.1. Présentation du sujet

Le projet consiste à faire parcourir par un robot un labyrinthe quelconque.

Nous avons choisi de représenter le labyrinthe par des lignes de couleurs différentes : une ligne noire signifie un mur et une ligne blanche signifie que l'on peut passer. Les lignes blanches délimitent chaque case du labyrinthe.



# 1.2. Présentation des outils

## Lego Mindstorm NXT

Lego Mindstorms NXT est un jeu de construction et de robotique présenté par Lego en 2006. Il succède à la gamme Lego Mindstorms RCX. En 2010, Mindsorms NXT en est à sa version 2.0.

Caractéristiques techniques : ARM7 48 MHz, 256 KB Flash RAM, 64 KB RAM.

## NXC

Le Not eXactly C (pas exactement du C) ou NXC est un langage de programmation spécifique au robot NXT basé sur le NBC, un langage assembleur. Dans sa version actuelle, le NXC ne supporte ni pointeurs ni récursivité (dans les structures ou les fonctions). L'IDE associé à ce langage est BricxCC.

## Subversion

Subversion (communément appelé SVN) est un système de gestion de versions. Il permet de conserver un historique de toutes les versions de code source qu’on a mis sur le dépôt. Ainsi, il est possible de revenir à une ancienne version, de créer une copie de travail parallèle en cas de modification autonome importante, ou encore de fusionner différentes versions.

2. Le projet

# 2.1. Répartition des tâches

  Au commencement du projet nous nous sommes mis d'accord sur les étapes que nous devions :

- Primitives et tâches : cette partie concerne les méthodes de base pour pouvoir se déplacer dans le labyrinthe et détecter les murs, les impasses et de la sortie.

- Structure de données : cette partie concerne la définition des données nécessaires au parcours en profondeur du labyrinthe.

- Algorithme de résolution : cette étape a pour objectif d'analyser les différentes possibilités de chemin pour arriver à la sortie par le plus proche chemin.

# 2.2. Les primitives et tâches

Afin que le robot puisse parcourir le labyrinthe, des primitives ont été implémentées.

Il s'agit de fonctions permettant de se déplacer comme tourner et avancer, mais également des fonctions de test de couleur de ligne.

Pour la rotation il y a un paramètre de durée qui a été réglé en coordination avec la puissance des moteurs afin d'obtenir une rotation de 90° avec le moins d'erreur possible, seulement on peut remarquer que toutes les rotations ne sont pas identiques. Ainsi, pour un même réglage, le robot peut parfois tourner de plus de 90° et parfois moins.

*// fait tourner le robot vers la gauche ou la droite*

**void** tourner(Direction s, **int** duree)

{

**if**(s == dir\_gauche)

{

OnRev(OUT\_A, PUISSANCE\_ROTATION);

OnFwd(OUT\_B, PUISSANCE\_ROTATION);

Wait(duree);

Off(OUT\_AB);

}

**else if**(s == dir\_droite)

{

OnRev(OUT\_B, PUISSANCE\_ROTATION);

OnFwd(OUT\_A, PUISSANCE\_ROTATION);

Wait(duree);

Off(OUT\_AB);

}

}

# 2.3. La structure de données

La structure de données a certainement été une des parties les plus épineuses du projet. L'idée générale est que l'on souhaite mapper le labyrinthe afin de savoir où sont les murs. Afin de maximiser les chances de trouver la sortie rapidement, nous avons décidé de faire un parcours du labyrinthe en profondeur. Il fallait donc, en plus du mapping, pouvoir se souvenir des cases "pivots" (ayant plusieurs sorties possibles) sur lesquelles le robot est passé.

Nous étions donc au départ d'accords pour représenter le labyrinthe sous la forme d'un arbre afin d'en faciliter le parcours. Cependant, il s'est avéré qu'il n'est pas possible de créer des structures récursives en NXC. Ce problème a été résolu par le choix d'utiliser une structure contenant une matrice et une pile de pivots. C'est alors qu'un second problème est survenu : la mémoire du bloc NXT est de 32Mo ! De ce fait, il nous était impossible de conserver nos structures.

Nous avons donc finalement opté pour la solution la moins couteuse en mémoire : des tableaux statiques déclarés de manière globale. Plus de structure Manager, plus de structure Pile, toutes les données sont globales.

Plus tard, nous avons ajouté à nos données une seconde pile contenant, elle, chaque direction prise par le robot à chaque étape de son parcours (par exemple : gauche, gauche, haut, droite, bas) et un indicateur permettant de savoir si cette direction a été prise à partir d'un pivot. Grace à cette pile nous avons résolu le problème du retour au dernier pivot traversé.

**enum** Direction {dir\_gauche=0, dir\_droite=1, dir\_haut=2, dir\_bas=3};

**enum** Coordonnee {coord\_x=0, coord\_y=1};

**struct** Case {

**int** x;

**int** y;

};

**struct** Pivot {

**bool** possibilites[4];

};

**struct** Element\_pile\_route {

Direction dir;

**bool** pivot;

};

**bool** manager\_routes[TAILLE\_LABYRINTHE\_X][TAILLE\_LABYRINTHE\_Y][4];

Pivot pile\_pivots[TAILLE\_PILE];

**int** pile\_pivots\_tete;

Element\_pile\_route pile\_route[TAILLE\_PILE];

**int** pile\_route\_tete;

Case robot\_position;

**int** direction\_x=0;

**int** direction\_y=1;

# 2.4. L'algorithme de résolution

Pendant que le robot parcours tout le labyrinthe, il construit une matrice et indique les différentes possibilités de chemin qu'il pourra emprunter (il fait un parcours en profondeur).

3. Conclusion

# 3.1. Bilan du projet

Au final, nous avons réussi à implémenter une structure compatible avec le robot. Les primitives fonctionnent et certaines tâches sont terminées. L’algorithme de résolution, quant à lui, est défini mais pas encore implémenté.

Malheureusement, nous ne sommes pas parvenus à faire résoudre un labyrinthe par notre robot, car de trop nombreuses complications sont  apparues dans le projet.La brique NXT n’est pas faite pour gérer une telle structure de données. La seule solution pour libérer un maximum de mémoire a été de mettre à jour le Firmware vers une version non officielle pour supprimer toutes les données implantées par LEGO (tests moteur, sons, etc.).

# 3.2. Ouverture

Pour que la résolution soit efficace il faut que le robot tourne exactement de 90° et avance parfaitement droit sans franchir une ligne qu'il ne devrait pas. Or étant donné que le robot subit de nombreuses manipulations, les erreurs s'accumulent exponentiellement. Lorsque le robot vérifie s’il y a un mur autour de sa case, cela implique 1 rotation vers la gauche, 2 vers la droite puis 1 nouvelle rotation vers la gauche. Pour une seule case, si le robot fait une erreur de 2 degrés par rotation, l'erreur finale sera donc de 8 à 10 degrés, soit, au bout de 4 cases, 32 à 40 degrés qui empêchent alors la résolution correcte du labyrinthe.

Une solution aurait été de construire un système plus complexe avec un troisième moteur qui aurait fait tourner le capteur de 90°, un peu comme une tourelle.

De cette façon les erreurs dues aux rotations auraient été divisées par 4, et la seule erreur qui aurait subsisté aurait été l'avancée non tout à fait rectiligne du robot.

Une autre solution (et peut être la meilleure)  permettant d'assurer une parfaite rotation de 90° et une avancée rectiligne aurait été de faire le labyrinthe avec de vrais murs (à la place du scotch) et d'utiliser le capteur à ultrason orienté à 90° (à gauche ou à droite) par rapport au robot pour calculer la distance au mur latéral. Un capteur de toucher aurait également pu être ajouté afin d’éviter les collisions.