Master 1 Informatique et Ingénierie des Systèmes Complexes (IISC)

Université de Cergy-Pontoise

Projet de synthèse

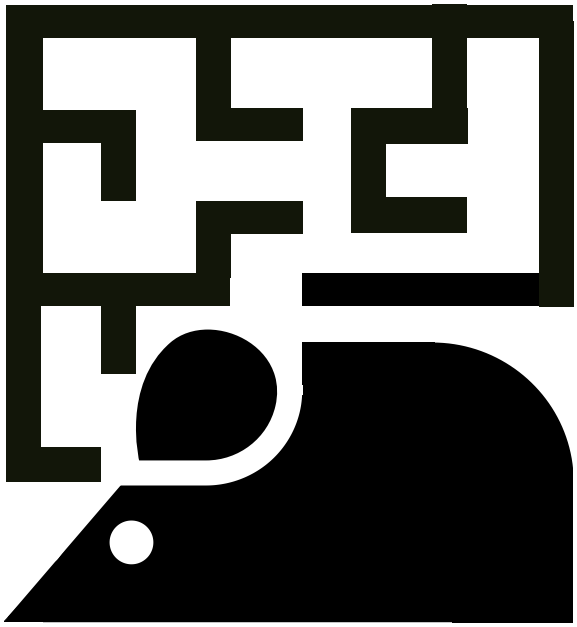
Résolution de labyrinthes par véhicule intelligent

Micromouse

rapporteur :

Auteurs :
Djahid ABDELMOUMENE
Amine AGRANE
Ishak AYAD
Donald LAY

Tuteur technique :
Pr. Alexandre PITTI
Encadrant de gestion de projet :
Pr. Tianxiao LIU



Rendu le
26 mai 2020

Remerciements

Résumé et abstract

Résumé

Abstract

Table des matières

Table des figures

Chapitre 1

Introduction

Chapeau du chapitre

1.1 Contexte du projet

1.2 Objectifs du projet

Chapeau

1.3 Mise en scénario

Étant donné que notre projet est composé de plusieurs briques applicatives, nous avons imaginé une mise en situation de notre produit permettant une démonstration globale des différentes fonctionnalités de ce dernier.

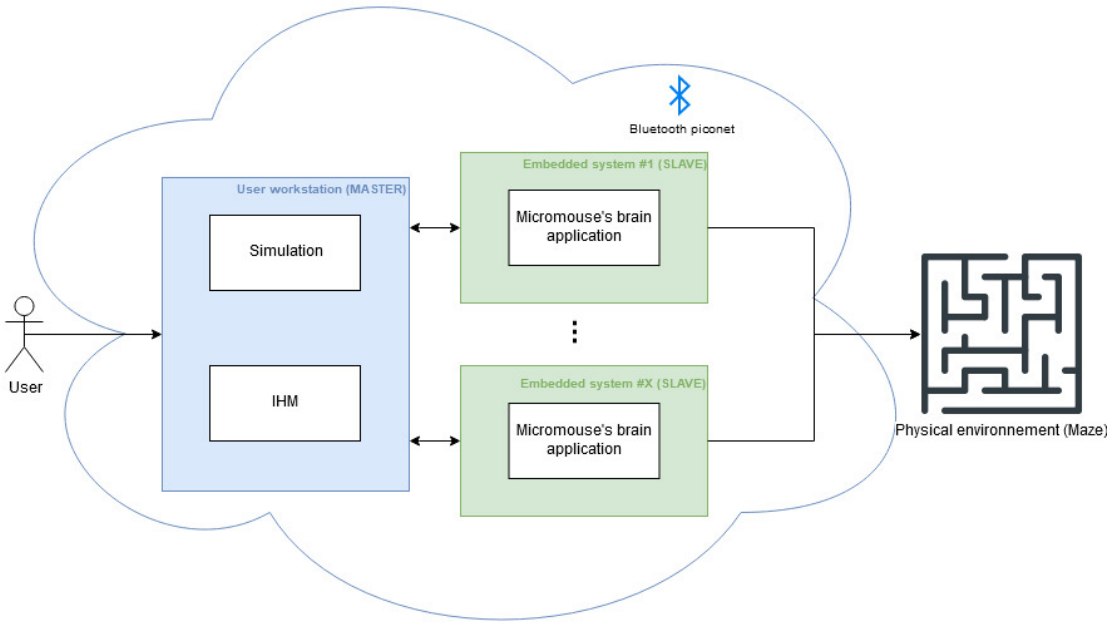


FIGURE 1.1 – Schéma d'infrastructure globale du projet.

La mise en situation que nous avons imaginée se déroule en trois étapes. Pour se faire, l'utilisateur aura accès à une interface depuis laquelle il pourra piloter les différentes phases de la démonstration (User workstation indiqué sur la Fig.[**??**]) :

1. À l'aide du module de simulation, l'utilisateur pourra simuler un environnement physique en paramétrant ou générant automatiquement un labyrinthe. L'intelligence, dont le code sera intégré au module de simulation, s'opérera sur cet environnement simulé. L'utilisateur pourra suivre les mouvements d'une micromouse virtuelle sur l'interface graphique ; mouvements qui seront synchronisés avec ceux d'une micromouse physique. Ainsi, l'utilisateur aura une double vision de l'intelligence du produit : une vision concrète avec le déplacement de la micromouse physique et une autre, plus théorique, avec l'interface graphique sur laquelle seront affichées les différentes métriques nécessaires au guidage de la micromouse ;

2. Une démonstration sur un environnement physique aura ensuite lieu sur un environnement que nous aurons préalablement construit. Cette démonstration se déroulera en deux phases : tout d'abord, une phase de mappage aura lieu dans laquelle la micromouse se déplacera dans le labyrintheen partant d'un point de départ jusqu'à une arrivée préétablie afin d'obtenir un modèle de l'environnement. Ensuite, depuis ce même point de départ, la micromouse sera amenée à se rendre au même point d'arrivée en empruntant un chemin unique qu'il aura jugé être le plus court ;

3. Enfin, sur ce même environnement physique, une démonstration mettant en scène nos quatre souris sera faite. Les micromouses se confronteront par équipe de deux dans un jeu <À DEFINIR – à priori sur un jeu type Pacman : i.e. 1v3 dans lequel la souris individuelle devra parcourir toutes les cases du labyrinthe sans rencontrer une des autres souris dans son chemin ou bien capture de drapeau >. La mise en scénario multi-micromouse pourra également être simulée depuis le module de simulation.

1.4 Organisation du rapport

— Un cahier des charges (**??**) de notre projet sera disponible. Il tirera un portrait de ce qui a déjà été fait dans le domaine dans une étude de marché. Par ailleurs, une liste des fonctionnalités attendues ainsi qu'une description de la conception globale du projet sera tenue. Cette dernière rendra compte des différentes vues (utilisateur, technique et logicielle) du projet et pourront mettre en évidence les différentes problématiques qui seront traitées au cours des chapitres techniques. Enfin, une dernière section concernera l'environnement de travail au cours de laquelle sera dressée une liste exhaustive du matériel ainsi que des logiciels et outils utilisés.

— Les problématiques soulevées précédemment dans le cahier des charges seront respectivement traitées dans des sections techniques. Ces sections, qui constituent les chapitres techniques, auront pour sujets principaux l'intelligence embarquée dans nos micromouses comprenant les algorithmes de mappage et de plus court chemin (**??**) ainsi que les algorithmes de contrôle (**??**) mettant en oeuvres les capteurs du véhicule ; les protocoles réseaux de communication entre les différentes briques du projets (**??**).

— Un chapitre concernant le rendu final (**??**) figurera également sur ce présent rapport. Il aura vocation à fournir le reflet du résultat attendu du projet pour l'utilisateur final. Ainsi, il présentera les différents outils terminaux que l'utilisateur sera amené à manipuler pour faire fonctionner le produit ainsi qu'une batterie de tests qui témoignent du bon fonctionnement du dispositif livré.

— Enfin, un chapitre sera consacré à la gestion projet (**??**) dans laquelle figureront les méthodes de travail employées ainsi que la répartition des tâches au cours de ce projet. Par ailleurs, une partie de ce chapitre aura une vocation didactique sur le sujet. Ainsi, seront évoqués <Choix1> ainsi que <Choix2>.

Chapitre 2

Présentation et spécification du projet

Ce chapitre présente le cahier des charges (CDC) du projet et les spécifications techniques, il contient les principaux éléments nécessaires pour comprendre le positionnement marketing du futur produit et la conception technique. Il mentionne également des informations relatives au développement du produit dans sa globalité et son architecture.

2.1 Étude du marché

Une compétition Micromouse est un évènement durant lequel plusieurs équipes s'affrontent en opposant leur robot souris, l'objectif étant de résoudre un labyrinthe le plus rapidement possible. Ces compétitions sont organisées depuis la fin des années 1970 et ont lieu un peu partout autour du monde.

Historique et Origine : Le magazine "IEE Spectrum magazine" est à l'origine de l'apparition des compétitions Micromouse. C'est un magazine anglophone qui vise à couvrir les tendances et avancées majeures dans les domaines des technologies, de l'ingénierie et des sciences. En 1977, le magazine met au défi ses lecteurs en leurs proposant de concevoir et construire un robot "Micromouse". Le robot devait agir selon sa propre logique et résoudre un labyrinthe imaginé par les rédacteurs du magazine IEE Spectrum.

Cette compétition fut appelée 'The Amazing Micromouse Competition'. Elle prit place à New York lors de la conférence nationale sur l'informatique en 1979. Cet évènement fut couvert par de grands médias tels que les chaînes CBS et ABC ainsi que le journal The New York Times. Le grand succès de cette première édition ainsi que sa large médiatisation participa au gain de popularité des compétitions Micromouse pour les années à venir. En quelques années, le défi des Micromouses était devenu un événement mondial.

En 1980, le premier concours européen eu lieu à Londres, suivi un an plus tard par un une compétition organisée à Paris. Le Japon a annoncé l'organisation du premier tournoi mondial de Micromouse qui aura lieu à Tsubaka en août 1985. La même année, l'IEE (devenue l'Institution of Engineering and Technology) a organisé un concours international à Londres. Au début des années 1990 des clubs Micromouse ont commencé à faire leur apparition au sein des écoles et universités. La "IEE Micromouse Competition" bénéficie au aujourd'hui d'une grande popularité auprès des étudiants du domaine informatique ou électronique.

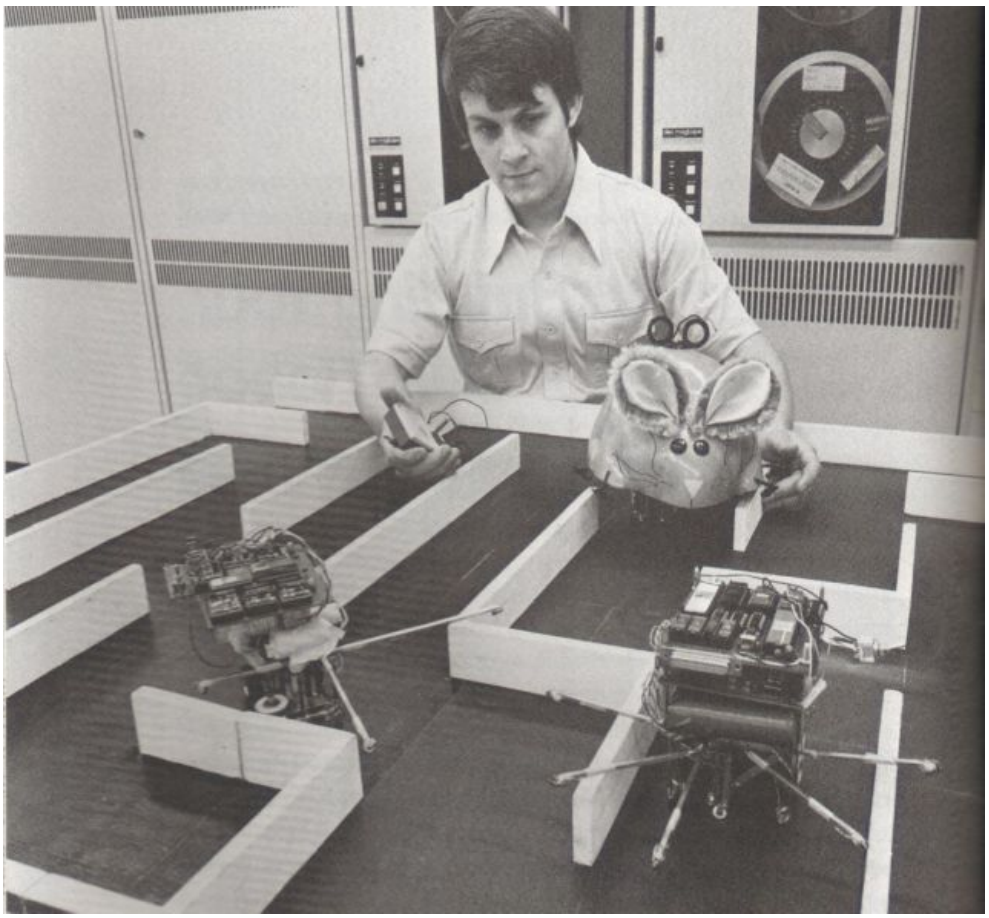


FIGURE 2.1 – Ancienne génération de Micromouse (Pacific Northwest National Laborator, 1980)

Organistaions des compétitions Au vu du nombre important de compétitions Micromouse ayant lieu chaque année, certaines règles et conditions peuvent varier d'une compétition à l'autre. Néanmoins, il existe des éléments généralement communs à toutes les compétitions :

— Le labyrinthe standard de micromouse mesure environ $2.5m^2$ et consiste en une grille de cellules (matrice) de 16 sur 16. Chaque micromouse est autorisée à effectuer un certain nombre de recherches afin de déterminer le chemin le plus court vers l'objectif. La micromouse devra garder une trace de sa position, découvrir les murs en explorant le labyrinthe et détecter quand elle a atteint son objectif (sortie du labyrinthe).

— La notation est basée à la fois sur la course la plus rapide et sur le temps total consommé pour toutes les courses. Les concurrents n'ont pas le droit de communiquer avec leur micromouse. Il existe de nombreuses versions de règles selon la compétition, et il existe un certain nombre de variations mineures sur la façon dont le score de la souris est déterminé.

Il existe un plétore de vidéos disponibles sur Internet qui illustrent le déroulement des compétitions Microumouse partout à travers le globe. Certaines de ses vidéos cumulent plusieurs millions de vus. Ces vidéos ont grandement susictées notres interêt, et nous ont poussées à implementer notre propre version de la Micromouse en tant que projet de synthèse.

2.2 Fonctionnalités attendues

Dans la section [??] le lecteur peut observer que le produit proposé fournit plusieurs options ie :fonctionnalités pour tester et manipuler différentes entités dans les deux environnements, physique et simulé dans cette section nous allons introduire ces fonctionnalités et expliquer les cas d'utilisation de ces derniers.

2.2.1 Vue d'ensemble du système

Le produit permet à l'utilisateur de simuler un véhicule autonome se déplaçant dans un labyrinthe et en essayant de trancher le point d'arriver, le labyrinthe est créé soit manuellement ou en insérant un fichier de configuration, ainsi la simulation pourra se lancer et l'utilisateur pourra observer le déplacement de la micro mouse et en même temps d'observer le journal et les statistiques de la simulation qui sont modifiées et stockées au fur et à mesure de la simulation, ainsi que les valeurs des différents composants de l'engin.

En outre, l'utilisateur du produit peut modifier le labyrinthe réel et d'y mettre le véhicule dedans pour lancer une observation physique, ainsi qu'il pourra voir sur sa machine sur un autre logiciel de visualisation le déplacement et les valeurs retournées par l'engin.

Le schéma suivant illustre le diagramme de cas d'utilisation du produit.

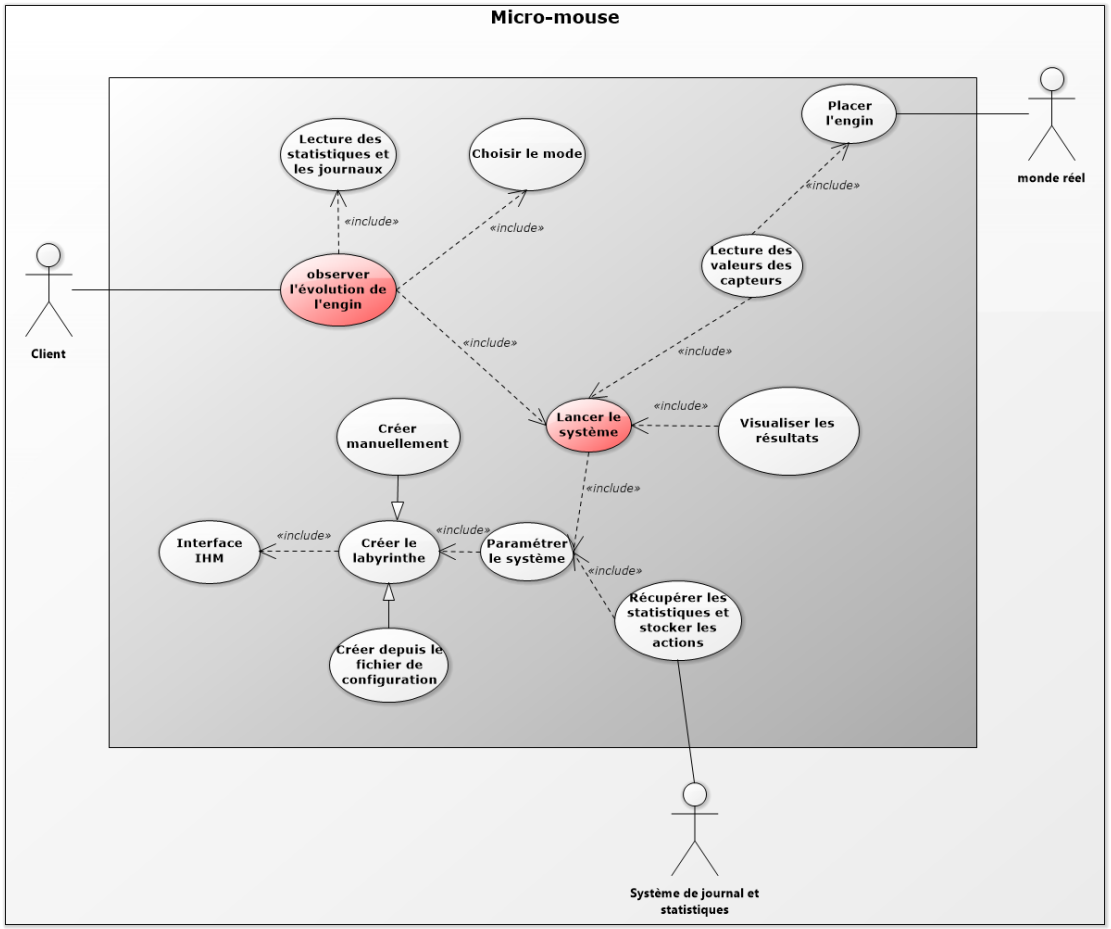


FIGURE 2.2 – Le diagramme de cas d'utilisation du système

2.2.2 Services fournis par le produit

Simulation : La simulation offre à l'utilisateur un ensemble de fonctionnalités pour paramétrer le monde, comme la création du labyrinthe soit avec un fichier de configuration ou manuellement et le choix de la complexité de l'algorithme de recherche du plus court chemin, en outre l'utilisateur peut visualiser le journal de la simulation et les statistiques sur la partie information sur la simulation, finalement l'utilisateur pourra observer les valeurs des différents composants de la micro mouse.

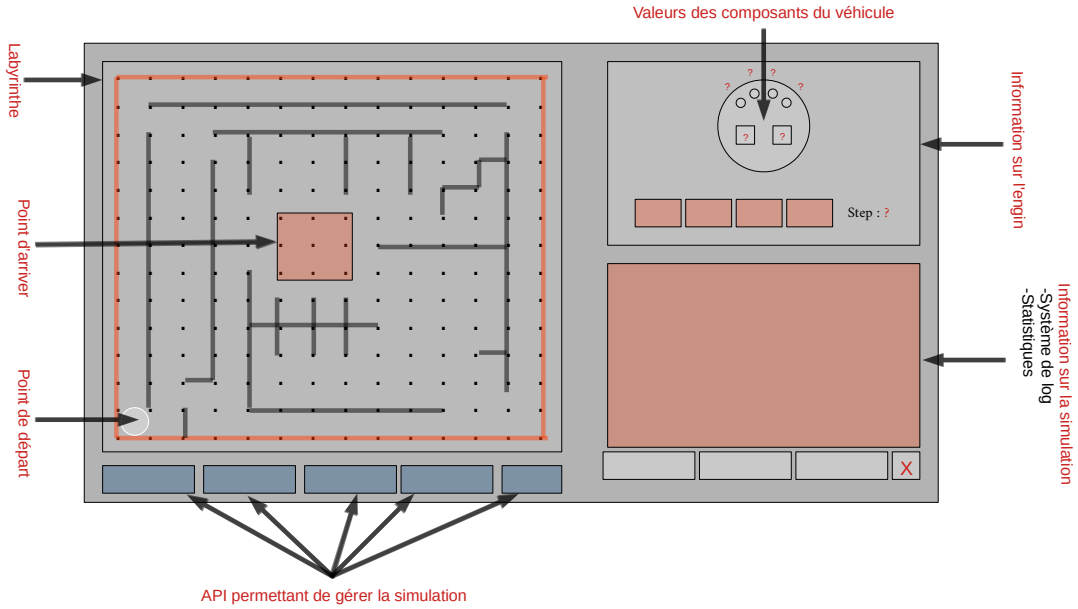


FIGURE 2.3 – Interface homme-machine pour la simulation

Véhicule réel : Notre système sera réalisé sur un prototype minimisé de micromouse appelé quarter-micromouse, le labyrinthe est constitué d'un quadrillage de cellules 8×8 de 180 mm de côté avec des murs de 50 mm de haut les souris sont des robots complètement autonomes qui doivent trouver leur chemin d'une position de départ prédéterminée à la zone centrale du labyrinthe sans aide. La souris doit savoir où elle se trouve, découvrir les murs en explorant, cartographier le labyrinthe et détecter quand elle a atteint son but. Une fois l'objectif atteint, la souris effectuera généralement des recherches supplémentaires dans le labyrinthe jusqu'à ce qu'elle ait trouvé un itinéraire optimal du début à la fin. Une fois que l'itinéraire optimal a été trouvé, la souris exécute cet itinéraire dans les plus brefs délais.

2.2.3 Fonctionnalités supplémentaires

Visualisation en temps réel : La visualisation des résultats au fur et à mesure du temps, au moment du lancement du système l'utilisateur peut lancer un logiciel, c'est le logiciel de visualisation des résultats de l'expérience, il permet à l'utilisateur de visualiser la vue du véhicule et les valeurs des capteurs au fur et à mesure de l'exploration, aussi de visualiser le journal de la simulation et les statistiques.

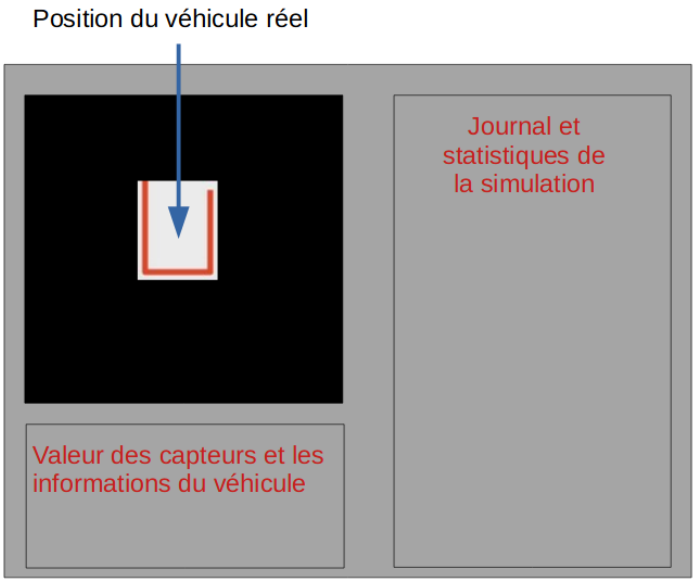


FIGURE 2.4 – Maquette du logiciel de visualisation

Multi-véhicule : Un mini-jeu est mis en place pour les deux univers, simulé et réel permettant d'introduire la notion de communication entre différents véhicules, le jeu consiste à parcourir toutes les cases du labyrinthe et à la fin se situer dans la case d'arriver par un véhicule autonome intelligent en évitant les autres véhicules gradient communiquant entre eux. Ceci en utilisant les modules déjà abordés tels que la simulation et l'univers réel.

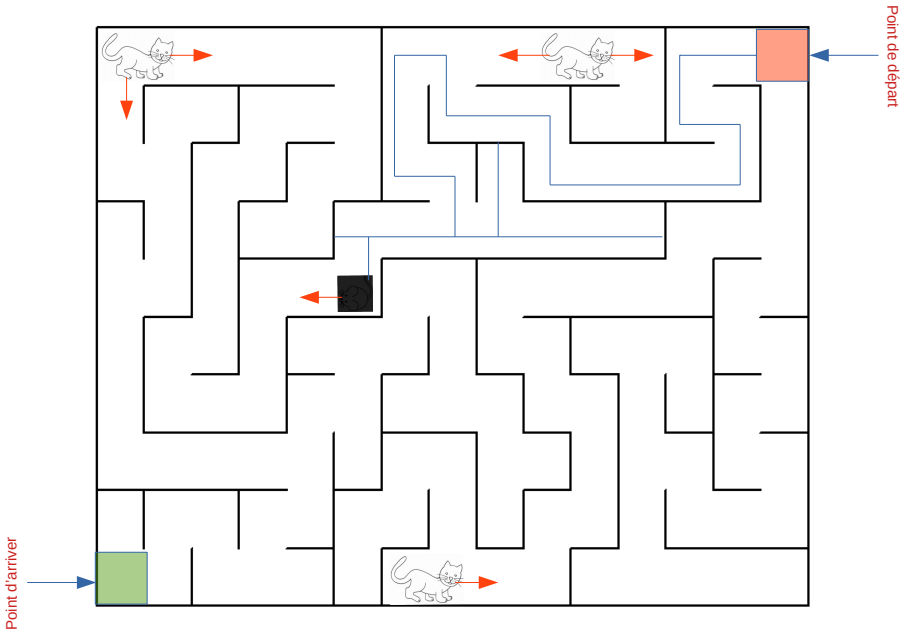


FIGURE 2.5 – Pac-Man avec l'entité micro mouse (Pac-Micro)

La figure [??] représente une illustration du mini-jeu, les trois chats représentent les gardiens du labyrinthe et le carré noir représente le véhicule intelligent qui doit résoudre le labyrinthe en parcourant toutes les cases tout en évitant de croiser un gradient.

2.3 Conception globale du projet

après avoir illustré les différentes fonctionnalités du système sur la section [**??**], il va falloir expliquer plus en détails la conception de ces fonctionnalités, en séparent la vue pour l'utilisateur on masquera les détails techniques informatiques et on illustrera seulement les composants ou modules fonctionnels, de l'architecturer technique et logiciel on va présenter les différentes parties techniques nécessaires permettant de réaliser les fonctionnalités décrites dans la section précédente [**??**]

2.3.1 Vue pour l'utilisateur

Le système offre à l'utilisateur deux choix pour l'exploiter :

1. Le système réalisé permet à l'utilisateur d'observer le fonctionnement et l'évolution de la micro mouse ie :véhicule autonome dans sans monde un labyrinthe de 8x8 [**??**] case dans un monde physique, pour but de résoudre ce labyrinthe et trouver le plus court chemin du point de départ au point final ;

   — **Le labyrinthe** est un carré de 1,44 m2 de surface composé de seize « cellules » carrées de 180 millimètres de côté et de 50 millimètres de haut. Il peut y avoir 8×8 cellules.

   — **Le robot** ne doit pas faire plus de 250 millimètres de large et de long (il n'y a pas de hauteur limite).
2. Le système permet aussi de simuler les fonctionnalités physiques dans un monde simulé, un labyrinthe de taille variante et des chemins à définir par l'utilisateur, aussi on peut charger un fichier de configuration pour créer le labyrinthe ;

   — **Le labyrinthe** peut être créé par l'utilisateur manuellement ou bien en passant un fichier de configuration.

   — **Le robot** Le robot est défini comme une entité qui se déplace dans le labyrinthe, il est sous la forme d'un cercle, surface du cercle ne dépasse pas la surface d'une case du labyrinthe.

Finalement, l'utilisateur peut observer en temps réel l'évolution de la micro mouse dans son univers, cette partie permet aussi d'afficher les valeurs des différents composants du véhicule comme les capteurs infrarouges et l'accéléromètre ainsi que d'autres informations sur la simulation.

2.3.2 Architecture technique

la micromouse sera composée de nombreuses parties electronique qui vont lui permettent de fonctionner.

Le plus important d'entre eux est le **PCB** (Printed Circuit Board) qui est le corps où tous les autres composants seront soudés, il sert aussi à les interconnecter et leur faire communiquer. Pour cela, nous avons trouvé un design open source [**?**] conçu pour les micro-souris qui contient

tous les modules nécessaires pour notre produit final.

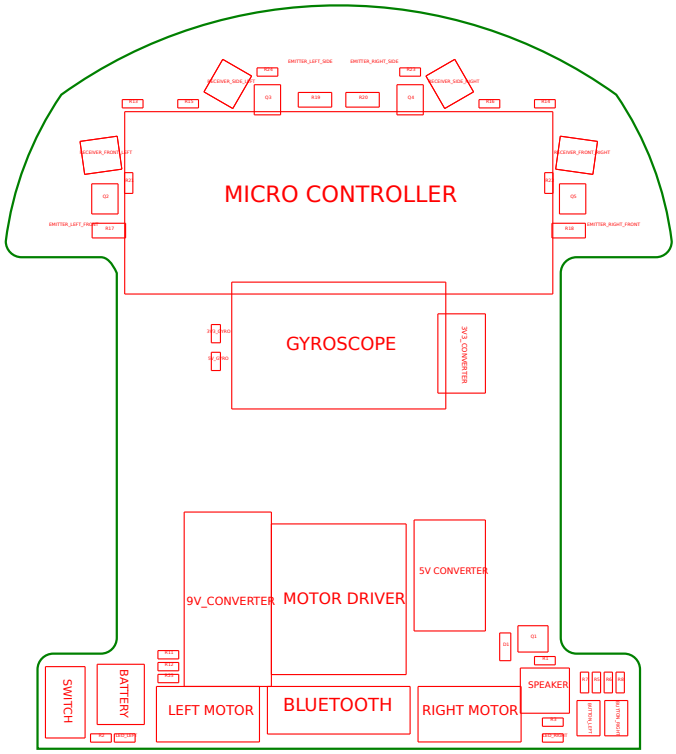


FIGURE 2.6 – digramme de composantes dans le PCB

Le cerveau de notre véhicule sera un micro contrôleur **STM32F103** [?] également connu sous le nom de Blue Pill, il a 64KB/128KB de mémoire flash et 20KB de RAM fonctionnant à 72MHz.

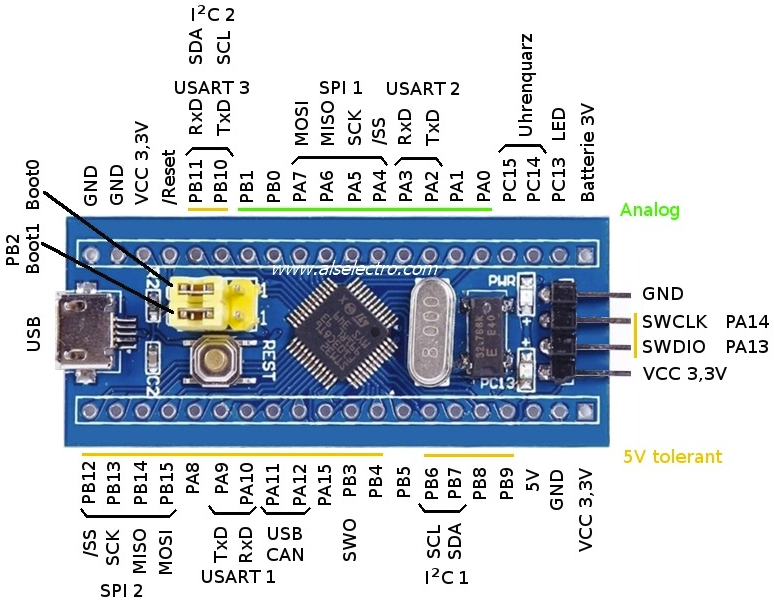


FIGURE 2.7 – Microcontrôleur, 32bit, STM32

On passe ensuite aux capteurs, qui sont les composants utilisés pour recevoir les informa-

tions du monde extérieur, qui est dans ce cas le labyrinthe.

Il y a trois types que nous allons utiliser, le premier est le **gyroscope** qui est employé pour détecter le changement de direction et l'angle de rotation, c'est important pour bien tracer le labyrinthe puisque le véhicule devra faire beaucoup de manœuvres dans le labyrinthe.

Ensuite, il y a **l'accéléromètre** qui, comme son nom l'indique, détecte l'accélération, et avec le gyroscope il peut trouver la position ou plutôt les changements de position, de micro-mouse, ce capteur alors sera le seul moyen de localisation pour notre véhicule.

Ces deux capteurs précédents sont généralement regroupés sur une seule circuit électronique parce qu'ils sont sovent utilisés ensemble et ont des fonctionnalités similaires, cela nous économise également de la place sur le PCB.

Enfin, il y a les **capteurs infrarouge** que nous utiliserons pour obtenir la distance du véhicule des murs et autres obstacles, ils seront placés sur toutes les directions pour évaluer plus précisément les alentours et avoir une meilleure cartographie du labyrinthe.

Nous disposerons également d'un **module bluetooth**, l'outil de communication principal. Lequel est utilisé pour envoyer des données en direct à l'interface afin de déboguer et de visualiser le processus de Mapping, ainsi que pour communiquer avec le simulateur de labyrinthe pour recevoir les signaux des capteurs artificiels.

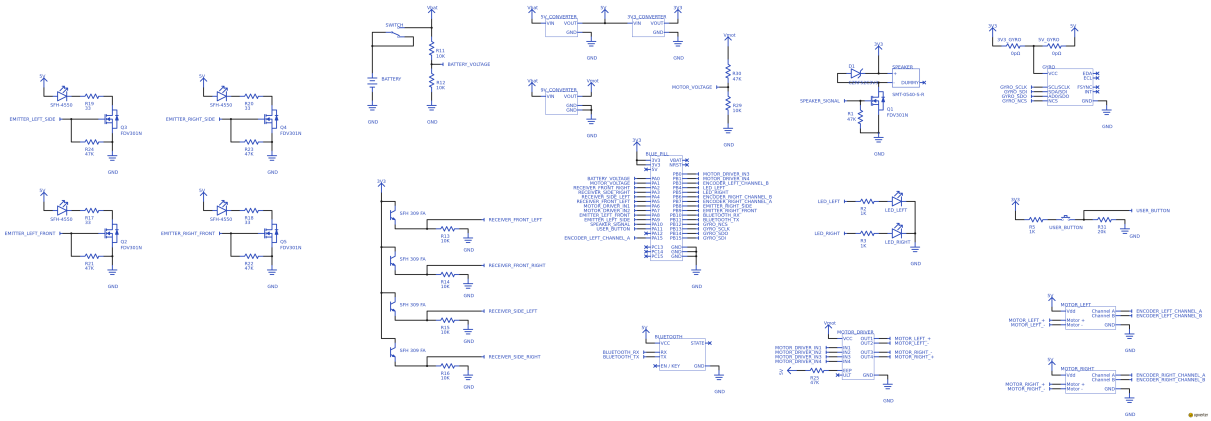


FIGURE 2.8 – Ensemble des composantes électroniques

Le contrôle du véhicule se fera avec deux **moteurs falhaber** [?] des deux côtés du véhicule. Les roues et leurs essieux sont imprimés en 3d pour s'adapter au design du circuit. Aucun volant ne sera utilisé à cause du poids supplémentaire qu'ils apportent et aussi parce que les deux que nous avons déjà peuvent faire des manœuvres très précises avec un code conducteur minimal.

Il existe d'autres petits composants électriques ou **SMD**s (Surface Mount Devices) tels que les résistances, condensateurs, interrupteurs, diodes, LEDs etc... qui sont utilisés pour contrôler le courant électrique sur le PCB.

2.3.3 Architecture logicielle

Contrôle du véhicule : à un moment de la simulation notre véhicule va décider de faire des virages, des marches arrière ...etc., des prises de décisions floues.

Pour gérer ça on utilise la logique floue. La logique floue (fuzzy logic, en anglais) est une logique polyvalente où les valeurs de vérité des variables - au lieu d'être vrai ou faux - sont des réels entre 0 et 1. En ce sens, elle étend la logique booléenne classique avec des valeurs de vérités partielles1. Elle consiste à tenir compte de divers facteurs numériques pour aboutir à une décision qu'on souhaite acceptable.

Mappage et navigation : Le véhicule a le droit de faire un repérage du labyrinthe pour déterminer le chemin vers le point d'arrivée, puis de retourner au point de départ pour réaliser l'épreuve en utilisant le chemin le plus court qu'elles ont déterminé.

Ce qui nous introduit au mappage et navigation du véhicule ie :La localisation et cartographie simultanées ; La localisation et cartographie simultanées connue en anglais sous le nom de SLAM (simultaneous localization and mapping) ou CML (concurrent mapping and localization), consiste, pour un robot ou véhicule autonome, à simultanément construire ou améliorer une carte de son environnement et de s'y localiser.

Calcul du plus court chemin : après avoir fait le mappage le robot se place sur la case départ, en le relançant le robot ne va pas refaire les mêmes pas il va calculer le plus court chemin entre la case de départ et la case d'arriver en utilisant toutes les données récupérer à l'étape de mappage et en utilisant un algorithme de calcul de plus court chemin.

En théorie des graphes, le problème de plus court chemin est le problème algorithmique qui consiste à trouver un chemin d'un sommet à un autre de façon que la somme des poids des arcs de ce chemin soit minimale.

Communication : Pour déboguer et visualiser les données du véhicule ou bien pour recevoir les signaux artificiels des capteurs de la simulation une communication entre les composants est nécessaire, ainsi le transfert de données se fait en temps réel.

2.4 Problématiques identifiées et solutions envisagées

D'un coup d'œil, plusieurs problèmes majeurs se posent d'un point de vue technique du projet. Dans cette section, nous décrivons ces problèmes et les solutions que nous y apportons.

Pour notre véhicule, les seules données qu'il reçoit du monde extérieur sont celles des capteurs, ce sont des signaux en temps réel des changements de rotation et d'accélération, ainsi que les distances des murs qui sont les principales informations que nous allons utiliser pour le traçage de la mappe.

Pour transformer ces signaux en une carte 2D du labyrinthe, nous allons devoir commencer par déterminer la position et la direction du véhicule. Pour cela, nous utiliserons les informations du gyroscope afin de garder une copie de la position et de la direction et de les actualiser a chaque iteration en utilisant les données reçues.

Après avoir obtenu ces entrées, nous allons les utiliser avec les informations restantes des capteurs infrarouges pour faire la mappage proprement dite, nous pouvons utiliser un algorithme de Flood fill pour déterminer la géométrie des murs et/ou des obstacles. Après un peu

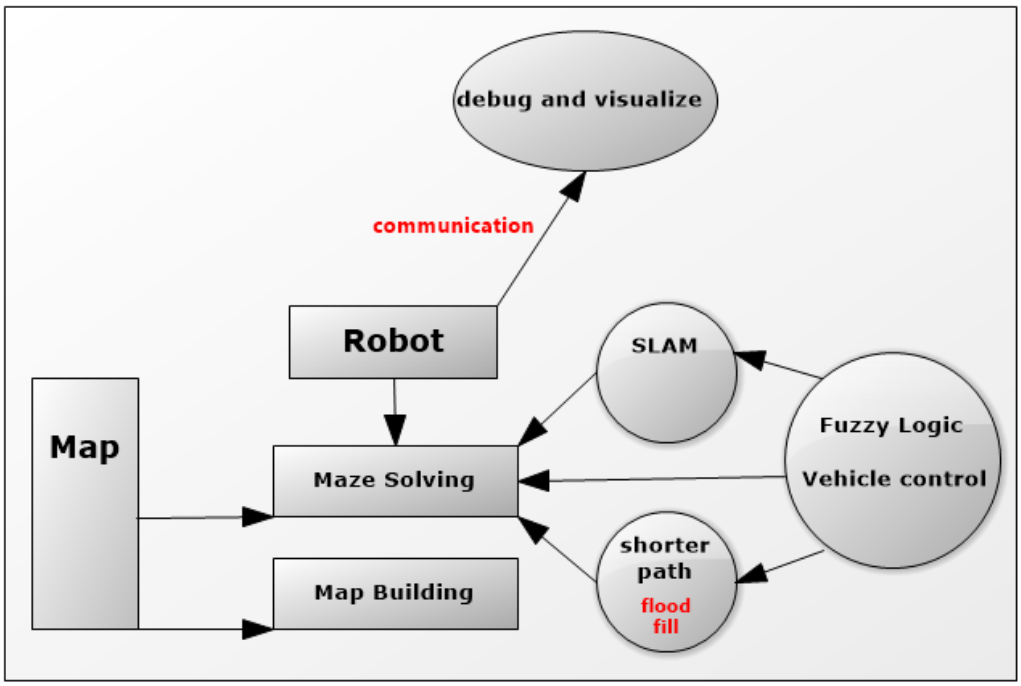


FIGURE 2.9 – schéma illustrant l'enchaînement des différentes parties

de nettoyage et interpolation nous devrions avoir une carte finale du labyrinthe.

Une fois le tour de mapping est terminé, nous devrons trouver le chemin le plus court entre la position du véhicule et le point d'arrivée. Une solution qui vient à l'esprit est l'algorithme A* qu'il est souvent utilisé dans les labyrinthes et qui est alors idéale pour notre projet. Bien sûr, l'algorithme doit être modifié pour tenir compte des erreurs occasionnelles de contrôle qui surviennent à cause des irrégularités du monde physique.

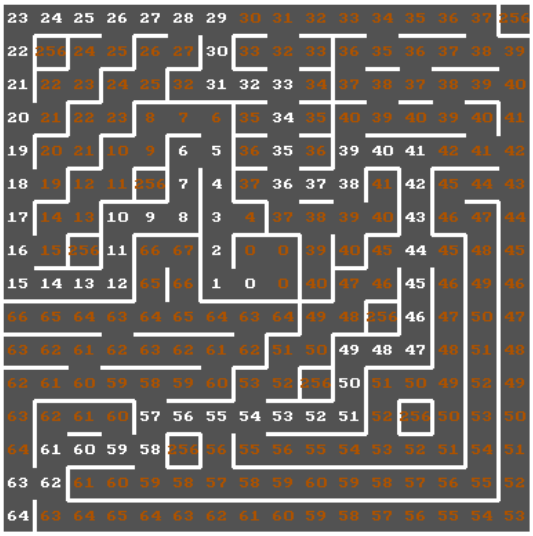


FIGURE 2.10 – Algorithme de Flood fill, Micromouse competition US 1982

2.4.1 Contrôle du véhicule

Notre véhicule se compose de deux moteurs que nous devrons contrôler pour parcourir les chemins avec fluidité, il devra faire des virages et éviter les collisions avec les murs et autres obstacles, tout en suivant le chemin créé pour lui.

Pour résoudre ces problèmes, nous allons utiliser la logique floue qui servira à générer des signaux pour les puissances des moteurs en temps réel. La façon dont cela fonctionne est que nous créons un moteur d'inférence et lui fournissons des règles qui décrivent la façon dont la véhicule doit se comporter en se déplaçant, et le moteur utilisera ces règles ainsi que les données du capteurs pour générer les signaux de sortie des moteurs qui seront sa décision finale.

2.4.2 Communication

Un problème qui se pose à cause des besoins de certaines fonctionnalités du projet est la communication, que ce soit pour déboguer et visualiser les données du véhicule ou bien pour recevoir les signaux artificiels des capteurs de la simulation.

Tous deux auront besoin de leur propre protocole de communication respectif à être efficaces, fiables et rapides puisque le transfert de données se fait en temps réel, avec les ressources limitées de micro contrôleur. et pour cela nous allons utiliser une communication serial entre le module Bluetooth de Micromouse et les autres parties communicantes.

2.4.3 Génération du labyrinthe

Le labyrinthe constitue un élément import du projet, ce dernier représente l'environnement externe dans lequelle va se mouvoir notre Micromouse.

Il existe un certain nombre de problématiques liées à la construction de notre labyrinthe. Des problématiques logiques :

- Quel algorithme utiliser afin de générer notre labyrinthe ?.
- Quelle sont les spécificités architecturales que notre labyrinthe devra respectre ?.
- Ainsi que des problématique liées au monde physique : Quel matériaux utiliser pour la construction de notre labyrinthe ? (prendre en compte l'adhérence avec les roues de la Micromouse).

2.4.4 Mappage et navigation

2.5 Environnement de travail

Plusieurs outils et matériels ont été utilisés pour développer les différents composants de la micromouse. Ici ils sont décrits et leurs rôles dans la réalisation de chacune de ces parties.

2.5.1 logiciels et environnements de développement

Le simulateur de labyrinthe et l'interface temps réel de la micro mouse sont écrits en langage de programmation Processing. Le contrôleur principal de l'Arduino est écrit en C.

Processing est un langage qui se concentre sur les graphiques et les interfaces 2D et 3D, il sera utilisé pour créer des courbes et des animations en temps réel très claires et informatives,

ainsi que des contrôles simples et faciles à utiliser. il fournit des bibliothèques pour la communication serial pour le transfert de signals depuis et vers le micro contrôleur.

Le **IDE**[1] **Processing** nous permettra d'organiser le projet, etd'exécuter le code de Processing. Il a également un débogueur intégré etd'autres outils de developpement, et une interface simple pour ajouter les bibliothèques et extensions de langage.

Box2D est une librairie de simulation physique 2D écrite pour C++, mais il existe des frameworks et wrappers pour cela en Java et Processing. elle sera utilisée pour simuler le véhicule et ses interactions avec son environnement. et pour générer en temps réel des signaux de capteurs artificiels pour tester la micro souris.

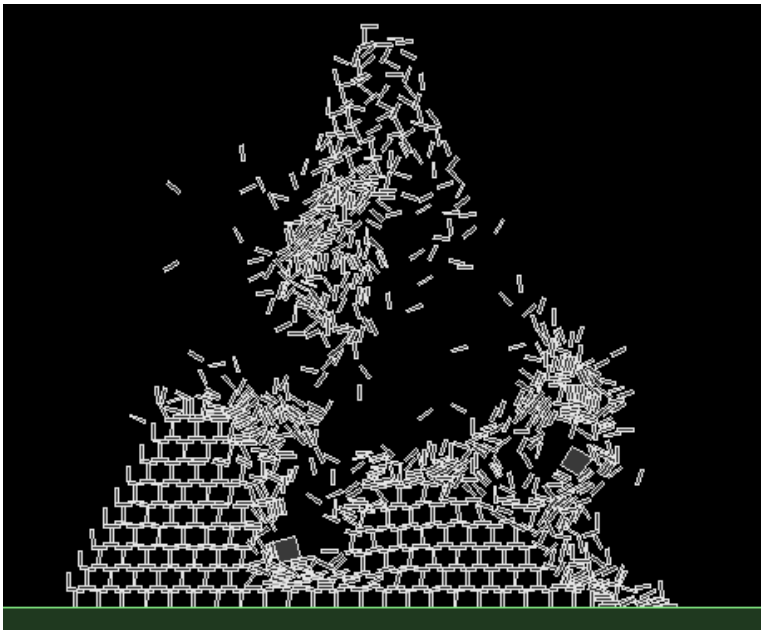


FIGURE 2.11 – Exemple d'usage de Box2D, Wikimedia creative commons

Le **IDE Arduino** sera utilisé pour développer le microcontrôleur principal en C, il dispose d'outils qui permettent la compilation et l'ectiture des sorties binaires dans le micro contrôleur.

2.5.2 Matériel informatique

Dans ce projet, nous avions une carte de circuit imprimé utilisée pour connecter les pièces de notre véhicule, celle qui devait être imprimée à l'aide d'une **imprimante PCB** qui grave les circuits de la carte pour pouvoir ensuite souder les pièces ensemble. Nous avons également des pièces qui nécessitent une **imprimante 3D** pour fabriquer les formes des roues et essieux du véhicule pour qu'ils s'adaptent au design du corps.

1. Integrated Development Environment

Chapitre 3

Construction du labyrinthe

Dans ce troisième chapitre, on traite des différents aspects liés à la construction et la génération d'un labyrinthe. Après une présentation de la problématique, on effectuera un état de l'art des solutions existantes, présenterons les algorithmes choisies et impléments et compléterons par une série de tests pour certifier nos solutions.

3.1 Qu'est-ce qu'un labyrinthe ?

3.1.1 Définition d'un labyrinthe

Un labyrinthe est une structure complexe de passages reliés entre-eux. L'objectif du solveur est de passer d'un point de départ à un point d'arrivée, c'est-à-dire trouver le passage reliant ces deux points. Le labyrinthe est une énigme qui teste l'intelligence, la réflexion, et la rapidité d'exécutions du solveur. D'un point de vu mathématique, les labyrinthes peuvent êtres représentés comme étant des surfaces connexes.

3.1.2 Histoire et application

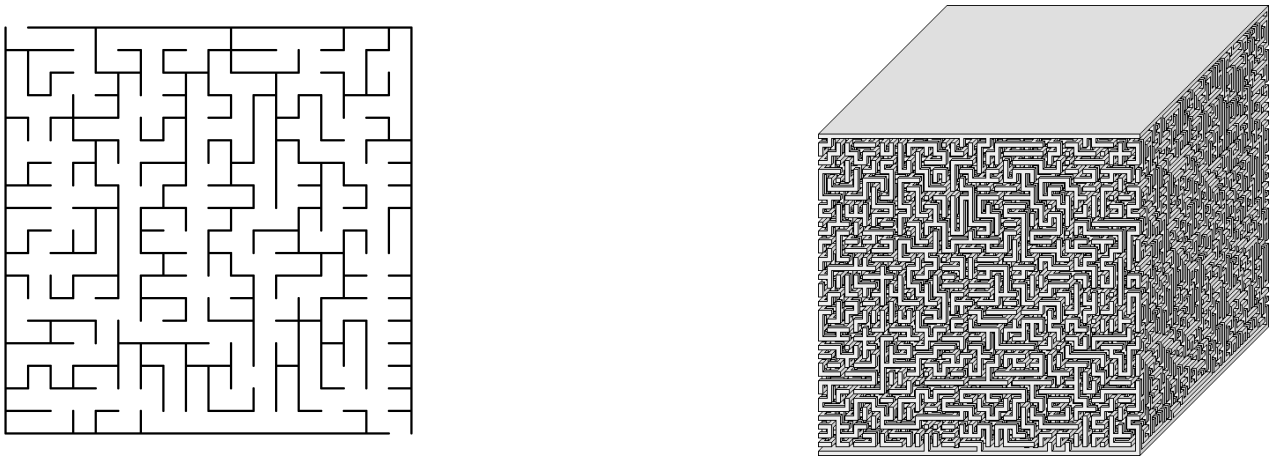
Le mot labyrinthe trouve son origine dans la mythologie grecque, c'était une structure constitué de galeries, construite par Dédale afin d'y enfermer le Minotaure. Le divertissement et l'entraînement cérébral peuvent être considérés comme les principaux objectifs d'application d'un labyrinthe. En ce qui concerne le domaine scientifique, les labyrinthes peuvent être vu comme des supports pour effectuer des démonstrations robotiques. Le concept de labyrinthe a permis la naissance de concours robotiques comme les compétitions Micromouses ou l'objectif est de résoudre une énigme (un labyrinthe) en ayant recours à différents algorithmes d'intelligence artificielle.

3.2 Classification d'un labyrinthe

Les labyrinthes (et les algorithmes responsables de leur génération) peuvent être organisés selon trois critères de classifications différents. Ces critères sont : la dimension, la topologie, et la tessellation. Un labyrinthe peut prendre un objet d'un de ses classes dans n'importe quelle combinaison.

3.2.1 Dimension d'un labyrinthe

La dimension d'un labyrinthe corresponds à l'espace dimensionnel couvert par le labyrinthe. Les labyrinthes peuvent êtres de dimensionnalité 2, 3 ou tout autre dimension supérieure.



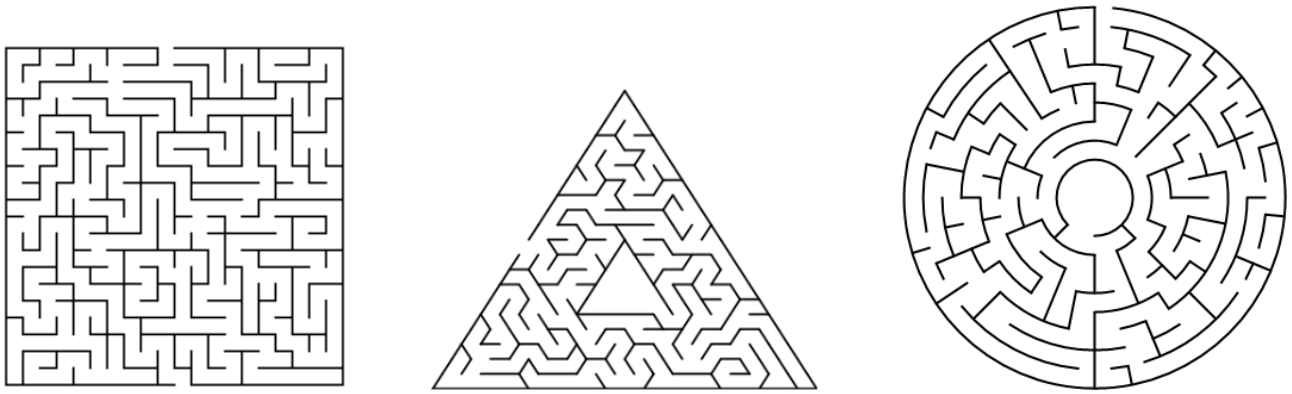
(a) Labyrinthe en 2 dimensions. (b) Labyrinthe en 3 dimensions.

FIGURE 3.1 – Exemple d'un labyrinthe 2D et d'un labyrinthe 3D

3.2.2 Tessellation d'un labyrinthe

La classe de tessellation est la géométrie des cellules individuelles qui composent le labyrinthe. Il existe de nombreux types de tessellation, on citera notamment :

- **Tesellation orthogonale :** Il s'agit d'une grille rectangulaire standard où les cellules ont des passages qui se coupent à angle droit formant des cellules sous forme de carrés.
- **Tesellation delta :** Un labyrinthe à tessellation delta est un composé de triangles imbriqués, où chaque cellule peut avoir jusqu'à trois passages connectés.
- **Tesellation theta :** Un labyrinthe à tessellation theta est composé de cercles concentriques. Les cellules ont généralement quatre connexions de passage possibles, mais peuvent en avoir plus en raison du plus grand nombre de cellules dans les anneaux externes.



(a) Labyrinthe orthogonal. (b) Labyrinthe delta. (c) Labyrinthe theta.

FIGURE 3.2 – Exemple de labyrinthes avec différentes classes de tessellation.

3.2.3 Topologie d'un labyrinthe

D'un point de vu mathématique, un labyrinthe est définie comme étant une surface connexe pouvant avoir deux types de topologies : topologie simple et topologie comportant des anneaux. Cette différence dans le type de topologie conduit à une distinction des labyrinthes en deux catégories : Les labyrinthes parfaits et les labyrinthes imparfaits.

Labyrinthe parfait

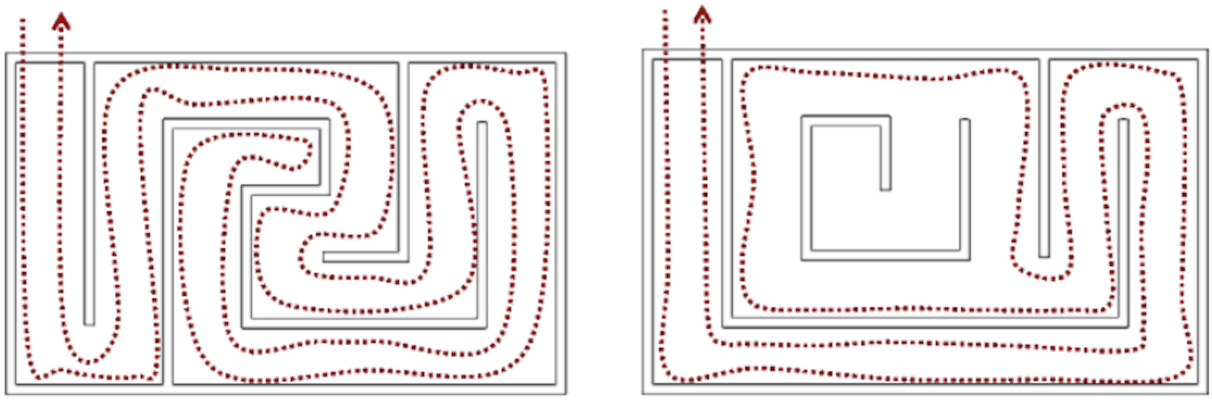
Afin qu'un labyrinthe soit labélisé comme étant parfait, ce dernier doit remplir deux conditions :

— Ne contient pas de cycles.

— Il existe un unique chemin entre la cellule de départ et la cellule d'arrivée du labyrinthe. Plus généralement, quelque soit deux cellules sélectionnées dans notre labyrinthe, le chemin entre ces deux cellules doit être unique.

Labyrinthe imparfait

Un labyrinthe qui ne remplit pas les conditions pour être labélisé comme parfait est dit imparfait. Les labyrinthes imparfaits peuvent donc contenir des boucles, des îlots ou des cellules inaccessibles.



(a) Labyrinthe parfait.

(b) Labyrinthe imparfait.

FIGURE 3.3 – Exemple d'un labyrinthe parfait et d'un labyrinthe imparfait.

Dans la suite de ce rapport, nous considérons l'ensemble de nos labyrinthes comme étant de dimension 2 et possédant une tessellation orthogonale. La distinction se feras sur le critère de la topologie, on distinguera alors deux types de labyrinthes : les labyrinthes parfaits et les labyrinthes imparfaits.

3.3 Introduction à la génération de labyrinthes

Les algorithmes utilisés pour la génération de labyrinthes suivent un ordre d'étapes prédéfini. Afin de créer des labyrinthes structurés de manière aléatoire, une ou plusieurs étapes de l'algorithme doivent être randomisées (c'est-à-dire que la prise de décision au sein de l'étape doit utiliser une fonction renvoyant un résultat aléatoire). Il existe deux approches algorithmiques générales utilisées pour la génération automatisée des labyrinthes. Ces approches sont : l'ajout de murs et la sculpture de passages.

— **Ajout de murs :** Dans cette approche, on se base sur l'ajout de murs pour générer progressivement notre labyrinthe. On démarre d'un labyrinthe vide, puis l'algorithme va placer au fur et à mesure des murs à des positions spécifiques.

— **Sculpture de passages :** Les algorithmes basés sur cette approche se concentrent sur les chemins et le positionnement des cellules du labyrinthe. On démarre d'une matrice de cellules (une grille complètement remplie de murs), puis l'algorithme de sculpture de passages va alors creuser à travers la grille et détruire certains murs reliant les différentes cellules, ce processus itérative générera au final un labyrinthe.

3.3.1 Rappels sur la théorie des graphes

Cette sous-section présente certains principes fondamentaux de la théorie des graphes sur lesquels sont basé les algorithmes de génération de labyrinthes.

Graphe

Un graphe est le concept fondamental dans la théorie des graphes, ce dernier est définie comme étant un triplet $G = (V, E, \epsilon)$, avec V représentant l'ensemble des sommets, E l'ensemble des arêtes et ϵ une fonction de mappage appelée relation d'incidence tel que $\epsilon : E \to V^2$

Arbre

Un arbre est un graphe connexe non orienté ne contenant pas de cycles, c'est-à-dire qu'il existe exactement un seul et unique chemin entre chaque paire de sommets du graphe. Un graphe connexe avec n sommets et $n - 1$ arêtes est un arbre. La suppression de toute les arête entraînerait la perte de connexité du graphe, et l'ajout d'une arête créerait un cycle au sein du graphe.

Arbre couvrant

Soit G un graphe. L'arbre couvrant de G est le sous graphe qui est connexe, sans cycle et contient tous les sommets de G.

3.3.2 Théorie des graphes et labyrinthes.

Sur la figure ci-dessous on peut apercevoir deux labyrinthes. Celui de gauche est un labyrinthe parfait, celui de droite un labyrinthe imparfait (car il contient des boucles). Il existe une autre approche pour visualiser nos labyrinthes. Au lieu de penser aux murs constituants le labyrinthe, nous pouvons penser aux chemins entre les cellules. Les chemins à l'intérieur du labyrinthe forment un graphe. Le graphe de chaque labyrinthe est illustré ci-dessous en rouge.

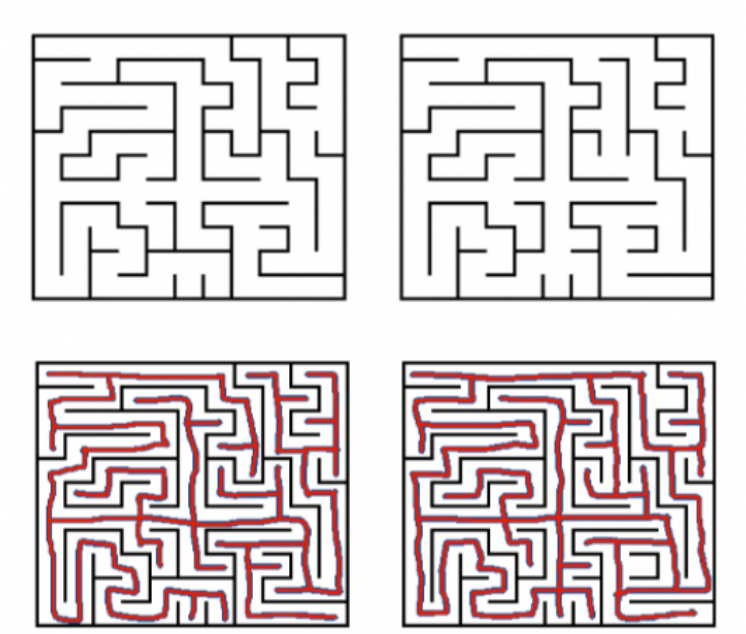


FIGURE 3.4 – Géénration d'un labyritnhe 3*3 à l'aide de l'algortihme d'arbre binaire.

Dans le cas du labyrinthe imparfait à droite, on va bien qu'il existe une boucle à l'intérieur du graphe.

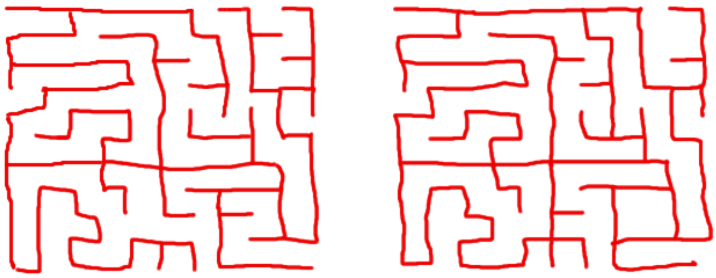


FIGURE 3.5 – Modélisation des labyritnhes sous forme de graphes.

De cela, nous arrivons à la conclusion que la génération d'un labyrinthe parfait aléatoire consiste à générer un arbre couvrant aléatoire qui relie toutes les cellules du labyrinthe. Les différentes procédures et approches utilisées pour la génération d'arbres couvrant forment les bases des différents algorithmes de génération de labyrinthes.

3.4 État de l'art : études des solutions existantes

Dans cette section, nous passons en revue certains des algorithmes les plus populaires utilisés pour la génération de labyrinthes parfaits. L'ensemble des algorithmes présentés dans cette section suivent l'approche "sculpture de passages" pour générer un labyrinthe. Pour chaque algorithme, on démarre à chaque fois d'une grille remplie de murs, puis l'algorithme se chargera de creuser un chemin parmis ces murs afin de construire son labyrinthe.

3.4.1 Algorithme de Kruskal

Le générateur de labyrinthe Kruskal est une version aléatoire de l'algorithme de Kruskal. C'est un algorithme qui permet de produire un arbre couvrant pour un graphe (un labyrinthe parfait). L'algorithme de Kruskal randomnisé pour la génération de labyrinthe fonctionne de la manière suivante :

1. On démarre d'un labyrinthe sous de grille complètement remplie de murs. On regroupe tous les murs séparant les cellules du labyrinthe dans un ensemble.
2. On choisit un mur à partir de cet ensemble de manière aléatoire et on le supprime du labyrinthe.
3. On réitère l'opération jusqu'à ce qu'il n'y est plus aucune cellule isolée.

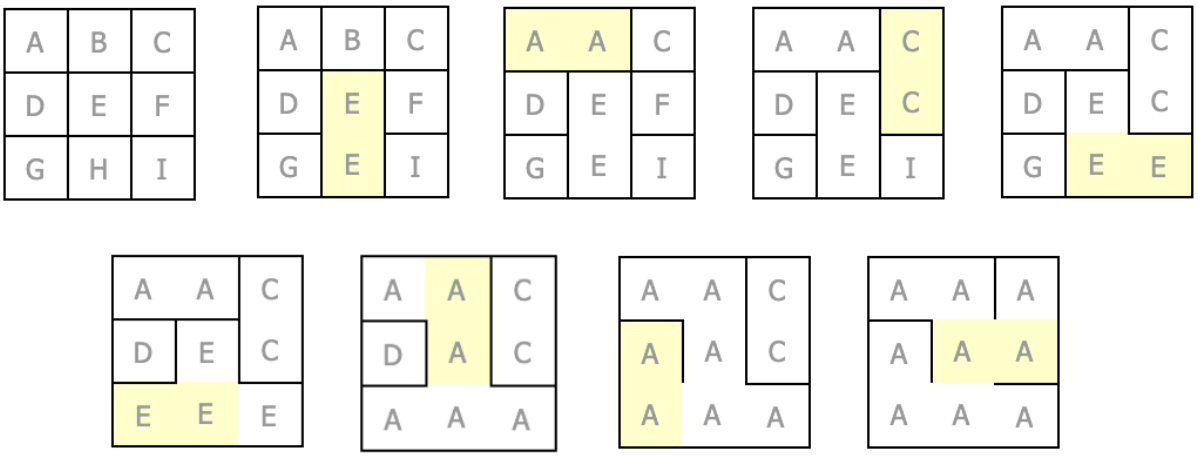


FIGURE 3.6 – Géénration d'un labyritnhe 3*3 à l'aide de l'algortihme de Kruskal.

3.4.2 Algorithme d'arbre binaire

L'algorithme de génération de labyrinthes "Arbre Binaire" est l'un des algorithmes les plus simples utilisé pour la génération de labyrinthe parfaits. Une des spécificités de l'algorithme est qu'il n'a pas besoin de sauvegarder en mémoire l'état des cellules du labyrinthe. Il peut construire un labyrinthe entier en regardant une seule cellule à la fois. Son principe de fonctionnement est très simple :

1. Pour chaque cellule au sein du labyrinthe, exécuter les étapes deux et trois.
2. Si ils existent, récupérer les voisins se situant au nord et à l'ouest de la cellule.
3. On choisit un mur à partir de cet ensemble de manière aléatoire et on le supprime du labyrinthe.
4. Sélectionner un des voisins de manière aléatoire, et perforer le mur reliant la cellule et le voisin choisit.

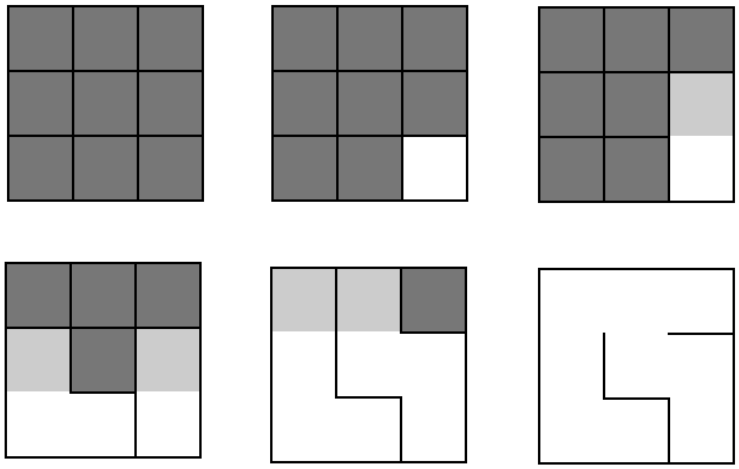


FIGURE 3.7 – Géénration d'un labyritnhe 3*3 à l'aide de l'algortihme d'arbre binaire.

3.4.3 Algorithme de Prim

Dans l'algorithme de Kruskal, on effectuait une sélection aléatoire parmis les murs du labyrinthe qu'on supprimait. L'algorithme de Prim aborde le problème de génération de labyrinthe sous un angle différent, ici, on démarre à partir de n'importe quelle cellule puis à partir de cette cellule l'algorithme va se charger de développer un chemin jusqu'à ce qu'on obtienne un labyrinthe parfait. L'algorithme de Prim randomnisé pour la génération de labyrinthe suit les étapes suivantes :

1. On choisit une cellule de départ de manière aléatoire qu'on ajoute à l'ensemble V.
2. On choisit une cellule aléatoire C de l'ensemble V dont au moins un des voisins n'appartient pas encore à l'ensemble V.
3. On choisit un voisin aléatoir de la cellule C et on perfore le mur reliant ces deux cellules. On ajoute le voisin choisit à l'ensemble V.
4. On réitère les étapes deux et trois jusqu'à ce que l'ensemble V contienne toutes les cellules du labyrinthe.

Sur la figure ci-dessous on est illustrée la génération d'un labyrinthe à l'aide de l'algorithme de Prim étape par étape. Les cellules appartenant à l'ensemble V (au labyrinthe) sont marquées en blanc, leurs cellules voisines en rose et en jaune on marque à chaque quelle cellule a été chosiit de manière aléatoire.

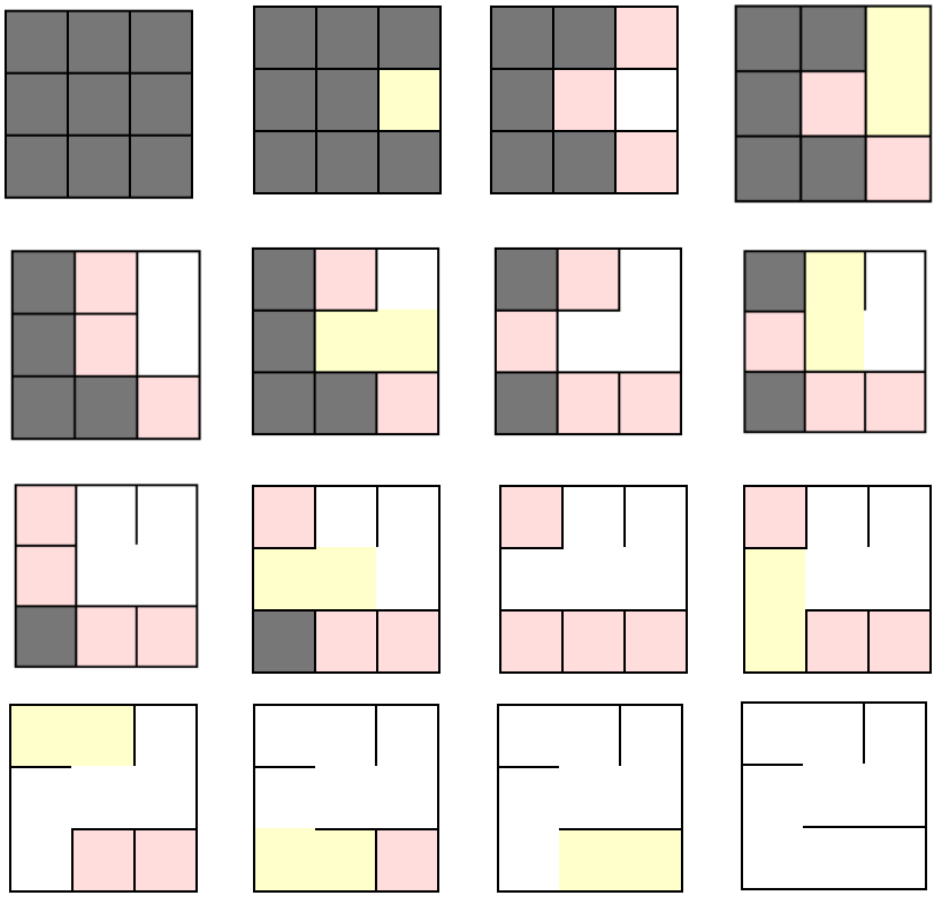


FIGURE 3.8 – Géénration d'un labyritnhe 3*3 à l'aide de l'algortihme de Prim.

3.4.4 Algorithme de Backtracking

Le Backtracking est une approche et une famille d'algorithmes utilisés pour résoudre des problèmes algorithmiques. Un algorithme de Backtracking (en français retour en arrière) est un algorithme récursif qui tente de résoudre un problème donné en testant tous les chemins possibles vers une solution jusqu'à ce qu'une solution soit trouvée. Chaque fois qu'un chemin est testé, si aucune solution n'est trouvée, l'algorithme revient en arrière pour tester un autre chemin possible. L'algorithme réitère cette logique jusqu'à ce qu'une solution soit trouvée ou que tous les chemins aient été Le Backtracking est donc un parcours en profondeur sur l'arbre de décision du problème, avec possibilité de revenir en arrière. En ce qui concerne l'algorithme de Backtracking pour la génération de labyrinthe parfait, on verra dans la section suivante que ça sera l'algorithme choisit pour générer les labyrinthes dans lequels va évoluer la Micromouse. On reviendra en détails sur cet algorithme et sa logique de fonctionnement dans la section **3.6.1.**

3.5 Comparaison des algorithmes de génération de labyrinthes

3.5.1 Introduction à la comparaison de labyrinthes

Maintenant qu'on a présenté un ensemble d'algorithmes utilisés pour la génération de labyrinthes parfaits, il faut se décider sur l'algorithme à implémenter afin de générer les labyrinthes dans lequels va évoluer notre Micromouse. Dans cette section, nous présentons les critères et métriques sur lequels on s'est basé afin de comparer les différents algorithmes ainsi que les labyrinthes qu'ils génèrent. Cette section se basent en grande partie sur les recherches et analyses menées et présentées dans le livre "Maze pour Programmers"[?].
Dans ce livre, les différents algorithmes de génération de labyrinthes sont comparés à l'aide des statistiques. Chaque algorithme est exécuté un nombre précis de fois pour générer des labyrinthes de taille fixe. Il suffit alors de choisir une propriété précise et d'analyser les labyrinthes résultants.

3.5.2 Critères de comparaison de labyrinthes

Impasse

Une impasse est définie comme une cellule qui n'est liée qu'à un seul voisin, c'est à dire une cellule qui ne possède qu'un seul mur perforé, ces autres murs étant fermés. Un labyrinthe avec peu d'impasses aura tendance à contenir plus de routes secondaires, ce qui aura pour effet de rendre le labyrinthe plus difficile à résoudre.
D'un point de vu de la Micromouse, et dans le but de tester l'intelligence de cette dernière, on préférera un labyrinthe contenant peu d'impasses afin de pouvoir mettre à épreuve les capacités de décisions de cette dernière. Par exemple, dans un labyrinthe contenant un nombre important d'impasses, on pourrait programmer la Micromouse de sorte qu'elle se contente d'avancer sans réel objectif et qu'elle revienne sur ses pas dans la cas ou elle tombe sur impasse et sauvegarde en mémoire la position de cette dernière. Ce n'est clairement pas intéressant comme algorithme de navigation, ce qui nous conforte dans notre choix d'opter pour un algorithme qui génère peu d'impasses. La figure suivante montre le nombre moyen d'impasses générées au sein du labyrinthe par différents algorithmes.

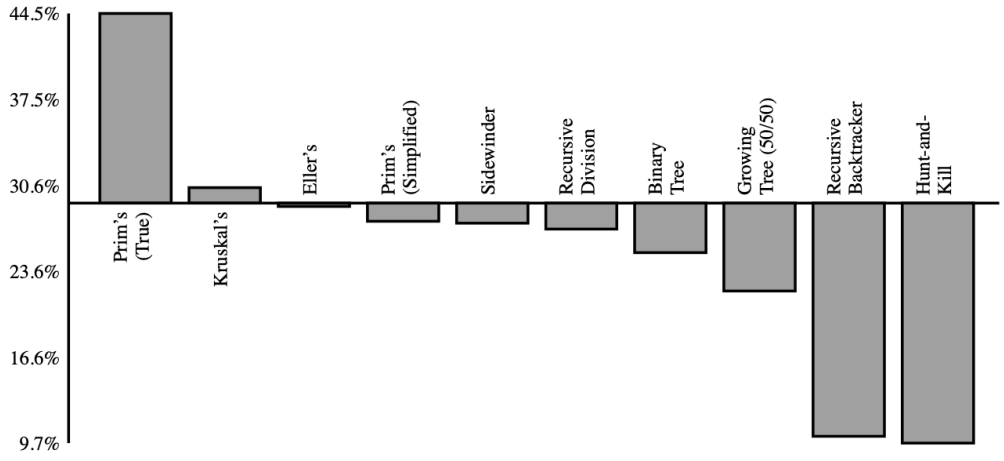


FIGURE 3.9 – Moyenne d'impasses engendrées par différents algorithmes de génération de labyrinthes

On constate que l'algorithme de Prim a tendance à produire des labyrinthes où environ 45% des cellules sont des impasses. En parallèle l'algorithme de Backtracking en génère un nombre beaucoup moins important.

Variation de directions

La variation de directions est une mesure de la fréquence à laquelle un passage change de direction. Il est pris comme le pourcentage de cellules avec un passage entrant sur un côté et sortant à gauche ou à droite. Dans un labyrinthe un faible variation de directions, la majorité des chemins seront des passages droits (horizontaux ou verticaux). Toujours dans l'optique d'utilisation du labyrinthe comme environnement pour la Micromouse, un labyrinthe à faible taux de variation n'est pas intéressant, en effet, l'intelligence de le Micromouse sera mise à l'épreuve lorsque cette dernière devra faire des décisions, l'objectif étant de choisir la direction assurant une solution optimale. Dans le cas de passages en ligne droite, la Micromouse ne fera qu'avancer tout droit en attendant de tomber sur un virage.

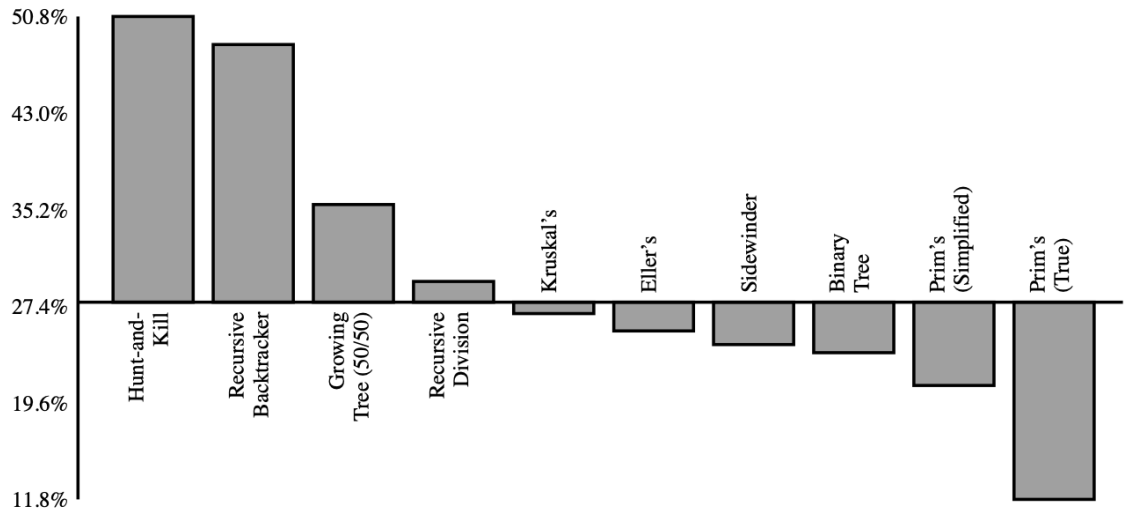


FIGURE 3.10 – Taux de variation de directions moyens pour différents algorithmes de génération de labyrinthes

On s'aperçoit que l'algorithme de Backtracking dispose d'un fréquence de variation de directions de quasiment 50%, tandis que l'algorithme de Kruskal, de Prim et d'arbre binaire ont des fréquences de variations plus petites.

3.5.3 Conclusion

3.6 Solution proposée et sa mise en œuvre

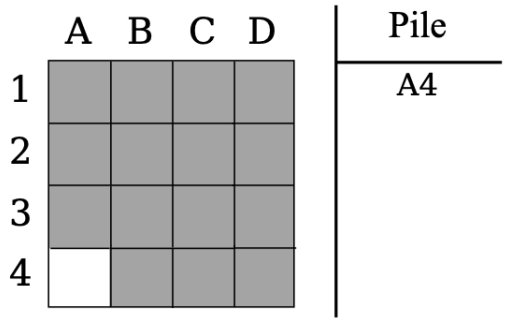
3.6.1 Algorithme de Backtracking pour la génération de labyrinthes parfaits

Les bases de l'algorithme Backtracking pour la génération de labyrinthes parfaits est qu'on démarre à partir d'une cellule prise au hasard de notre labyrinthe, et qu'on "creuse" un chemin dans une direction aléatoire (haut, bas, gauche, droite). La contrainte est qu'on ne peut jamais creuser vers une cellule qu'on a déjà visitée. À chaque itération, l'algorithme garde en mémoire les cellules visitées et les voisins des ces dernières. Dans le cas ou on arrive sur une cellules dont l'ensemble des voisins ont déjà été visités, on revient en arrière jusqu'à ce qu'on trouve une cellule à qui il reste des voisins non visités. On utilisera une pile comme structure de données afin de stocker les cellules visitées. Le pseudo code de l'algorithme est le suivant :

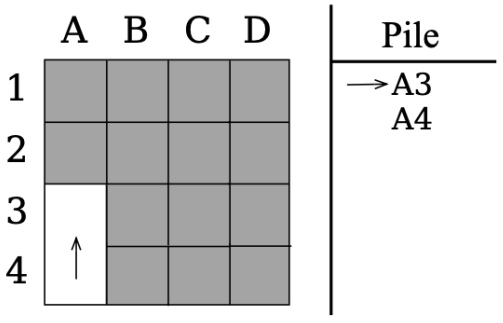
1. Choisir une cellule de départ aléatoire, la marquer comme visitée et l'empiler.
2. Tant que la pile n'est pas vide :
   (a) Dépiler une cellule de la pile et la marquer comme visitée.
   (b) Si la cellule actuelle possède des voisins qui n'ont pas encore été visités :
      i. Choisir au hasard un des voisins non visité.
      ii. Supprimer le mur reliant la cellule actuelle et le voisin choisi.
      iii. Empiler la cellule actuelle dans la pile et se déplacer vers le voisin choisi.

Pour une meilleure compréhension, nous allons illustrer et expliciter la génération d'un labyrinthe par l'algorithme de Backtracking étape par étape. Nous utiliserons une pile afin de garder une trace des cellules que nous avons visitées. On rappelle qu'un pile ne peut être manipulé que via ses opération "empiler" et "dépiler", ce qui assure un ordre précis avec lequel ses éléments sont consultés, ordre tout à fait adapté à l'algorithme de Backtracking dans lequel on souhaite ajouter des cellules au fur et à mesure de leur découverte et les dépiler dès qu'elles ne possèdent plus de voisins nos visités.

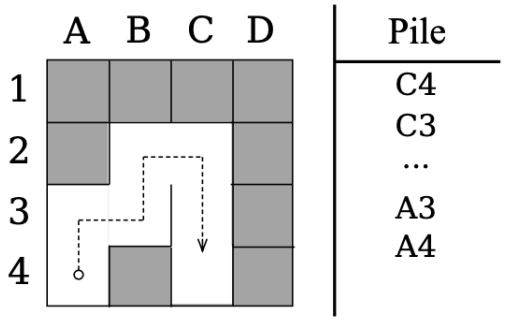
Au départ de l'algorithme, on choisit une cellule aléatoire du labyrinthe qu'on empile dans notre pile. Dans notre exemple on démarre avec la cellule A4. À chaque itération, la cellule au sommet de la pile est considérée comme étant la cellule actuelle.



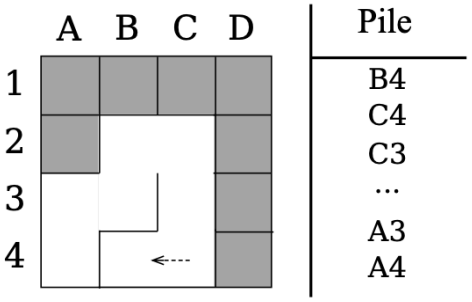
On choisit un voisin aléatoire de la cellule actuelle (dans notre exemple A3) et on supprime le mur qui sépare les deux cellules. On empile le voisin choisit dans notre pile, ce qui en fait la nouvelle cellule actuelle.



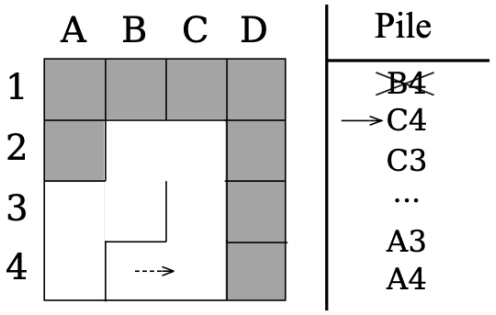
Ce processus itérative se poursuit, on explore aléatoirement les cellules voisines comme on peut le voir sur la figure. Chaque cellule visitée sera ajouter à notre pile.



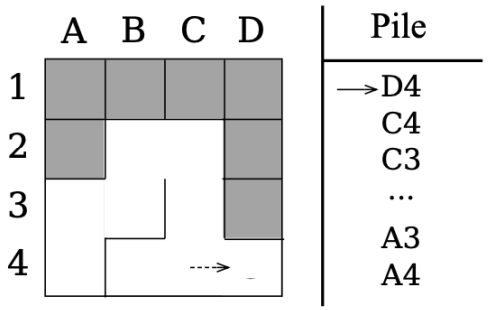
Notre processus aléatoire nous emmène vers la cellule B4 qui elle ne dispose pas de voisins nos visités.



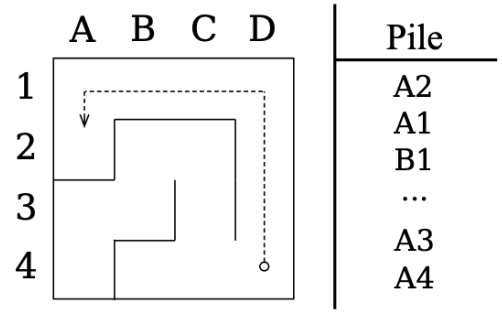
À cette étape, on dépile la cellule B4 de notre pile car elle ne possède plus de voisins non visités. Cette opération a pour effet de faire de la cellule précédente (cellule C4) la nouvelle cellule actuelle de l'algorithme.



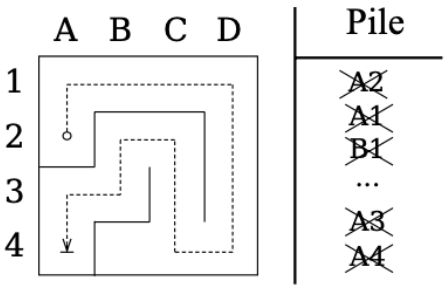
La cellule C4 dispose encore d'un voisin non visité (D4). On poursuit alors notre exploration aléatoire en choisissant ce voisin.



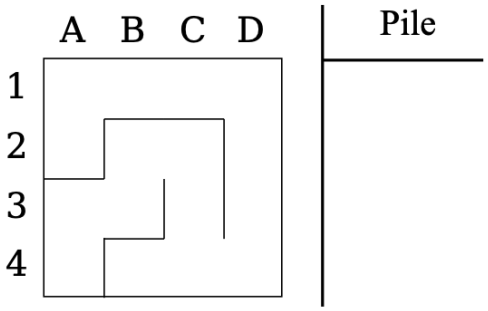
Le processus se poursuit de cette manière, en revenant en arrière à chaque fois qu'on tombe sur une cellule ne possédant pas de voisins visitable jusqu'à ce que chaque cellule du labyrinthe ait été visitée.



La dernière cellule à visiter sera toujours une cellule n'ayant pas de voisins visitables. Nous revenons alors sur nos pas, en dépilant les cellules de la pile, à la recherche d'un voisin nos visité. Cependant dans notre cas, l'ensemble des cellules du labyrinthe ont été visitées, on dépilera alors les cellules jusqu'à revenir à notre cellule de départ (la cellule A4) qu'on supprimera également de la pile, ce qui nous laisse avec une pile vide.



Une pile vide est le signal de terminaison de l'algorithme. On se retrouve alors avec le résultat final qui est un labyrinthe parfait.



Sur les figures ci-dessous, on montre différents labyrinthes parfaits générés par notre algorithme de Backtracking une fois ce dernier implémenté.

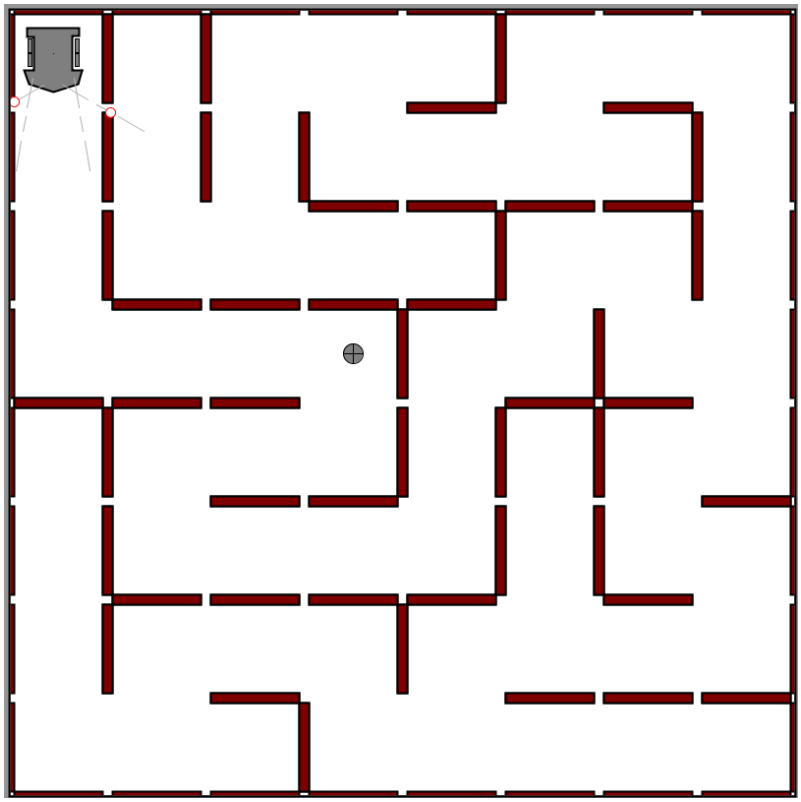


FIGURE 3.11 – Génération d'un labyrinthe parfait 8*8 dans notre simulation.

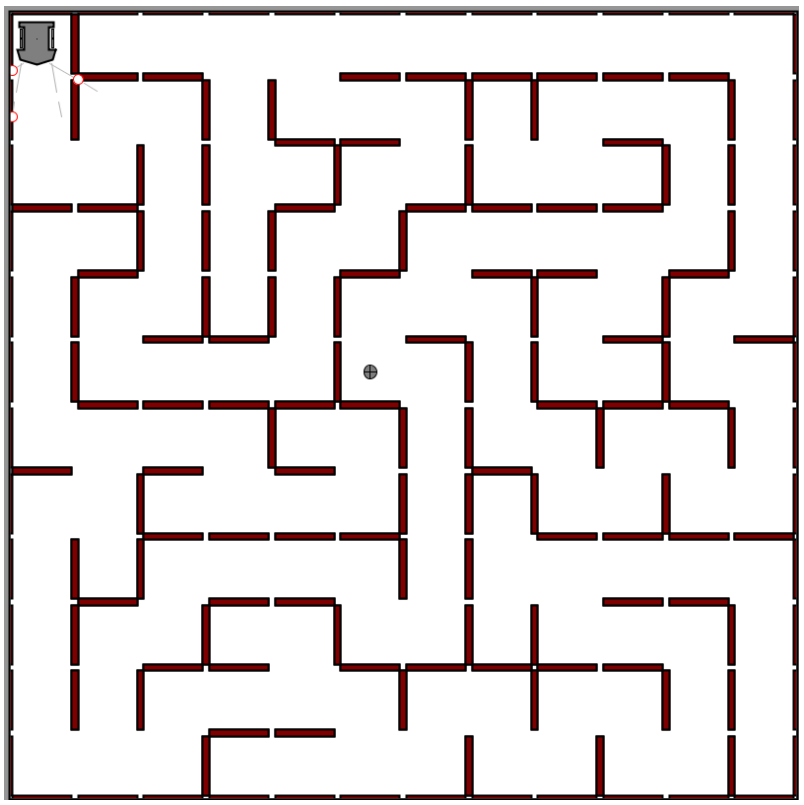


FIGURE 3.12 – Génération d'un labyrinthe parfait 12*12 dans notre simulation.

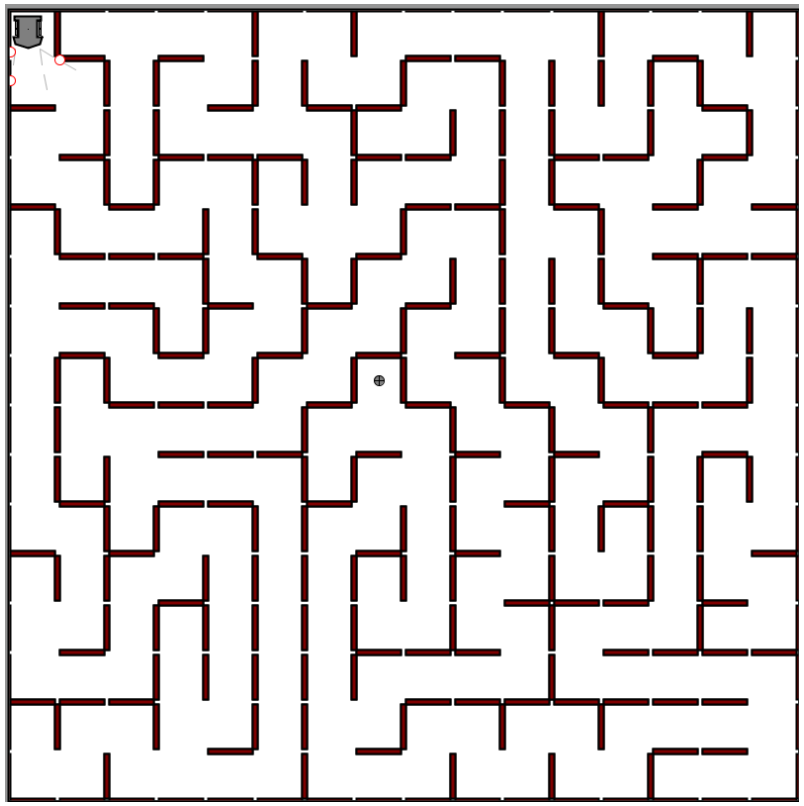


FIGURE 3.13 – Génération d'un labyrinthe parfait 16*16 dans notre simulation.

3.6.2 Génération de labyrinthes imparfaits

Une des propriétées de base d'un labyrinthe parfait et qu'il existe un seul et unique chemin reliant deux cellules du labyrinthe. Cette propriété n'est pas respecté dans un labyrinthe imparfait, tel qu'il peut exister des boucles ou plusieurs chemins reliant deux cellules. Un labyrinthe imparfait peut être intéressant dans le cas ou l'on souhaite avoir plusieurs parcours possible entre deux cellules du labyrinthe afin de tester l'intelligence de notre Micromouse. Par exemple, dans le cas ou notre Micromouse démarre d'une cellule de départ A, qu'elle a pour objectif d'atteindre la cellule B, et qu'entre ces deux cellules il existe plus d'un chemin possible, on souhaiterais qu'après un certain nombre de parcours la Micromouse puisse distinguer entre les deux chemins et choisir le chemin optimal (le chemin le plus court).

On génère un labyrinthe imparfait de la manière suivante :

1. On commence d'abord par générer un labyrinthe parfait.
2. Initialiser la position de la cellule départ (cellule sur laquelle démarre la Micromouse) et la cellule objectif (cellule que doit atteindre la Micromouse)
3. On divise les cellules du labyrinthe en deux ensemble : l'ensemble "départ" et l'ensemble "objectif". La classification se fait sur le critère de proximité des cellules, les cellules les plus proches de la cellule départ seront mises dans l'ensemble départ et les cellules les plus proches de la cellule objectif seront mises dans l'ensemble objectif.
4. Chercher un mur du labyrinthe qui sépare les deux régions (ensemble) et le supprimer.

On peut voir sur la figure ci-dessous le processus de génération d'un labyrinthe imparfait :

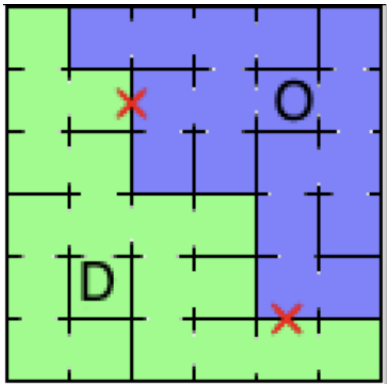


FIGURE 3.14 – Génération d'un labyrinthe 6*6 imparfait.

Sur la figure ci-dessous est illustrée un labyrinthe imparfait généré par notre algorithme au sein de notre simulation :

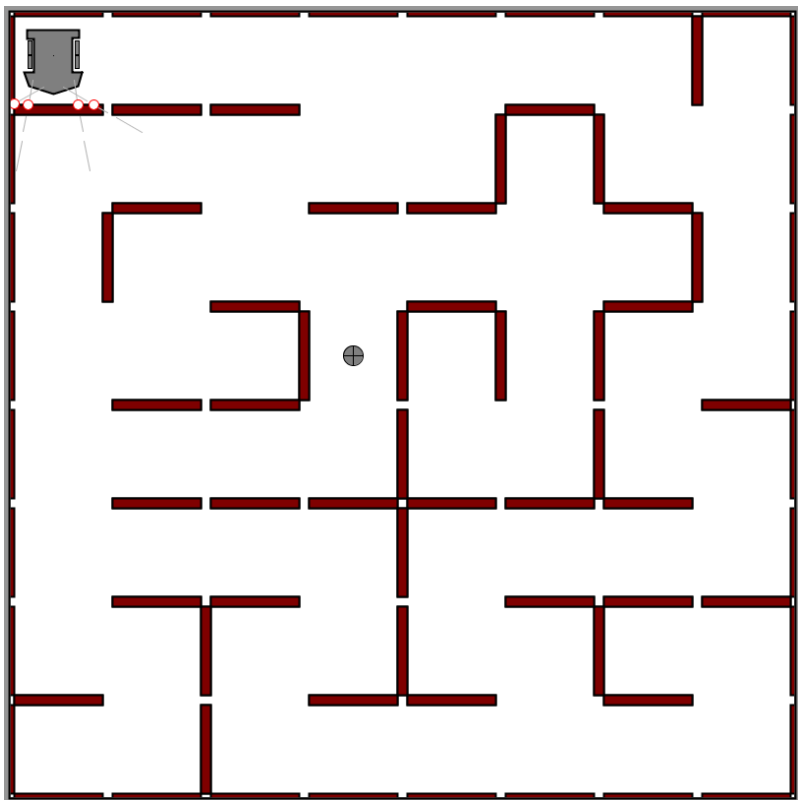


FIGURE 3.15 – Génération d'un labyrinthe 8*8 imparfait dans notre simulation.

Chapitre 4

Mappage et navigation

Chapeau du chapitre

4.1 Analyse de la problématique du mappage et navigation du véhicule

Chapeau

4.2 État de l'art : études des solutions existantes

Chapeau

4.3 Solution proposée et sa mise en œuvre

Chapeau

4.4 Tests et certifications de la solution

Chapeau

Chapitre 5

Contrôle du véhicule

Chapeau du chapitre

5.1 Analyse de la problématique du contrôle du véhicule

Chapeau

5.2 État de l'art : études des solutions existantes

Chapeau

5.3 Solution proposée et sa mise en œuvre

Chapeau

5.4 Tests et certifications de la solution

Chapeau

Chapitre 6

Communication

Chapeau du chapitre [10pt]article [usenames]color amssymb amsmath [utf8]inputenc

6.1 Analyse de la problématique de la communication

Puisque nous avons présenté lors des parties précédentes les différentes composantes du projet, il s'agit maintenant de les faire communiquer entre eux. Nous nous interrogerons donc, au cours de cette partie, sur un moyen de communication inter-process entre la simulation en Java/Processing et le brain en C en local (sur une même machine) puis à distance (entre la micromouse et l'IHM).

6.2 État de l'art : études des solutions existantes

Pour faire communiquer plusieurs processus, de nombreux moyens existent. Cette partie a pour but d'étudier ces différentes alternatives afin d'en tirer les plus avantageux en fonction de nos besoins.

6.2.1 Communication par sockets

Un socket permet une abstraction de la couche logicielle d'un programme se rapportant au réseau. Il s'appuie sur les modules du système d'exploitation afin de faire communiquer deux processus selon deux modes de connexions :

- le mode connecté s'appuyant sur le protocole de la couche transport TCP (Transmission Control Protocol) qui établit une connexion durable entre les deux interlocuteurs. Ce mode de communication est reconnu comme fiable puisqu'il s'appuie sur un mécanisme d'acquittement et de contrôle CRC (Cyclic Redundancy Control) pour chaque paquet IP envoyé. Une erreur détectée par un de ces deux mécanisme déclenche le renvoi des données alors perdues ou corrompues ;
- le mode non-connecté utilisant le protocole de la couche transport UDP (User Datagram Protocol) qui présente l'avantage d'être plus léger puisqu'il s'affranchit des mécanismes d'acquittement et de contrôle cités précédemment.

Ainsi, si dans notre projet, sockets il y a, c'est le mode non-connecté qui sera privilégié. En effet, les données échangées étant soumises à des contraintes de temps réels, lorsqu'un paquet IP est perdu, il n'est pas utile de le renvoyer car les données qu'il porte sont alors inactuelles et donc inutilisables.

Dans le langage C, ces deux types sockets sont implémentés dans la bibliothèque `<sys/socket.h>` alors qu'en Java/Processing, c'est la bibliothèque `hypermedia.net.*` qui est utilisée pour la création de socket UDP.

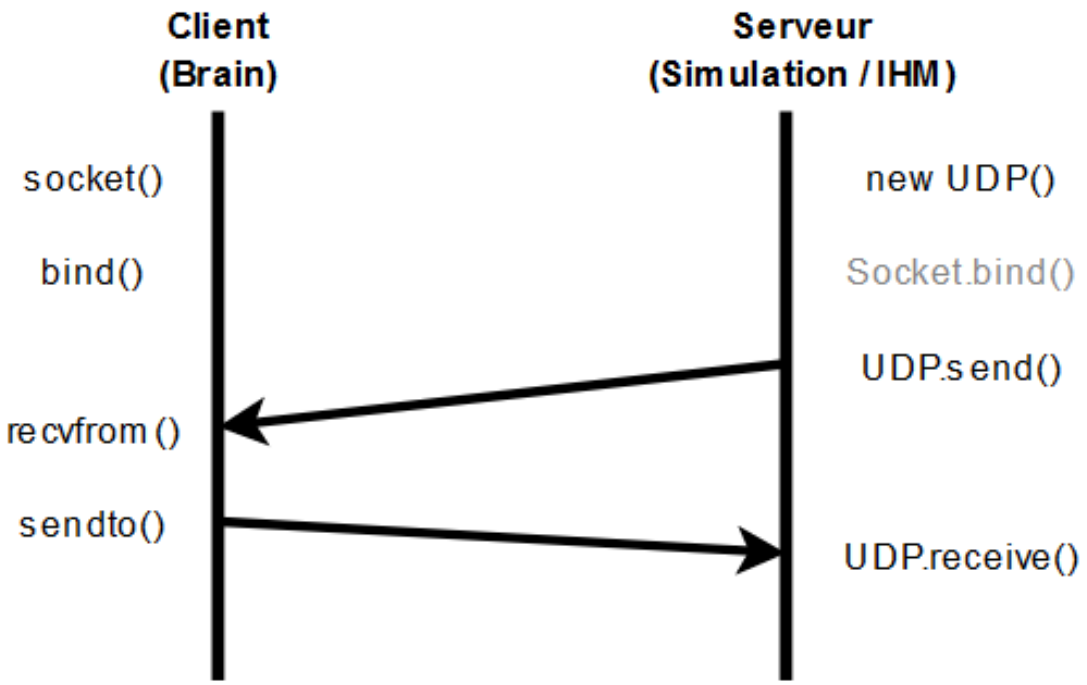


FIGURE 6.1 – Diagramme décrivant le fonctionnement des sockets UDP

Par ailleurs, la communication par sockets adopte une architecture clients/serveur. Il est donc possible d'envisager un dispositif composé de plusieurs micromouses en utilisant cette solution de communication ; la partie simulation représentant le serveur et les brains, les clients.

6.3 Solution proposée et sa mise en œuvre

Au vu des choix matériels, nous avons opté pour les canaux nommés afin de faire communiquer les différentes composantes du projet. En effet, notre micromouse ne possédant pas de carte réseau, c'est par Bluetooth que se fera le communication entre les deux programmes. Ainsi, bien qu'elle semblait être la plus adaptée, la solution employant les sockets se retoruve alors inutilisable dans le contexte décrit.

6.3.1 Qu'est-ce qu'un canal nommé (named pipe) ?

6.4 Tests et certifications de la solution

Chapeau