LABYRINTHE

BOLTENHAGEN Mathilde
L2S3 G1

Sommaire :

1. Introduction

Le Labyrinthe est un jeu qui se représente par un espace creusé d'une entrée à une sortie définies. On distingue donc des chemins et des murs. Les chemins peuvent déboucher ou non sur la sortie et ainsi il existe un passage unique et obligatoire entre l'entrée et la sortie. Le projet consiste à faire un labyrinthe muni de certaines conditions. Pour cela, le travail se décompose entre trois grands axes. Tout d'abord, il faudra définir le cadre du labyrinthe qui est une matrice de booléens. On y trouvera les murs dans les cases “false” et les chemins dans les autres. De même, on s'interessera à la définition des chemins ainsi que leurs différentes utilisations nécessaires. La phase principale est la connection que l'on va créer entre ces deux dernières structures tout en respectant les contraintes du jeu. Ainsi on développera un affichage permettant de mieux visualiser le code et de pouvoir partager le projet. Pour finir, on concluera sur des exemples précis munis d'images explicatives.

2. Sortes

A. Labyrinthe

a. Specification

spec : LABY (ELEM) étend BASE

sorte : Laby, pos

Opérations (pos est une structure de deux entiers) :

-init : → Laby

-AjoutMur : Laby * pos → Laby

-retirerMur : Laby * pos → Laby

-Nb_cases_Mur : Laby → Int

-case_chemin : Laby * pos → Bool

Préconditions :

pré AjoutMur(l, p) = p ∈ l

pré retirerMur (l,p)= p ∈ l

pré case_chemin (l,p) = p ∈ l

pré Nb_cases_Mur (l) = l <> NULL

Axiomes :

retirerMur (AjoutMur (l, p1,),p2)= if p1=p2 then l
else AjoutMur(retirerMur(l,p2),p1) fi

case_chemin (AjoutMur(l,p1),p2) = if p1=p2 then false
else case_chemin(l,p2) fi

case_chemin(retirerMur(l,p1),p2)= if p1=p2 then true
else case_chemin(l,p2) fi

Nb_cases_Mur (AjoutMur(l,p))= 1+ Nb_case_Mur(l)

b. Algorithme

Structure sur position d'entiers :

```
typedef struct strpos { int x ; int y ; } *pos , strpos ;
```

Structure du Labyrinthe :

```
typedef struct strlab  { int x ; int y ; Bool** v ; } *laby, strlab ;
```

Structure des Booléens :

```
typedef enum {true=1,false=0,bonhomme=8,} Bool;
creation d'un nouveau booléen, bonhomme, utile pour l'affichage
```

Allocation de la place de la structure Laby :

```
laby init ();
```

Creation d'un Laby :

```
void cree_lab(int l,int c,laby lab);
```

paramètre : *-l : un pointeur sur un labyrinthe alloué*

-c : un entier ; le nombre de colones

-l : un entier ; le nombre de ligne

Libère la place du Labyrinthe :

```
void libere_lab(laby lab);
```

paramètre *: - lab : un pointeur sur un labyrinthe*

Met à 1 la case du labyrinthe correspondant à la position :

```
void retirerMur (laby l, pos p);
```

paramètre : *-l : un pointeur sur un labyrinthe*

-p : pointeur sur une position

Met à 0 la case du labyrinthe correspondant à la position :

```
void AjoutMur (laby l, pos p);
```

paramètre : *-l : un pointeur sur un labyrinthe*

-p : pointeur sur une position

Met à 8 la case du labyrinthe correspondant à la position :

```
void bonhomme_init(laby l, pos p);
```

paramètre : *-l : un pointeur sur un labyrinthe*

-p : pointeur sur une position

fonction auxilaire de "Nb_cases_Mur" : renvoie 1 quand on a un mur, 0 sinon :

```
int contraire_case(int i);
```

paramètre : *-i : un entier correspondant à un bool*

return *: int*

Compte le nombre de case de type Mur, utile pour fonction "finition" :

```
int Nb_cases_Mur (laby l);
```

paramètre : *-l : un pointeur sur un labyrinthe*

return : *int*

Annonce si on est sur une case Chemin (true) ou non (false):

```
Bool case_chemin (laby l, strpos p);
```

paramètre : *-l : un pointeur sur un labyrinthe*

-p : pointeur sur une position

return : *bool*

Donne le nombre de cases que représente le pourcentage en paramètre :

```
int pourcentage_voulu(int i, laby l);
```

paramètre : *-l : un pointeur sur un labyrinthe*

-i : un entier entre 40 et 60 selon labyrinthe

return : *int*

B. Chemins

a.Spécification

spec : CHEMIN (ELEM) étend BASE

sorte : Chem

opération :

-creation : → Chem

-adjq : Chem * pos → Chem

-supt : Chem → Chem

-tete : Chem → pos

-clone : Chem → Chem

-kieme_Chem : Chem* int → pos

-longueur : Chem → int

précondition :

-pré supt(c) = c<>NULL

pré tete(c) = c<>NULL

pré kieme_Chem (c,i) = I ∈]0;longueur(c)[

axiomes :

-supt (adjq(c,p))= if l=NULL then l else adjq(supt(c),p) fi

-tete(adjq(c,p)) = if l= NULL then p else tete(c) fi

-kieme_Chem (adjq(c,p),i)=if i=longueur(c) then p else kieme_Chem (c,i) fi

-longueur(adjq(c,p))= if c =NULL then 1 else 1+ longueur(c) fi

b. Algorithmes

Structure d'un Chemin, pointeur sur la queue : (file)

```
typedef struct strchem { strpos * a ; struct strchem* suiv ; } *chem , strchem;
```

Renvoie Null :

```
chem creation();
```

Ajout une position à la queue du Chemin :

```
chem adjq(chem c, pos a1);
```

paramètre *: -c : pointeur sur un Chemin*

-a1 : pointeur sur une position

return : *Chem*

Retire la première position du chemin :

```
void supt(chem c);
```

paramètre : *-c : pointeur sur un chemin*

Détruit un Chemin entier en rappelant "supt" autant de fois que de pos :

```
void destruction (chem c);
```

paramètre : *-c : pointeur sur un chemin*

Donne le nombre de position que contient le chemin :

```
int longueur(chem c);
```

paramètre : *-c : pointeur sur un chemin*

return : *int*

Renvoie la première position du Chemin :

```
pos tete (chem c);
```

paramètre : *-c : pointeur sur un chemin*

return : *pos*

Renvoie un clone d'un chemin avec les allocations mémoires necessaires :

```
chem clone (chem c);
```

paramètre *: -c : pointeur sur un chemin*

return : *chem*

Renvoie la k-ième position d'un chemin, on l'utilisera avec “alea_nb_chemin”:

```
strpos kieme_chem (chem c, int n);
```

paramètre *: -c : pointeur sur un chemin*

-n : un entier, la place de la position voulue

return : *strpos*

Renvoie un nombre compris entre 2 et la longueur du chemin :

```
int alea_nb_chemin(chem c);
```

paramètre *: -c : pointeur sur un chemin*

return : *int*

Renvoie aléatoirement un entier égal à 1 ou 0, utile dans de nombreuses fonctions :

```
int unounull ();
```

return *: int*

3. Le Jeu

A. Les contraintes du chemin

Le Labyrinthe est composée d'une position de départ, l'entrée et d'une position d'arrivée, la sortie. Ces dernières doivent être désignées aléatoirement selon le nombre de colones du Labyrinthe. Elles ne doivent pas être dans les extrémités :

```
int alea_x (laby l);  /*renvoie un nombre aleatoire entre 1 et l->x-1*/
strpos entree(laby l);
strpos sortie(laby l);
```

paramètre : *-l :pointeur sur un labyrinthe*

Les positions suivantes, entre celle de début et celle de fin, ne doivent pas créer des places publiques. Pour cela, il faut regarder autour de la position donnée si ses voisines sont des chemins de tel sorte que la nouvelle donne ou non un carré de 4 positions. Il y a quatres cas à vérifier, en haut à gauche, en bas à gauche, en haut à droite et en bas à droite :

```
Bool ca_donnera_place_publique(laby l, pos a1);
```

paramètre : *-l : pointeur sur un labyrinthe*

-a1 : pointeur sur la position à étudier

return : *bool : true si ça donne une place publique, false sinon*

Un chemin ne doit pas passer plusieurs fois au même endroit. Il faut donc vérifier que la position est de type Mur :

```
Bool on_est_deja_passe_par_la(laby l, pos a);
```

paramètre : *-l : pointeur sur un labyrinthe*

-a : pointeur sur la position à étudier

return : *bool : true si la position est de type Chemin, false sinon*

Les Chemins ne doivent pas non plus sortir du labyrinthe ou même toucher les bords. Il faut donc vérifier que la position ne sort pas de cette condition :

```
Bool on_est_au_bord(pos a,laby l);
```

paramètre : *-l : pointeur sur un labyrinthe*

-a : pointeur sur la position à étudier

return : *bool : true si la position est au bord ou en dehors, false sinon*

Les chemins ne doivent pas se croiser. Pour cela, on vérifie que la position n'a pas de voisines à ses cotés (hormis la position du chemin auquel elle appartient, la position précedante) :

```
Bool on_touche_ailleurs(pos av, strpos ap, laby l);
```

paramètre : *-l : pointeur sur un labyrinthe*

-av : pointeur sur la position précédante

-ap : position à étuidier

return : *bool : true si la position a des voisins, false sinon*

B. Méthode de tracage

Pour commencer, on va tracer le chemin initial qui part de l'entrée et va à la sortie de façon aléatoire (ou presque). Pour ce premier chemin, j'ai décidé qu'on ne pourrait pas descendre. Le principe de "chem_init" est de créer une file de position. Ma méthode est de la position suivante, puis s'assurer qu'elle vérifie toutes les conditions necessaires c'est à dire qu'elle ne touche pas le bord, qu'elle ne donne pas de place publique ou qu'elle ne fait pas déjà partie du chemin. Si les conditions sont vérifiées, on l'ajoute à la queue et on modifie le labyrinthe. On refait cette étape tant qu'on est pas à l'avant derniere ligne. A cette ligne, on regarde où se situe la sortie et on se rapproche (négligeant donc les places publiques) et l'aléatoire. :

```
chem chem_init (laby l, strpos ent, strpos sort);
```

paramètre : *-l : pointeur sur un labyrinthe*

-a : pointeur sur la position à étudier

return : *-c : le chemin initial*

A partir du chemin initial, on crée les autres chemins qui, eux, débouchent sur un cul de sac. Pour cela, on choisit une position aléatoire du chemin initial et à partir de celle-ci on reprend le principe du chemin initial en ajoutant la condition de ne pas croiser un chemin. Un chemin s'arrête s'il a atteint la longueur demandée ou s'il est bloqué par les conditions. Pour ces nouveaux chemins, j'autorise de descendre. Ces nouveaux chemins sont ajoutés à la queue de l'ancien :

```
chem chemins_suivants(chem c_init,int longueur_max,laby l);
```

paramètre : *-l : pointeur sur un labyrinthe*

-c : le chemin initial

-longueur_max : int, longueur voulue

return : *-c : le chemin initial + le nouveau crée*

Suite à cette fonction, on désire créer des chemins afin de remplir le labyrinthe. Ainsi, le principe est de rappeler la fonction "chemins_suivants" autant de fois que voulus. Pour cela, j'utilise la fonction qui compte le nombre de cases Mur et je fais une boucle tant que mon pourcentage de case Mur est supérieur à celui voulu :

```
chem finition(laby l, chem c,int pourcentage_traduit);
```

paramètre : *-l : pointeur sur un labyrinthe*

-c : le chemin initial

-pourcentage_traduit : int venant de la fonction "pourcentage_voulu"

return : *-c : le chemin finale composée de l'initial et des autres*

C. Le tableau des Directions

Dans les fonctions "chem_init" et "chemins_suivants" , je parle d'une nouvelle position qui est la position suivante. Cette position se trouve grâce à l'ancienne. Par exemple, la deuxième position à été définie grâce à la position d'entrée. Pour cela, j'ai créé quatres petites fonctions qui renvoie une nouvelle position à côté de la position précédante :

```
strpos monter (pos b);
strpos descendre (pos b);
strpos gauche (pos b);
strpos droite (pos b);
```

paramètre : *-b : ancienne position, allouée*

return : *strpos: nouvelle position non allouée car allouée dans "adjq"*

Le chemin devant être aléatoire, je crée la fonction "avancer" qui prend la position précendante et un tableau d'entier et renvoie, parmis les choix possibles donné

dans le tableau, une nouvelle position. Ce tableau est composé de quatres entiers vallant 0,1,2 et 3 à la base. Ainsi, je choisis un nombre alétoire est comprit entre 0 et 3 inclus et retrouve la case correpondante du tableau à laquelle il vaut. Si une case du tableau vaut 4 elle est inaccessible. J'ai donc un cas de retour à la fonction de manière récursive afin d'atteindre une direction :

strpos avancer (pos avant, int* t)

strpos avancer (pos avant, int*t);

paramètre : *-b : ancienne position, allouée*

-t : tableau de 4 entiers

return : *strpos: nouvelle position non allouée car allouée dans "adjq"*

```
        strpos avancer (pos avant, int*t)
{
    int x;
    x=(int)rand()%4;
    if (x==t[3]) return monter(avant);
    else
    {
        if (x==t[2]) return descendre(avant);
        else
        {
            if (x==t[1]) return gauche(avant);
            else
            {
                if (x==t[0]) return droite (avant);
                else return avancer(avant,t) ;
            }
        }
    }
}
```

Cette fonction aléatoire permet donc de choisir une direction en fonction des directions possibles. Ainsi le tableau varie quand certaines directions ne respecte pas les conditions. Dans les fonctions "chem_init" et "chemins_suivants", on modifie alors

le tableau lorsque l'on rentre dans la boucle des conditions. On compare ainsi la nouvelle position et l'ancienne afin de déterminer quelle direction ne marche pas et on met la valeur 4 dans la case correspondant à cette direction :

```
int *comparaison(pos av, pos ap, int* t);
```

paramètre : *-av : ancienne position, allouée*

-ap : la nouvelle position

-t le tableau à modifier

return : *int* : le tableau t avec un 4 supplémentaire dans une case*

Ainsi, il arrive des cas où un chemin est entièrement bloqué (au bord par exemple) et le tableau est rempli de chiffre 4. On aurait une boucle infini lors de la cherche de la future direction ("avancer"). J'ai donc décidé de refaire le chemin dans ce cas là :

```
Bool faut_refaire (int*i);
```

paramètre : *-t le tableau à vérifier*

return : *bool, true si le tableau est rempli de 4, false sinon*

4. Affichage

L'affichage est l'étape finale. On affiche le labyrinthe avec ses valeurs, 0 pour un mur, 1 pour un chemin et 8 pour le joueur ainsi que les regles du jeu:

```
void affichage(laby l);
void cmd_et_regle();
```

Puis on crée une interface graphique prenant les caractères entrés au clavier qui permet de se deplacer :

```
void interface_graph(laby l, chem chemin);
```

Dans cette fonction, on évalue la lettre entrée, on regarde si l'action est possible et on renvoie un nouvel affichage du labyrinthe. Pour cela, on efface le précédant et on déplace le 8 :

```
void efface (laby l);
void deplacer_bonhomme(laby l, strpos a1, strpos a2);
```

paramètre *: -a1 : l'ancienne position du 8 que l'on garde à chaque fois*

-a2 : la nouvelle que l'utilisateur veut essayer

Arrivé en haut, le jeu se termine, on le lui annonce et la lettre du joueur se met automatiquement à 'q' afin de quitter proprement et de libérer la mémoire utilisée :

```
void GAGNE(chem chemin, laby l);
```

L'option 'a' permet d'optenir un autre affichage du même principe que le précédant avec un pointeur sur fonction qui vaut soit “affichage” ou “copie_affichage_sauvage”, un affichage inspiré un exercice de M.Sauvage :

```
void (*style_affichage) (laby) = affichage ;
```

5. Exemples via des captures d'écran

A. Le chemin initial

```
0 0 0 0 0 1 0 0
0 0 0 0 1 1 0 0
0 0 0 0 1 0 0 0
0 0 0 0 1 1 1 0
0 0 0 0 0 0 1 0
0 0 0 0 0 1 1 0
0 0 0 1 1 1 0 0
0 0 0 8 0 0 0 0
                LABYRINTHE
a : changer le mode d'affichage
h : aller en HAUT       b : aller en BAS
d : aller à DROITE      g : aller à GAUCHE
q : QUITTER
```

On a bien le chemin tracé avec des 1 de l'entrée à la sortie et le joueur se situe à l'entrée. Les zéros représentent les murs.

B. Le Labyrinthe complet

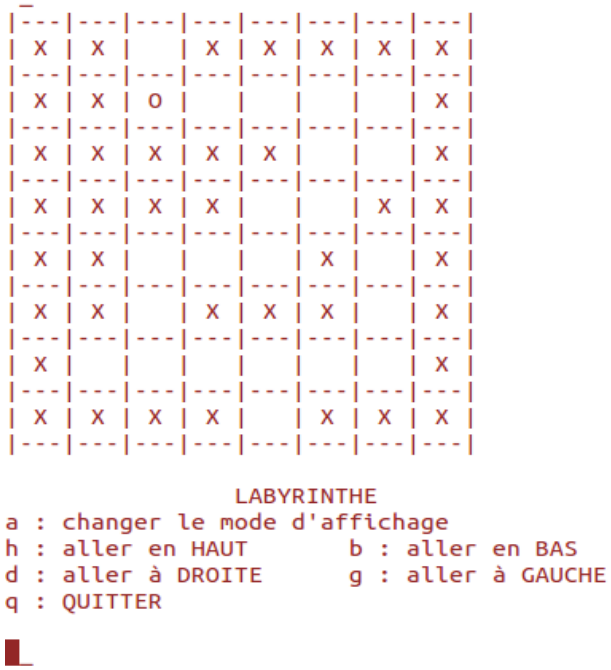
```
0 0 0 0 0 1 0 0
0 0 1 1 1 1 1 0
0 0 1 0 1 0 1 0
0 1 1 1 0 1 1 0
0 1 0 1 1 0 0 0
0 1 0 0 1 1 0 0
0 1 1 0 0 1 1 0
0 0 0 0 0 8 0 0
                LABYRINTHE
a : changer le mode d'affichage
h : aller en HAUT       b : aller en BAS
d : aller à DROITE      g : aller à GAUCHE
q : QUITTER
```

```
|---|---|---|---|---|---|---|---|
| X | X | X | X | X |   | X | X |
|---|---|---|---|---|---|---|---|
| X | X |   |   |   |   |   | X |
|---|---|---|---|---|---|---|---|
| X | X |   | X |   | X |   | X |
|---|---|---|---|---|---|---|---|
| X |   |   |   | X |   |   | X |
|---|---|---|---|---|---|---|---|
| X |   | X |   |   | X | X | X |
|---|---|---|---|---|---|---|---|
| X |   | X | X |   |   | X | X |
|---|---|---|---|---|---|---|---|
| X |   |   | X | X |   |   | X |
|---|---|---|---|---|---|---|---|
| X | X | X | X | X | O | X | X |
|---|---|---|---|---|---|---|---|

                LABYRINTHE
a : changer le mode d'affichage
h : aller en HAUT       b : aller en BAS
d : aller à DROITE      g : aller à GAUCHE
q : QUITTER
```

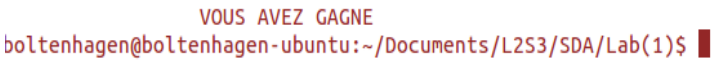
A gauche, on a un exemple d'un labyrinthe complet. Quand le joueur tape 'a', il obtient l'image de droite.

C. Avancer et fin du jeu

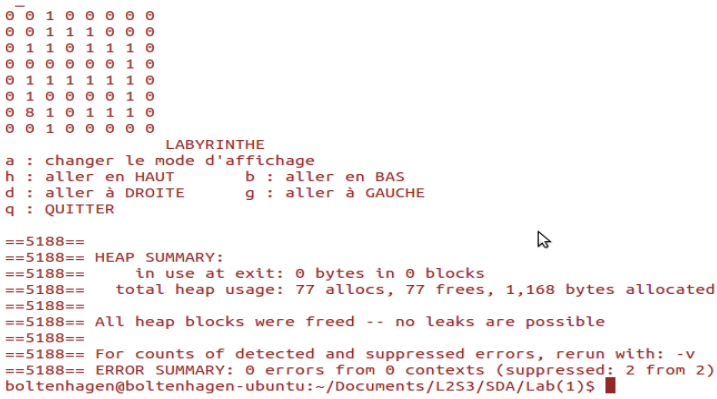


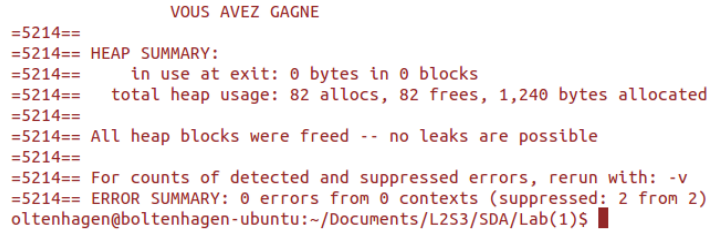
Le joueur peut donc avancer avec les regles du jeu et le rond se déplace si c'est possible. Après sa saisie, le joueur doit appuyer sur la touche entrée.

Dans cet exemple, on obtient l'image suivante si le joueur appuie sur 'h'.



D. Valgrind





Tous les frees necessaires sont faits en quittant ou en gagnant.

6. Conclusion

Le labyrinthe est ainsi un jeu dont le tracé varie et la solution aussi. On peut varier la taille ainsi que le nombre de chemins. Un grand labyrinthe avec beaucoup de chemins est de difficulté supérieure. Pour le tracage des chemins, nous avons utilisés des files qui permettent de rajouter la suite d'un chemin en queue de file. Il existe surement d'autres méthodes de tracage.

De Nombreuses options auraient pu être ajoutées. On aurait pu y trouver un compteur, notant le nombre de fois que le joueur a voulu allé vers un Mur. Dans ce même sens on aurait pu donner un nombre maximum de ces echecs. Le joueur aurait pu, à ce momement là, perdre au labyrinthe.

Avec un niveau supérieur et un temps de projet plus long, on aurait aussi pu rajouter des "monstres" qui se baladent dans les chemins et qu'il faut éviter, ou même des "monstres" qui traversent les murs... les options sont innombrables.

7. Codes

lab.h et lab.c : code des labyrinthes principalement (2.A.)

chemin.h et chemin.c : code des chemins principalement (2.B)

jeu.h et jeu.c : code rassemblant les deux (3.)

io.h et io.c : affichage (4.-5.)