RECHERCHE DE CHEMIN PAR CARTOGRAPHIE

DOSSIER PROJET ISN 2017-2018

LYCEE MARSEILLEVEYRE

LABARRE CHARLES

RISCH GUILLAUME

THIBAULT LUCAS

SOMMAIRE

INTRODUCTION

PRESENTATION DU PROJET

Dans ce dossier, nous allons traiter de notre projet portant sur la recherche de chemin à partir d'une carte. Nous avons programmé en C sur un compilateur en ligne “OnlineGBD” site très pratique pour l'utilisation de ce langage qui permet de coder sans passer par une interface de programmation en C ou par une version de linux.carnigan et richie

Nous avons choisi le langage C relativement aisé pour des novices car l’un des langages de base les plus efficace et dont la syntaxe est à la base de beaucoup de langages structurés actuels (c++, c #, python et java) dans le domaine de la

programmation et car il est la base de tous les langages. Sa manipulation permet d'acquérir des bonnes méthodes et outils de code.

HISTORIQUE

“Le C est un langage de programmation impératif et généraliste. Inventé au début des années 1970 pour réécrire UNIX. Il est devenu un des langages les plus utilisé. De nombreux langages plus modernes comme C++, C#, Java et PHP reprennent des aspects de C.” Wikipédia

Nous allons maintenant nous intéresser à notre projet concernant la recherche de chemin à partir d'une carte. L'objectif premier de ce projet est de proposer une route menant d'un point A choisi au préalable à un point B grâce à une carte intégrée directement dans la matrice de notre programme.

ANALYSE DU BESOIN

Le but de notre projet était de créer un programme semi autonome capable de tracer un chemin par lui-même d'un endroit cartographié

L'utilité de ce programme serait pour l'exploration d'endroit inaccessible ou dangereux pour l'homme comme une planète, une usine nucléaire des zones radioactives. On peut aussi imaginer l'utilisation de ce programme pour transporter des marchandises dans un hangar de façon autonome.

L’intérêt est vraiment de se passer d'aide extérieure ou de télécommande pour être le plus efficace possible.

RECHERCHE D'IDEES

L'idée de ce programme nous est venu grâce à notre TPE de première en science de l'ingénieur qui traitait de l'intelligence artificielle notre objectif était de retrouver le plus court chemin dans un labyrinthe codé dans un matrice grâce aux algorithmes de Dijkstra et A*. Ces algorithmes étant trop compliqués à implémenter à notre niveau, nous nous sommes inspirés de notre TPE pour rester dans la continuité de nos recherches. Nous voulions réaliser une entité capable d'analyser son environnement et d'en traduire une carte lui permettant de trouver par lui-même un chemin. Le processus de création d'une représentation informatique d'un plan étant trop complexe, nous avons décidé de donner au programme une carte déjà faite qu'il pourrait par la suite analyser et résoudre.

REPARTITION DES TACHES ET DEMARCHES COLLECTIVES

		Guillaume	Charles	Lucas
Algorigramme		X	X	
Recherche d'idées et analyse du besoin		X	X	
Programmation	Main	X	X	
	AffichageLabyrnthe			

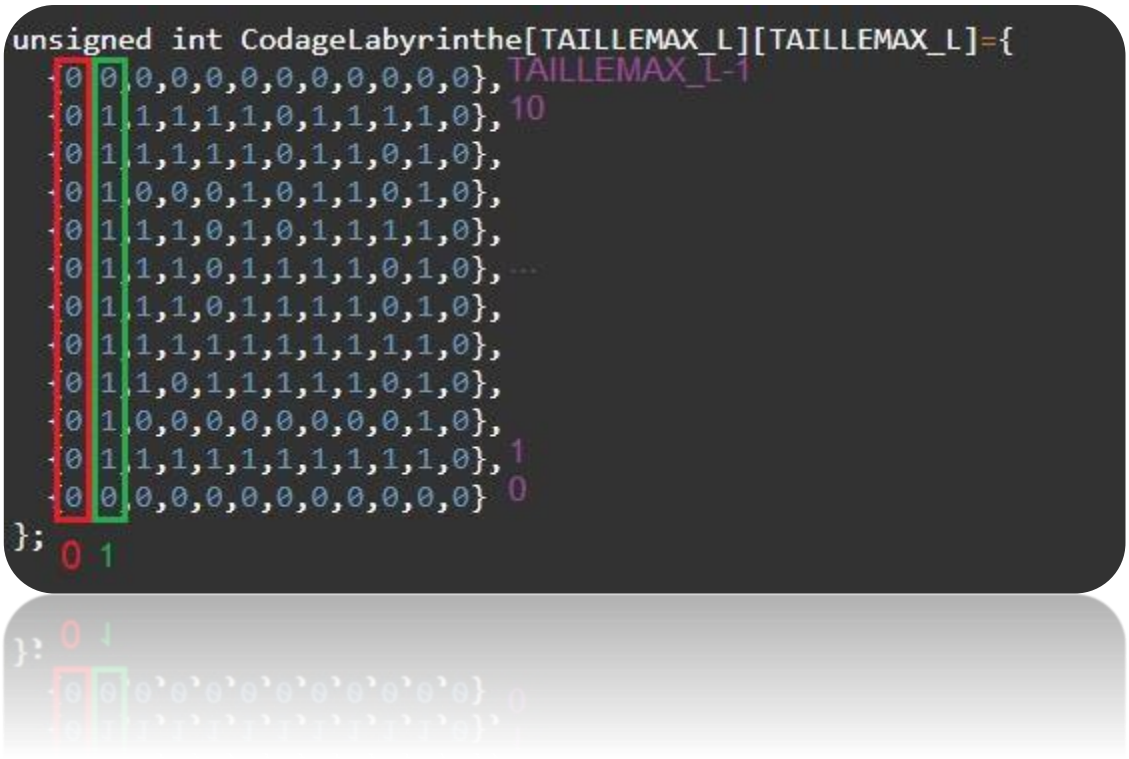
REALISATION

Par soucis de compatibilité, nous avons préféré utiliser un IDE en ligne pour coder et assembler notre programme. Le code complet est disponible en annexes ainsi que les fonctions non illustrées et marquées d'un astérisque.

REPRESENTATION INFORMATIQUE DU PLAN

Pour représenter notre plan nous avons utilisé une matrice binaire c'est-à-dire qui ne peut contenir que des 0 et des 1. 0 pour un mur et 1 pour un espace vide. Chaque case étant notée pas son abscisse et son ordonnée de 0 à TAILLEMAX_L-1. TAILLEMAX_L = 12 dans notre cas et partant des zéro, les 144 cases seront notées :

(0,0), (0,1), (0,2), …, (0,TAILLEMAX_L-2), (0,TAILLEMAX_L-1)
(1,0), (1,1), (1,2), …, (1,TAILLEMAX_L-2), (1,TAILLEMAX_L-1)



STRUCTURE DU PROGRAMME PRINCIPAL (MAIN)

Le main est la fonction principale du programme, celle qui le dirige et est la première à se lancer. Ici notre main va définir les coordonnées de l'entrée et de la sortie grâce à la fonction InitCoordonnees* appliquée aux cases A (1,TAILLEMAX_L -2) et B (TAILLEMAX_L -2,1) où TAILLEMAX_L représente la longueur d'un côté de la matrice représentant le plan. Comme ici TAILLEMAX_L = 12, A (1,10) et B (10, 1).

```
int main (void) {
  Entree = InitCoordonnees(1,TAILLEMAX_L-2);
  Sortie = InitCoordonnees(TAILLEMAX_L-2,1);

  AfficherCodageLabyrinthe();
  TrajectoireRobot = Recherche(Entree, Sortie);

  return 0;
}
```

Le programme va ensuite lancer la fonction AfficherCodageLabyrinthe.

FONCTION AFFICHERCODAGELABYRINTH

Cette fonction a pour but de dessiner le labyrinthe sur l'écran : grâce à deux boucles for imbriqués, l'une pour lin de 0 à TAILLEMAX_L et l'autre pour col de 0 à TAILLEMAX_L qui doit écrire « * » ou « » en fonction de mur ou espace vide.

```
void AfficherCodageLabyrinthe(void) {
  // Affiche murs et passages du labyrinthe tel que code dans CodageLabyrinthe[][]
  unsigned int lin, col;

  for (lin=0; (lin<TAILLEMAX_L); ++lin) {
    for (col=0; (col<TAILLEMAX_L); ++ col) {
      printf("%c",!CodageLabyrinthe[lin][col]?'*':' ');
    }
    printf("\n");
  }
}
```

Cette fonction ne retourne aucune valeur, notre fonction main peut donc continuer.

```
int main (void) {
  Entree = InitCoordonnees(1,TAILLEMAX_L-2);
  Sortie = InitCoordonnees(TAILLEMAX_L-2,1);

  AfficherCodageLabyrinthe();
  TrajectoireRobot = Recherche(Entree, Sortie);

  return 0;
}
```

Nous allons maintenant nous intéresser à la fonction recherche qui va être appliquée aux variables Entree et Sortie, puis stockées dans la variable TrajectoireRobot.

FONCTION RECHERCHE

La fonction recherche est la fonction qui va calculer le chemin à parcourir pour aller de l'entrée à la sortie. Etant la dernière fonction du main, elle est la plus lourde et importante car celle qui va ensuite appelle toutes les dernières fonctions. Ses deux premières lignes pointent les variables CellSuivante et CheminCourant grâce aux pointeurs COORDONNEES et CHEMIN.

```
CHEMIN * Recherche(COORDONNEES * Entree, COORDONNEES * Sortie) {
  // Renvoi du premier chemin de l'Entree vers la Sortie dans le Labyrinthe

  COORDONNEES * CellSuivante;
  static CHEMIN * CheminCourant;

  // Initialisation du chemin
  CheminCourant = AjouteDansChemin(Entree,NULL);

  while (!Egal(CheminCourant->Coordonnees, Sortie)) {
      CellSuivante    = NextCellNord(CheminCourant->Coordonnees);
      if (Ouvert(CellSuivante) && EnDehors(CellSuivante,CheminCourant)) {
        CheminCourant = AjouteDansChemin(CellSuivante, CheminCourant);
        AfficherParcours (CheminCourant);
        printf("nord\n");
      }
      else {
        CellSuivante = NextCellEst(CheminCourant->Coordonnees);
        if (Ouvert(CellSuivante) && EnDehors(CellSuivante,CheminCourant)) {
          CheminCourant = AjouteDansChemin(CellSuivante, CheminCourant);
          AfficherParcours (CheminCourant);
          printf("est\n");
        }
     else {
          CellSuivante = NextCellSud(CheminCourant->Coordonnees);
          if (Ouvert(CellSuivante) && EnDehors(CellSuivante,CheminCourant)) {
            CheminCourant = AjouteDansChemin(CellSuivante, CheminCourant);
            AfficherParcours (CheminCourant);
            printf("sud\n");
          }
       else {
            CellSuivante = NextCellOuest(CheminCourant->Coordonnees);
            if (Ouvert(CellSuivante) && EnDehors(CellSuivante,CheminCourant)) {
              CheminCourant = AjouteDansChemin(CellSuivante, CheminCourant);
              AfficherParcours (CheminCourant);
              printf("ouest\n");
            }
```

La première fonction appelée est la fonction AjouteDansChemin qui va ajouter au chemin les coordonnées du chemin courant, la position actuelle.

FONCTION AJOUTEDANSCHEMIN

Cette fonction a pour but de permettre au programme de stocker le chemin à parcourir grâce au pointeur CHEMIN en lui allouant un espace mémoire indexé.

```
CHEMIN * AjouteDansChemin(COORDONNEES * Coordonnees, CHEMIN * CheminConnu) {
  // Ajoute une cellule du labyrinthe au chemin connu, et renvoie le nouveau
  // chemin obtenu
  static CHEMIN * Cell;

  Cell               = malloc(sizeof(CHEMIN));
  Cell->Coordonnees  = Coordonnees;
  Cell->CrossedCells = CheminConnu;

  return Cell;
}
```

La suite du programme est composée d'une boucle while se répétant tant que les coordonnées ne correspondent pas à celles de la sortie par la fonction Egal*. C'est dans cette partie que le résultat visible pour l'utilisateur va être calculé puis affiché.

```
CHEMIN * Recherche(COORDONNEES * Entree, COORDONNEES * Sortie) {
  // Renvoi du premier chemin de l'Entree vers la Sortie dans le Labyrinthe

  COORDONNEES * CellSuivante;
  static CHEMIN * CheminCourant;

  // Initialisation du chemin
  CheminCourant = AjouteDansChemin(Entree,NULL);

  while (!Egal(CheminCourant->Coordonnees, Sortie)) {
      CellSuivante    = NextCellNord(CheminCourant->Coordonnees);
      if (Ouvert(CellSuivante) && EnDehors(CellSuivante,CheminCourant)) {
        CheminCourant = AjouteDansChemin(CellSuivante, CheminCourant);
        AfficherParcours (CheminCourant);
        printf("nord\n");
      }
      else {
        CellSuivante = NextCellEst(CheminCourant->Coordonnees);
        if (Ouvert(CellSuivante) && EnDehors(CellSuivante,CheminCourant)) {
          CheminCourant = AjouteDansChemin(CellSuivante, CheminCourant);
```

FONCTIONS NEXTCELL(NORD/SUD/EST/OUEST)

Ensuite on cherche une case vide autour de nous dont on va stocker les coordonnées dans CellSuivante.

```
CellSuivante    = NextCellNord(CheminCourant->Coordonnees);
```

On va commencer par les coordonnées de la case au nord à partir des coordonnées actuelles. Le programme va appeler la fonction NextCellNord qui va stocker dans CellSuivante la valeur de la case au nord de la case courante, en ajoutant ou 1 ou -1 en abscisse ou en ordonnée.

```
COORDONNEES * NextCellNord(COORDONNEES * CoordonneesCourantes) {
  // Coordonnees de la case nord a partir de la case courante
  static COORDONNEES * CoordonneesNext;

  CoordonneesNext = malloc(sizeof(COORDONNEES));
  CoordonneesNext->lin=CoordonneesCourantes->lin-1;
  CoordonneesNext->col=CoordonneesCourantes->col;
  return CoordonneesNext;
}
```

Ensuite le programme lance une condition if, qui est exécutée si la cellule suivante, stockée au préalable dans CellSuivante est ouverte et si celle-ci ne fais pas partie du chemin déjà parcouru (grâce aux fonctions Ouvert* et En Dehors*).

```
 if (Ouvert(CellSuivante) && EnDehors(CellSuivante,CheminCourant)) {
   CheminCourant = AjouteDansChemin(CellSuivante, CheminCourant);
   AfficherParcours (CheminCourant);
   printf("nord\n");
 }
```

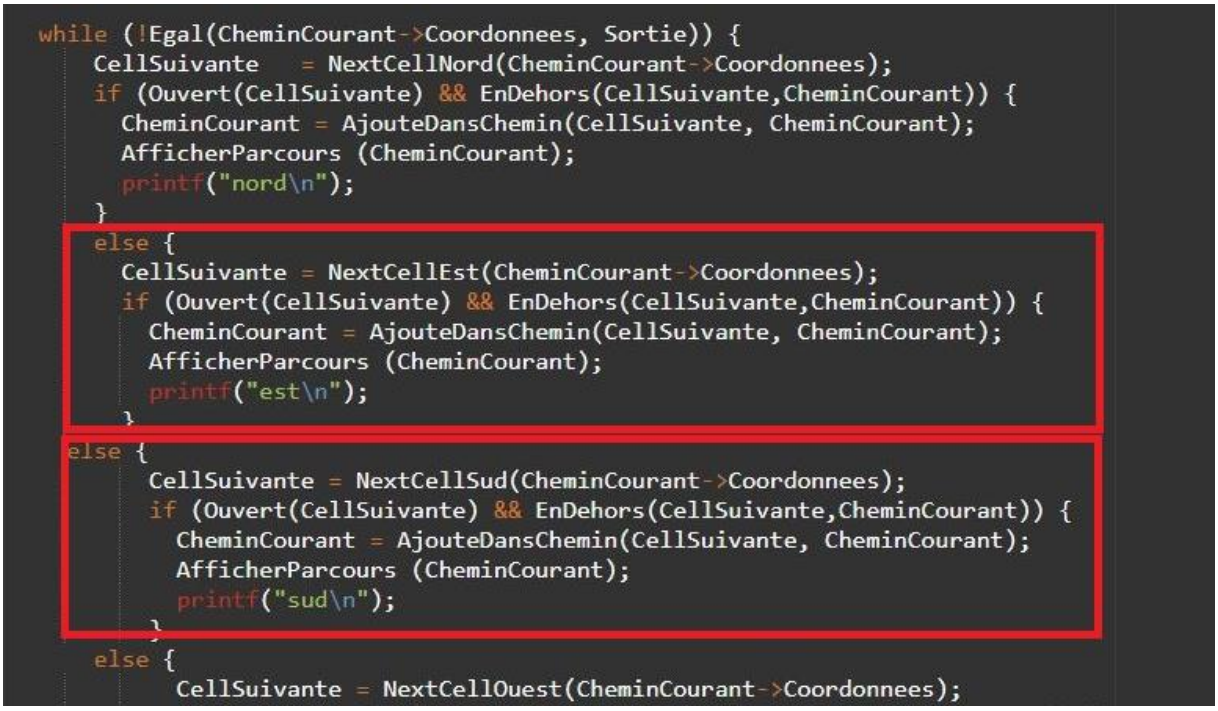
Si la condition est vérifiée, cette case est ajoutée au chemin grâce à la fonction AjouteDansChemin puis le programme lance la fonction AfficherParcours, qui va donner les coordonnées de la case, enfin il affiche la direction prise avec le mot « nord ».

FONCTION AFFICHERPARCOURS

```
void AfficherParcours (CHEMIN * Cell) {
  // Affiche la cellule parcourue
  printf("\n lin = %d ",Cell->Coordonnees->lin); printf("col = %d, ",Cell->Coordonnees->col);
}
```

Cette fonction va afficher à l'écran les coordonnées de la cellule, valeurs stockées dans la variable Cell, on donne lin= et col= pour les coordonnées des lignes et des colonnes.

Si la condition if n’est pas vérifiée, le programme continue vers le même type de bloc mais concernant cette fois une autre direction, si chaque direction est vérifiée, la première à être reconnue comme une case vide et non explorée sera la case enregistrée comme case suivante.



BILANS PERSONNELS

LABARRE CHARLES

RISCH GUILLAUME

CONCLUSION

ANNEXES

- Fonction InitCoordonnee

```
// ------------------------------------------------------------------------
// Initialisations
// ------------------------------------------------------------------------

COORDONNEES * InitCoordonnees(unsigned int lin, unsigned int col) {
  // Affectation de coordonnees
  static COORDONNEES * UneCase;

  UneCase      = malloc(sizeof(COORDONNEES));
  UneCase->lin = lin;
  UneCase->col = col;
  return UneCase;
}
```

- Fonction Egal

```
unsigned int Egal(COORDONNEES * Cell1, COORDONNEES * Cell2) {
  // Verifie si Cell1 est egale a Cell2
  return (Cell1->lin==Cell2->lin) && (Cell1->col==Cell2->col);
}
```

- Fonction Ouvert

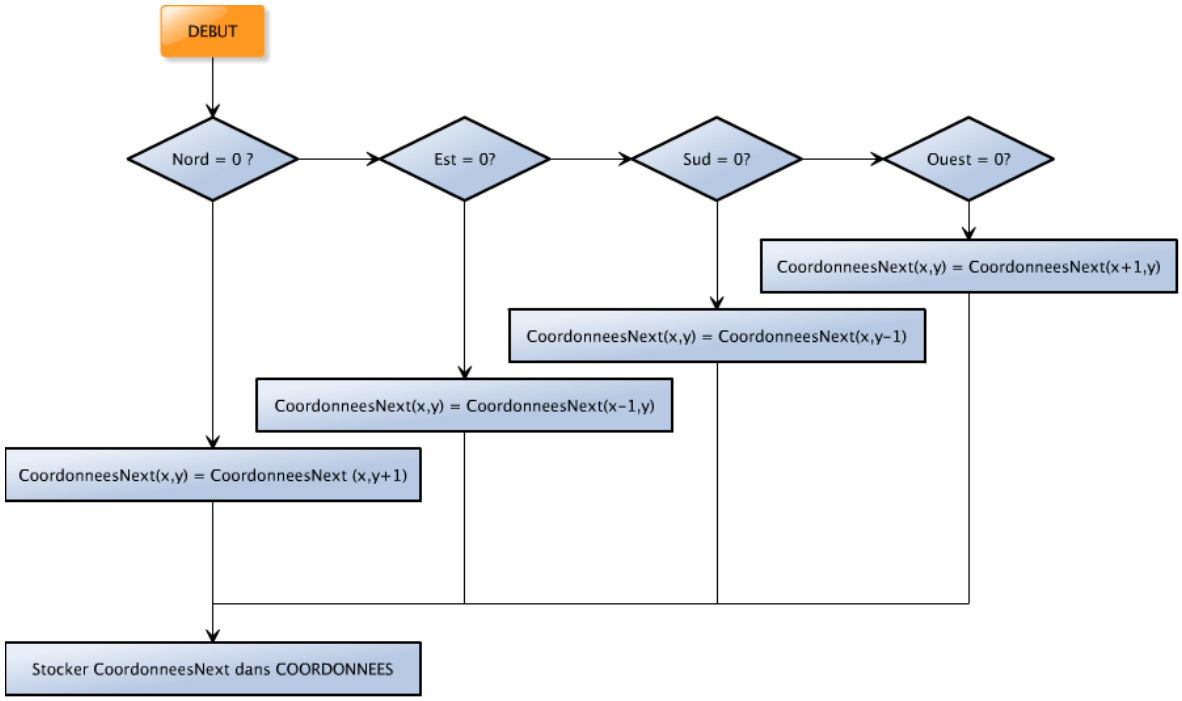
```
unsigned int Ouvert (COORDONNEES * CellCourante) {
  // Renvoie le statut de la cellule courante : mur ou passage
  return CodageLabyrinthe[CellCourante->lin][CellCourante->col];
}
```

- Fonction EnDehors

```
unsigned int EnDehors(COORDONNEES * Cell, CHEMIN * CheminParcouru) {
  // Verifie l'appartenance de Cell a CheminParcouru

  while ((CheminParcouru != NULL) && (!Egal(Cell,CheminParcouru->Coordonnees)))
    CheminParcouru=CheminParcouru->CrossedCells;
    return (CheminParcouru == NULL);
}
```

- Algorigramme



- Programme complet

```c
#include <stdio.h>
#include <stdlib.h>

#define TAILLEMAX_L 12  // taille Max du labyrinthe avec murs externes

// ------------------------------------------------------------------------
// Structures de donnees
// ------------------------------------------------------------------------

unsigned int CodageLabyrinthe[TAILLEMAX_L][TAILLEMAX_L]={
  {0,0,0,0,0,0,0,0,0,0,0,0},
  {0,1,1,1,1,1,0,1,1,1,1,0},
  {0,1,1,1,1,1,0,1,1,0,1,0},
  {0,1,0,0,0,1,0,1,1,0,1,0},
  {0,1,1,1,0,1,0,1,1,1,1,0},
  {0,1,1,1,0,1,1,1,1,0,1,0},
  {0,1,1,1,0,1,1,1,1,0,1,0},
  {0,1,1,1,1,1,1,1,1,1,1,0},
  {0,1,1,0,1,1,1,1,1,0,1,0},
  {0,1,0,0,0,0,0,0,0,0,1,0},
  {0,1,1,1,1,1,1,1,1,1,1,0},
  {0,0,0,0,0,0,0,0,0,0,0,0}
};
```

```c
// On a donc en termes de coordonnees :
// (0,0),(0,1),(0,2),...,(0,TAILLEMAX_L-1)
// (1,0),(1,1),(1,2),...,(1, TAILLEMAX_L-1)
// ...
// (TAILLEMAX_L-1,0),...,(TAILLEMAX_L-1,TAILLEMAX_L-1)
// avec
// Entree du labyrinthe : [1][TAILLEMAX_L-2]
// Sortie du labyrinthe : [TAILLEMAX_L-2][1]

typedef struct coordonnees {
  unsigned int lin;
  unsigned int col;
} COORDONNEES;

COORDONNEES * Entree, * Sortie;

typedef struct chemin {
  COORDONNEES * Coordonnees;
  struct chemin * CrossedCells;
} CHEMIN;

CHEMIN * TrajectoireRobot;


// ----------------------------------------------------------------------------
// Acces a la cellule suivante
// ----------------------------------------------------------------------------

COORDONNEES * NextCellNord(COORDONNEES * CoordonneesCourantes) {
  // Coordonnees de la case nord a partir de la case courante
  static COORDONNEES * CoordonneesNext;

  CoordonneesNext = malloc(sizeof(COORDONNEES));
  CoordonneesNext->lin=CoordonneesCourantes->lin-1;
  CoordonneesNext->col=CoordonneesCourantes->col;
  return CoordonneesNext;
}

COORDONNEES * NextCellEst(COORDONNEES * CoordonneesCourantes) {
  // Coordonnees de la case est a partir de la case courante
  static COORDONNEES * CoordonneesNext;

  CoordonneesNext = malloc(sizeof(COORDONNEES));
  CoordonneesNext->lin=CoordonneesCourantes->lin;
  CoordonneesNext->col=CoordonneesCourantes->col+1;
  return CoordonneesNext;
}

COORDONNEES * NextCellSud(COORDONNEES * CoordonneesCourantes) {
  // Coordonnees de la case sud a partir de la case courante
  static COORDONNEES * CoordonneesNext;

  CoordonneesNext = malloc(sizeof(COORDONNEES));
  CoordonneesNext->lin=CoordonneesCourantes->lin+1;
  CoordonneesNext->col=CoordonneesCourantes->col;
  return CoordonneesNext;
}
```

```
COORDONNEES * NextCellOuest(COORDONNEES * CoordonneesCourantes) {
  // Coordonnees de la case ouest a partir de la case courante
  static COORDONNEES * CoordonneesNext;

  CoordonneesNext = malloc(sizeof(COORDONNEES));
  CoordonneesNext->lin=CoordonneesCourantes->lin;
  CoordonneesNext->col=CoordonneesCourantes->col+1;
  return CoordonneesNext;
}


// ------------------------------------------------------------------------
// Initialisations
// ------------------------------------------------------------------------

COORDONNEES * InitCoordonnees(unsigned int lin, unsigned int col) {
  // Affectation de coordonnees
  static COORDONNEES * UneCase;

  UneCase      = malloc(sizeof(COORDONNEES));
  UneCase->lin = lin;
  UneCase->col = col;
  return UneCase;
}


// ------------------------------------------------------------------------
// Gestion des listes de pointeurs (Chemin et Frontiere)
// ------------------------------------------------------------------------

CHEMIN * AjouteDansChemin(COORDONNEES * Coordonnees, CHEMIN * CheminConnu) {
  // Ajoute une cellule du labyrinthe au chemin connu, et renvoie le nouveau
  // chemin obtenu
  static CHEMIN * Cell;

  Cell               = malloc(sizeof(CHEMIN));
  Cell->Coordonnees  = Coordonnees;
  Cell->CrossedCells = CheminConnu;

  return Cell;
}

void EffaceDansChemin(CHEMIN * DebutChemin) {
  // Rend la mC)moire precedemment allouC)e a partir de DebutChemin
  CHEMIN * CellSuivante;

  CellSuivante = DebutChemin->CrossedCells;
  while (CellSuivante != NULL) {
    free(DebutChemin);
    DebutChemin = CellSuivante;
    CellSuivante = DebutChemin->CrossedCells;
  }
}
```

```
// ------------------------------------------------------------------------
// Recherche et Affichage
// ------------------------------------------------------------------------

void AfficherCodageLabyrinthe(void) {
  // Affiche murs et passages du labyrinthe tel que code dans CodageLabyrinthe[][]
  unsigned int lin, col;

  for (lin=0; (lin<TAILLEMAX_L); ++lin) {
    for (col=0; (col<TAILLEMAX_L); ++ col) {
      printf("%c",!CodageLabyrinthe[lin][col]?'*':' ');
      printf(" ");
    }
    printf("\n");
  }
}

void AfficherParcours (CHEMIN * Cell) {
  // Affiche la cellule parcourue
  printf("\n lin = %d ",Cell->Coordonnees->lin);
  printf("col = %d, ",Cell->Coordonnees->col);
}

unsigned int Egal(COORDONNEES * Cell1, COORDONNEES * Cell2) {
  // Verifie si Cell1 est egale a Cell2
  return (Cell1->lin==Cell2->lin) && (Cell1->col==Cell2->col);
}

unsigned int EnDehors(COORDONNEES * Cell, CHEMIN * CheminParcouru) {
  // Verifie l'appartenance de Cell a CheminParcouru

  while ((CheminParcouru != NULL) && (!Egal(Cell,CheminParcouru->Coordonnees)))
    CheminParcouru=CheminParcouru->CrossedCells;
    return (CheminParcouru == NULL);
}

unsigned int Ouvert (COORDONNEES * CellCourante) {
  // Renvoie le statut de la cellule courante : mur ou passage
  return CodageLabyrinthe[CellCourante->lin][CellCourante->col];
}
```

```
CHEMIN * Recherche(COORDONNEES * Entree, COORDONNEES * Sortie) {
  // Renvoi du premier chemin de l'Entree vers la Sortie dans le Labyrinthe

  COORDONNEES * CellSuivante;
  static CHEMIN * CheminCourant;

  // Initialisation du chemin
  CheminCourant = AjouteDansChemin(Entree,NULL);

  while (!Egal(CheminCourant->Coordonnees, Sortie)) {
      CellSuivante   = NextCellNord(CheminCourant->Coordonnees);
      if (Ouvert(CellSuivante) && EnDehors(CellSuivante,CheminCourant)) {
        CheminCourant = AjouteDansChemin(CellSuivante, CheminCourant);
        AfficherParcours (CheminCourant);
        printf("nord\n");
      }
      else {
        CellSuivante = NextCellEst(CheminCourant->Coordonnees);
        if (Ouvert(CellSuivante) && EnDehors(CellSuivante,CheminCourant)) {
          CheminCourant = AjouteDansChemin(CellSuivante, CheminCourant);
          AfficherParcours (CheminCourant);
          printf("est\n");
        }
    else {
          CellSuivante = NextCellSud(CheminCourant->Coordonnees);
          if (Ouvert(CellSuivante) && EnDehors(CellSuivante,CheminCourant)) {
            CheminCourant = AjouteDansChemin(CellSuivante, CheminCourant);
            AfficherParcours (CheminCourant);
            printf("sud\n");
          }
      else {
            CellSuivante = NextCellOuest(CheminCourant->Coordonnees);
            if (Ouvert(CellSuivante) && EnDehors(CellSuivante,CheminCourant)) {
              CheminCourant = AjouteDansChemin(CellSuivante, CheminCourant);
              AfficherParcours (CheminCourant);
              printf("ouest\n");
            }
      }
    }
   }
  }
  return CheminCourant;
}


// ------------------------------------------------------------------------
// main
// ------------------------------------------------------------------------

int main (void) {
  Entree = InitCoordonnees(1,TAILLEMAX_L-2);
  Sortie = InitCoordonnees(TAILLEMAX_L-2,1);

  AfficherCodageLabyrinthe();
  TrajectoireRobot = Recherche(Entree, Sortie);

  return 0;
}
```