Rapport de Stage

Période du 25 Février au 5 Mars 2019

MATON Hugo – REBRAY Malik

Réalisation d'un logiciel de pathfinding dans un Labyrinthe

I – Spécification de la demande

À partir d'une interface graphique réalisée précédemment, il nous a été demandé de concevoir la solution logicielle permettant les fonctionnalités suivantes :

> Dans un premier temps le déplacement automatique d'une entité (robot) dans un labyrinthe, de son entrée à sa sortie.
>
> Et dans un second temps, l'implémentation d'un algorithme de calcul de chemin le plus court (pathfinding) dans ce labyrinthe.

Ce programme est destiné aux élèves de Master de l'UBS de Lorient, l'interface graphique en tant qu'outil pour travaux pratiques, et les fonctionnalités demandées en tant que référence pour la correction.

Les fonctionnalités attendues étaient les suivantes :

- Chargement/sauvegarde du labyrinthe (et de la position actuelle du robot) dans un fichier.

- Notion de difficulté de terrain dans le labyrinthe.

- Contrôleur souris pour le déplacement manuel du robot dans le labyrinthe.

- Déplacement automatique du robot dans un labyrinthe.

- Organisation du code.

- Automatisation des tâches de compilation à l'aide d'un fichier makefile.

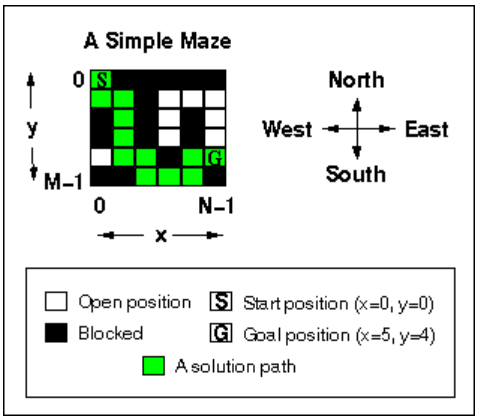
II – Étude des solutions

Afin de maintenir des standards de qualité au niveau de l'optimisation, il s'agissait de limiter la complexité des algorithmes choisis ou réalisés.

Pour des standards de sécurité, sous les conseils de notre tuteur, nous avons optés pour la création d'un fichier makefile pour automatiser la compilation, tout en sécurisant le programme contre de potentielles erreurs.

La première étape était développer l'algorithme de déplacement automatique du robot de la case de départ à celle d'arrivée.

Pour y parvenir, nous sommes allés nous documenter sur différents sites. Notre attention s'est portée sur la solution proposée par l'Université de Boston, s'apparentant à ce schéma :



Il s'agit ici pour le robot de vérifier les cases environnantes pour se diriger, en suivant le cycle Nord -> Est -> Sud -> Ouest.

Cette solution permet au robot de trouver son chemin de l'entrée à la sortie, cependant, dans le cadre d'un labyrinthe comportant plusieurs chemins, il n'emprunte pas nécessairement le plus court.

Pour pallier ce problème, nous avons opté pour la solution qui nous semblait à la fois la plus adaptée, mais également la plus abordable : l'algorithme de Dijkstra.

En effet, ce dernier permet de calculer le plus court chemin entre les sommets d'un graphe non orienté.

Voici le principe en pseudo-code :

```
Entrées : G = (S, A) un graphe avec une pondération positive poids des arcs, s_deb un sommet de S

P := ∅
d[a] := +∞ pour chaque sommet a
d[s_deb] = 0
Tant qu'il existe un sommet hors de P
    Choisir un sommet a hors de P de plus petite distance d[a]
    Mettre a dans P
    Pour chaque sommet b hors de  P voisin de a
        d[b] = min(d[b], d[a] + poids(a, b))
    Fin Pour
Fin Tant Que
```

III - Mise en place de l'environnement de travail

Pour piloter le projet, nous avons créé un espace Trello, regroupant les différentes builds (versions) au fil de notre progrès, les tâches à réaliser ainsi que la documentation nécessaire.

Afin de développer les solutions choisies, il fallait ensuite mettre en place l'environnement adapté.

Dans l'anticipation du déploiement de la solution, nous avons installé et configuré la VMbox d'Oracle (machine virtuelle) en paramétrant une image de Debian 9.
Il s'agissait de mettre en place et de se familiariser avec le système Linux, puis d'installer la bibliothèque *graphics.h*, avec laquelle l'interface a été réalisée.

IV – Développement et débug

1 – Algorithme récursif de calcul du chemin

D'abord, nous nous sommes dirigés vers la réalisation d'une fonction récursive, permettant au robot de vérifier successivement si les cases au Nord, Sud, Est et Ouest de sa position sont :

- Dans le labyrinthe
- Un mur
- Un chemin marqué comme une impasse auparavant

Dans le cas où un chemin libre mène à une impasse, la fonctionnalité récursive remonte la pile d'appel de fonction (backtracking) jusqu'à la dernière intersection, tout en marquant les cases concernées comme menant à une impasse.

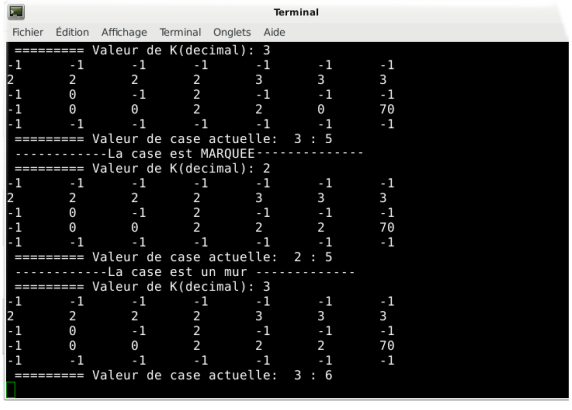
Le labyrinthe est représenté par un fichier texte, où les éléments sont traduits comme ceci :

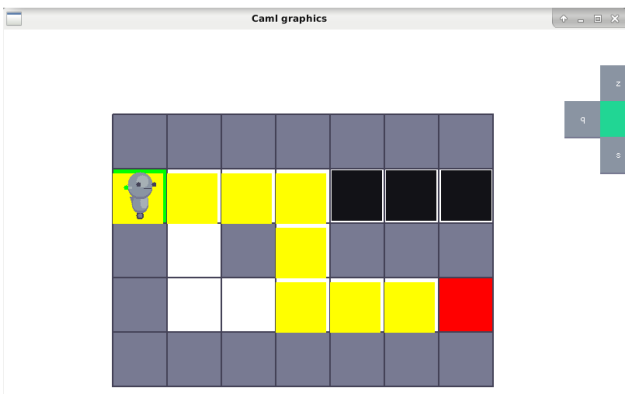
	Fichier	Programme
Mur	*	-1
Case libre (non visitée)	0	0
Chemin parcouru	2	2
Chemin menant à une impasse	3	3
Fin	F	70

Étant donné que le programme lit un fichier texte, il est nécessaire de traduire les caractères selon leur correspondance dans la table ASCII.

Afin de visualiser ce chemin, nous l'avons mis en évidence avec les outils de la libraire graphique, en coloriant les cases parcourues en jaune, ainsi que les celles menant à une impasse en noir

Voici un extrait de phase de test montrant la démarche :





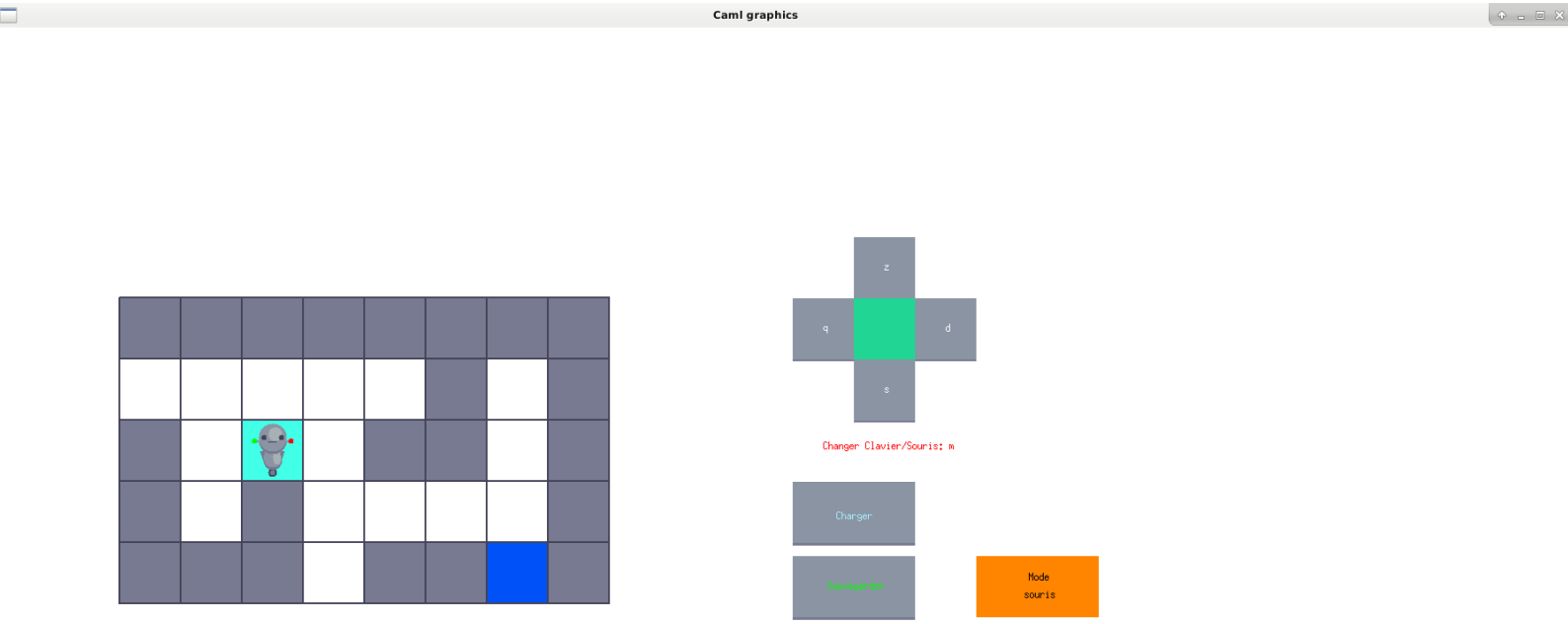
Pendant l'élaboration de cette fonction, nous avons rencontré une erreur de segmentation empêchant le progrès. Après schématisation sur papier et l'utilisation de gdb, nous avons déterminé la cause : la condition de sortie n'étant jamais atteinte, le programme tournait en boucle.

Cela était dû au marquage des cases menant à une impasse, qui ne fonctionnait pas à ce moment.

La dernière étape pour cette version était de faire évoluer l'interface graphique afin d'y rajouter un bouton « charger » et un bouton « sauvegarder ».

Pour ce faire, il fallait également ajouter une fonctionnalité pour rendre l'intégralité des boutons de l'interface cliquables.
Il s'agissait de récupérer la position de la souris pour vérifier que cette dernière corresponde bien à la zone définie des boutons, à l'aide d'une structure fournie par la libraire *graphics.h*.

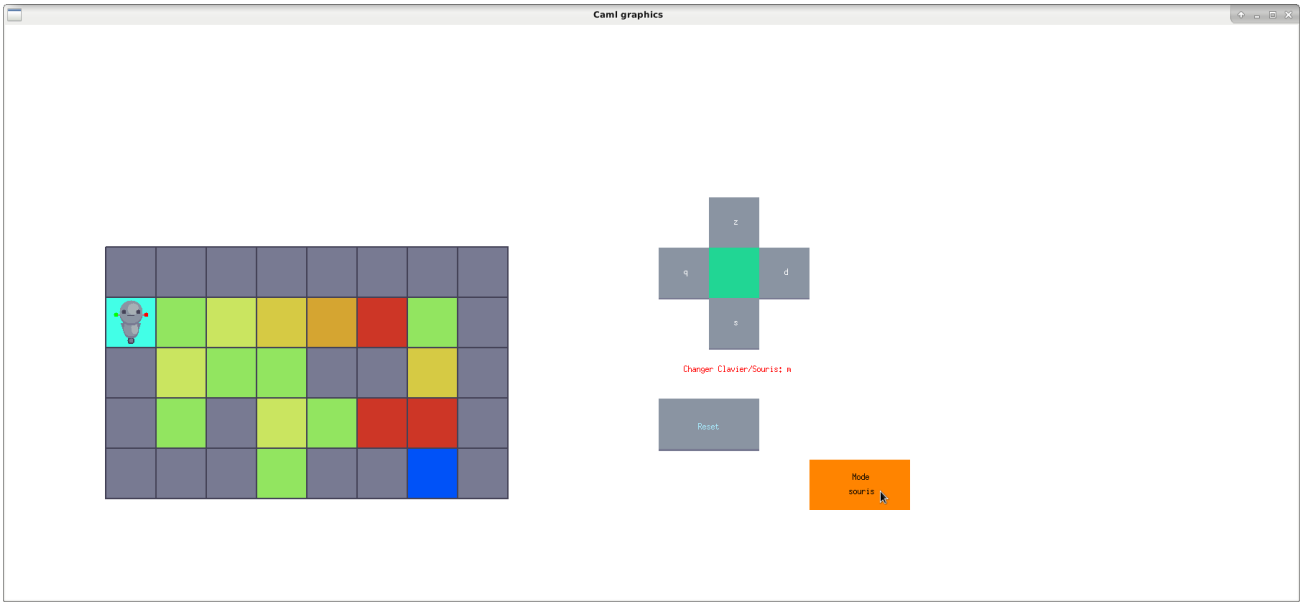
Voici l'interface après ajout des boutons en question, avec un élément indiquant le mode en cours (clavier ou souris) :



2 – Algorithme de Dijkstra

Une fois notre algorithme récursif fonctionnel, nous avons fait valider cette étape en réunion, puis nous avons proposé l'implémentation de l'algorithme de Dijkstra afin de déterminer le chemin le plus court dans le labyrinthe.

L'approche du calcul du chemin étant différente, il n'était plus pertinent d'avoir les fonctions « charger » et « sauvegarder » implémentées précédemment. Nous avons donc supprimé entièrement le bouton « sauvegarder », et avons remplacé la fonction de chargement en une simple réinitialisation du labyrinthe. Nous avons également ajouté la notion de difficulté de terrain (n'ayant pas d'intérêt dans la version récursive) et une coloration adéquate des cases concernées :



Pour implémenter cet algorithme, il nous a fallu l'adapter au projet. En effet, le labyrinthe devant être considéré comme un graphe, il fallait trouver un moyen de le traduire de manière à ce qu'il soit pris en compte dans l'algorithme, à savoir sous forme de matrice d'adjacence.

Chaque sommet du graphe correspond donc à une case du labyrinthe.

C'est sur ce principe que nous avons élaboré une fonction permettant le remplissage de la matrice d'adjacence par rapport aux cases du labyrinthe, prenant également en compte la difficulté de terrain. (Voir code sur la page suivante)

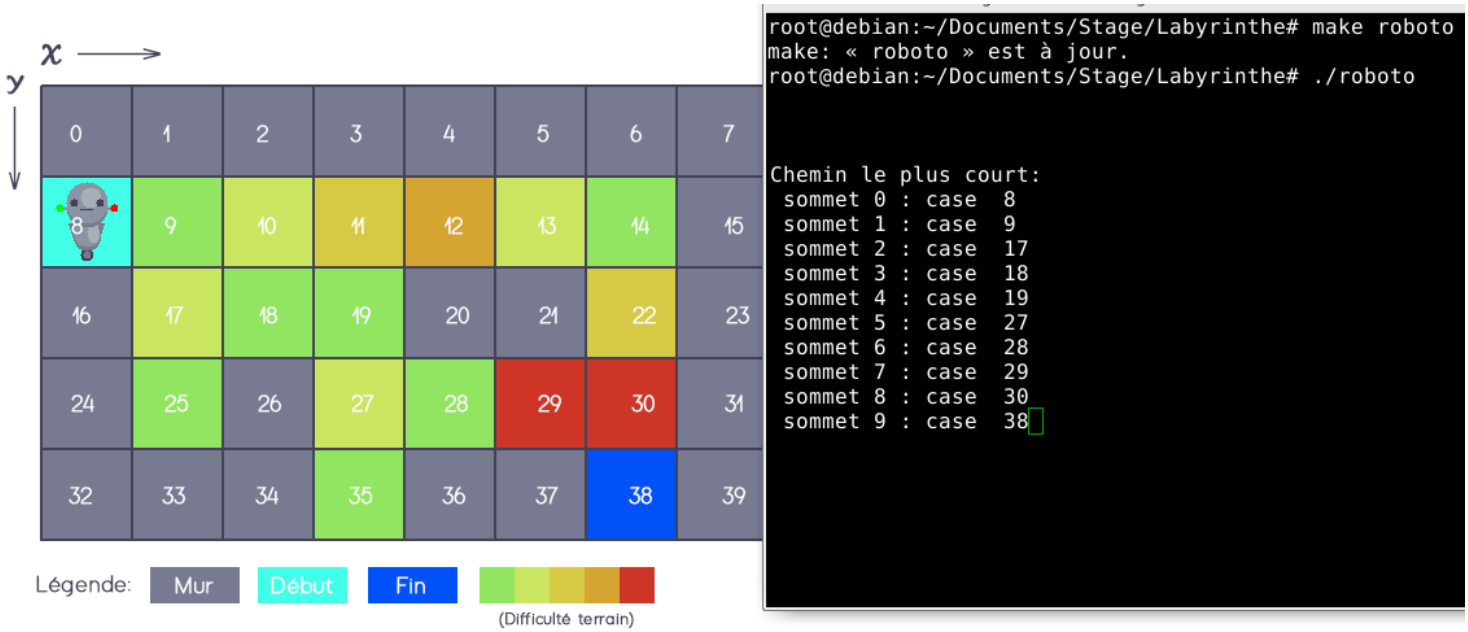
Fonction de remplissage de la matrice d'ajacence :

```
void remplissageMatriceAdj(int nbL, int nbC, int *Laby, int tailleAdj, int *Adjacence)
{
    int i;
    int j;

    for(i = 0; i<nbL; i++)
        {
            for (j = 0; j<nbC; j++)
            {
                if(Laby[i*nbC+j] != -1)
                {
                    /* SI la position du dessus n'est pas un mur ET SI elle est dans le labyrinthe */
                    if(Laby[i*nbC+j-nbC] != -1
                        && ((i*nbC+j-nbC) >= 0))
                    {

                        if(Laby[i*nbC+j-nbC] == 'D' || Laby[i*nbC+j-nbC] == 'F')
                        {
                            Adjacence[((i*nbC+j)*tailleAdj)+(i*nbC+j)-nbC] = 1;/* en haut */
                        }
                        else
                        {
                            Adjacence[((i*nbC+j)*tailleAdj)+(i*nbC+j)-nbC] = Laby[i*nbC+j-nbC];/* en haut */
                        }
                    }

                    /* SI la position du dessous n'est pas un mur ET SI elle est dans le labyrinthe */
                    if(Laby[i*nbC+j+nbC] != -1
                        && ((i*nbC+j+nbC) <= (tailleAdj)-1))
                    {
                        if(Laby[i*nbC+j+nbC] == 'D' || Laby[i*nbC+j+nbC] == 'F')
                        {
                            Adjacence[((i*nbC+j)*tailleAdj)+(i*nbC+j)+nbC] = 1; /* en bas */
                        }
                        else
                        {
                            Adjacence[((i*nbC+j)*tailleAdj)+(i*nbC+j)+nbC] = Laby[i*nbC+j+nbC]; /* en bas */
                        }
                    }

                    /* SI la position à gauche n'est pas un mur ET SI elle est comprise dans les limites de la ligne */
                    if(Laby[i*nbC+j-1] != -1
                        && (((i*nbC+j-1) >= (i*nbC)) && ((i*nbC+j-1 <= ((i+1)*nbC-1)))))
                    {
                        if(Laby[i*nbC+j-1] == 'D'  || Laby[i*nbC+j-1] == 'F')
                        {
                            Adjacence[((i*nbC+j)*tailleAdj)+(i*nbC+j)-1] = 1; /* gauche */
                        }
                        else
                        {
                            Adjacence[((i*nbC+j)*tailleAdj)+(i*nbC+j)-1] = Laby[i*nbC+j-1]; /* gauche */
                        }
                    }

                    /* SI la position à droite n'est pas un mur ET SI elle est comprise dans les limites de la ligne */
                    if(Laby[i*nbC+j+1] != -1
                        && (((i*nbC+j+1) >= (i*nbC)) && ((i*nbC+j+1 <= ((i+1)*nbC-1)))))
                    {
                        if(Laby[i*nbC+j+1] == 'D' || Laby[i*nbC+j+1] == 'F')
                        {
                            Adjacence[((i*nbC+j)*tailleAdj)+(i*nbC+j)+1] = 1; /* droite */
                        }
                        else
                        {
                            Adjacence[((i*nbC+j)*tailleAdj)+(i*nbC+j)+1] = Laby[i*nbC+j+1]; /* droite */
                        }
                    }
                }
            }
        }

        /* ==== DEBUG ==== =================
        for (i = 0; i < tailleAdj; i++)
        {
            for(j = 0; j < tailleAdj; j++)
            {
                printf("%d ", Adjacence[i*tailleAdj+j]);
            }
            printf("\n");
        }
        ==================================*/
}
```

Une fois la matrice exploitable, l'algorithme de Dijkstra était utilisable.

En prenant un paramètre un sommet source ainsi qu'un sommet de destination, il crée un tableau de sommets parents. En parcourant celui-ci, à partir de la fin en remontant jusqu'à la source, on récupère les sommets du chemin le plus court (qui sont stockés dans un nouveau tableau).

En voici le résultat fonctionnel (avec légende et numérotation des cases ajoutées pour faciliter la lecture) :



Comme montré ci-dessus, le chemin le plus court est déterminé en fonction de la difficulté de terrain, nous avons opté pour un affichage clair du chemin dans la console plutôt que d'encombrer l'interface graphique.

V – Documentation technique et utilisateur

1 – Documentation technique

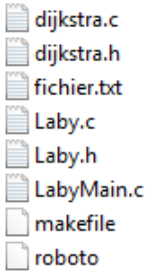
Afin de respecter des standards de qualité, nous avons commenté l'intégralité code au mieux pour apporter de la clarté aux solutions employées.

Voici un extrait du code, une partie de la zone de déclaration des variables, montrant certains commentaires aidant à la compréhension :

```
int x;
int y;
int I, J;
int i = 0;
int j = 0;
int taille = 70;
int nbL;
int nbC;

char touche;

int *Laby;
char debut = 'D',fin = 'F';

/* Les sosies contiennent les valeurs à l'origine de notre graphe (le labyrinthe se dessine de haut en bas et de gauche vers la droite) */
int sosieX;
int sosieY;

/*========== Variables pour Dijkstra ==========*/
int *Adjacence;
int tailleAdj; /* nombre de sommets dans le graphe */
int depart; /* sommet source */
int arrivee; /* sommet de destination */

int *leChemin; /* Liste d'adjacence */
int tailleDuChemin; /* Taille du chemin le plus court de la source à la destination */
/*============================================*/

int menu;/* Menu clavier(1) ou souris(0)  */

char *tabfichier;
FILE *fichier;
```

Toujours dans un souci de lisibilité et d'adaptabilité du code, nous avons opté pour une programmation modulaire, de manière à séparer les prototypes de fonctions, les fonctions elles-mêmes, et le programme principal les utilisant.

Voici le contenu du dossier du programme :



On peut discerner les fichiers contenant les prototypes (.h) de ceux contenant les fonctions (.c)

Le programme principal fait appel aux prototypes et aux fonctions à travers des « #include » en tête de fichier :

```
#include <stdio.h>
#include <stdlib.h>
#include "graphics.h"
#include "dijkstra.h"
#include "Laby.h"
```

2 – Documentation utilisateur

Pour faciliter la prise en main du logiciel, nous avons réalisé une documentation utilisateur pour les deux versions.

Voici une vue d'ensemble pour la version récursive :

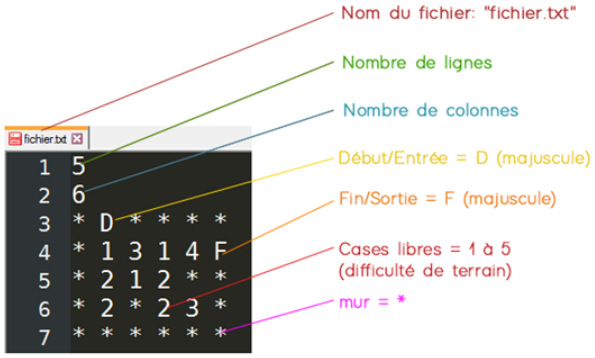
Manuel utilisateur

Outils nécessaires :

- Librairie graphique « graphics.h »
- Environnement Linux (avec compilateur gcc)

I - Création d'un Labyrinthe

Créer un fichier texte dans le répertoire principal :



II - Compilation et lancement du programme

- Ouvrir un terminal dans le répertoire principal
- Entrez la commande « make roboto »
- Entrez la commande « ./roboto »
-

Liste des commandes make disponibles :

make roboto : compilation

make cleanbin : supprime les binaires
make cleanexec : supprime l'exécutable
make cleanall : combine cleanbin et cleanexec
} Nettoyage des fichiers

III - Déplacement manuel du robot, rechargement du labyrinthe

ATTENTION :

Le déplacement est **sensible à la casse**, assurez-vous de ne pas être en majuscules

Mode clavier (par défaut) :

Déplacement : z (Haut) q (Gauche) s (Bas) d (Droite)

Chargement: p

Sauvegarder: v

Calcul du chemin vers la sortie: a

(Affichage décomposé, appuyer sur une touche pour progresser, ne déplace pas la position du robot)

Mode souris (appuyer sur « m » pour y basculer/cliquer sur le bouton « mode souris » pour en sortir) :

Cliquer sur les boutons correspondants à droite du Labyrinthe.

Et voici une vue d'ensemble pour la version Dijkstra:

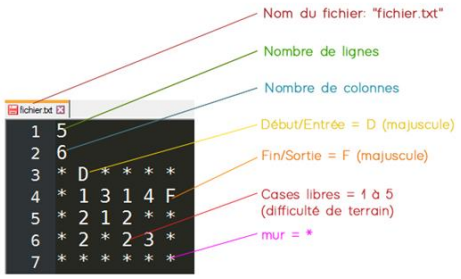
Manuel utilisateur

Outils nécessaires :

- Librairie graphique « graphics.h »
- Environnement Linux (avec compilateur gcc)

I - Création d'un Labyrinthe

Créer un fichier texte dans le répertoire principal :



II - Compilation et lancement du programme

- Ouvrir un terminal dans le répertoire principal
- Entrez la commande « make roboto »
- Entrez la commande « ./roboto »
-

Liste des commandes make disponibles :

make roboto : compilation

make cleanbin : supprime les binaires

make cleanexec : supprime l'exécutable

make cleanall : combine cleanbin et cleanexec

} Nettoyage des fichiers

III - Déplacement manuel du robot, rechargement du labyrinthe

ATTENTION :

Le déplacement est **sensible à la casse**, assurez vous de ne pas être en majuscules

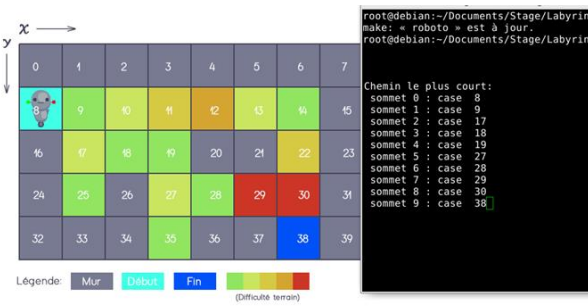
Mode clavier (par défaut) :

Déplacement : **z** (Haut) **q** (Gauche) **s** (Bas) **d** (Droite)

Reset: **p**

Mode souris (appuyer sur « m » pour y basculer/cliquer sur le boutton « mode souris » pour en sortir) :

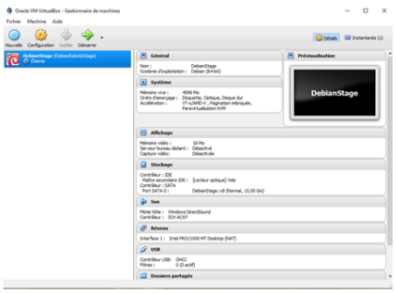
IV - Lecture des informations



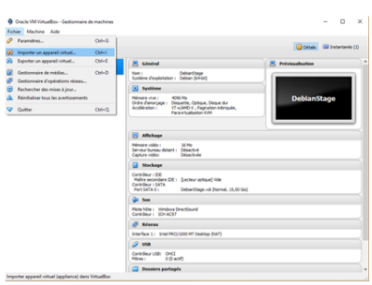
Enfin, notre environnement de travail étant virtualisé (sous VMbox), nous avons réalisé une procédure d'accompagnement de l'utilisateur pour son installation, dont voici une vue d'ensemble également :

Procédure d'installation de l'image Debian sous VM VirtualBox d'Oracle

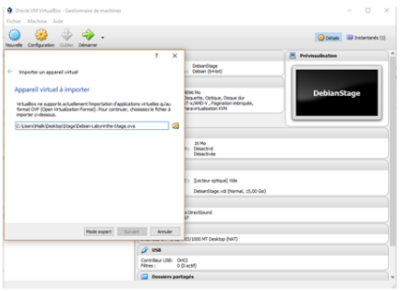
1. Lancer le Gestionnaire de machines (VirtualBox.exe)



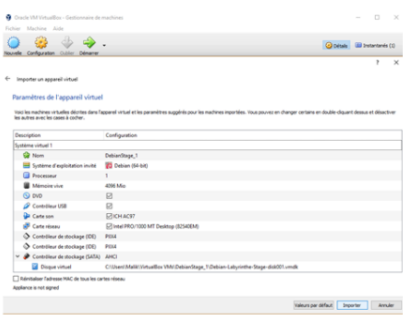
2. Sélectionner l'importation d'un appareil virtuel (Ctrl+I)



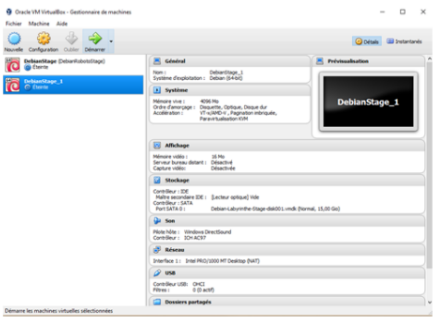
3. A l'aide du parcours de fichiers, sélectionnez l'image (.ova) qui vous a été fournie



4. Vérifier les paramètres de l'appareil virtuel et l'importer.



5. La nouvelle machine virtuelle est prête, vous pouvez la démarrer



Fin de document