M . DI PASQUALE Mathieu
L2 Informatique

Rapport de Projet :

Génération d’un labyrinthe et parcours par un automate

Année 2017 - 2018

Sommaire

Introduction

Définition du labyrinthe :

Un labyrinthe est composé de murs et de couloirs que l'on peut représenter sous la forme d'un tableau où les cellules valant 0 sont accessibles et les cellules valant 1 sont inaccessibles (ce sont des murs). Un labyrinthe de row lignes et col colonnes possède également obligatoirement une entrée et une sortie. Nous considérerons que l'entrée se situe en (0,0) (coin supérieur gauche) et la sortie en (row−1,col−1)(coin inférieur droit). Les bordures du labyrinthe seront infranchissables et nous ajouterons donc un mur tout autour.

Ce labyrinthe doit contenir en plus de son entrée et de sa sortie, au moins un chemin permettant de rejoindre les 2 bords.

Définition de l'automate :

L'automate sert à parcourir le labyrinthe. Il peut se déplacer sur toutes les cellules du labyrinthe, pour peu que ces dernières soient accessibles (pas un mur ou pas bloquée par des murs). Les déplacements le cet automate entre l'entrée et la sortie du labyrinthe seront stockés dans un tableau selon les règles définies par l'énoncé. Ce dernier s'arrêtera dès qu'un chemin aura été trouvé entre l'entrée et la sortie du labyrinthe.

Journal de Bord

Génération du labyrinthe

20 décembre 2017

Pour commencer je vais créer 2 constantes : row et col qui auront pour valeur arbitraire 20, puis, au niveau de ma fonction main, un tableau d'entiers de row par col.

Je vais créer une fonction disp qui permetra l'affichage du tableau tel que décrit dans l'énoncé ; à savoir : « capable d'afficher un labyrinthe au format ASCII (les murs sont des étoiles) en fonction d'un tableau à deux dimensions contenant des 0 et des 1 (la bordure du labyrinthe n'est pas contenue dans le tableau). »
Problème : la taille du tableau n'est pas connue à l'avance. Cependant on connaît ses dimensions maximales. On peut imaginer que l'utilisateur peut donner les dimensions qu'il souhaite dans la limite de row et col.

5 janvier 2018

Après un petit moment laissé de côté, je me remets sur ce projet. Pour la taille non connue des tableaux, je l'ai finalement construit par un malloc(et ai pensé à libérer la mémoire). J'ai construit un premier jet de génération de labyrinthe. J'ai donc commencé à créer ma fonction en lui passant comme argument les tailles du labyrinthe définies par l'utilisateur ainsi que la référence audit tableau. Son constructeur : generer(int li, int co, int tab[li][co]).
J'ai mis toutes les valeurs à 1 à l'aléatoire de créer un parcours (tant qu'on n'est pas à la derniere case, on monte, descend, avance, recule selon un rand et les bords du tableau.

Le problème de cet algo est qu'il a tendance à vider le labyrinthe. En effet, sachant que nous devons parcourir le chemin le plus long possible d'un bord à l'autre du tableau, Les chances d'y arriver directement diminuent au fur et à mesure de l'augmentation de la taille. Complexité de l'ordre exponentielle. L'idéal pour ce labyrinte serait une complexité linéaire.
Le code présenté ci dessous présente le fonctionnement de ce premier jet

```
while (!(i==li-1 && j==co-1))
{
	int rnd=rand()%4;
	tab[i][j]=0;
	switch(rnd)
	{
	   case 0:
	      if (i<li-1)
	      tab[++i][j]=0;
	      break;
	   case 1:
	      if(j<co-1)
	         tab[i][++j]=0;
	      break;
	   case 2:
	      if(i>0)
	         tab[--i][j]=0;
	      break;
	   case 3:
	      if(j>0)
	         tab[i][--j]=0;
	      break;
	   default:
```

```
            break;
        }
}
```

Le déroulement de la fonction précédente m'amène à voir le problème différemment. Je vais m'inspirer de ce premier jet en prenant en compte les cellules déjà modifiées.
Ce dernier devrait ressembler à celui présenté ci-dessous :

```
for(int i=0 ;i<li;i++) //pr chaque cellule
{
        for(int j=0;j<co;j++)
        {
                tab[i][j]=(i%2 || j%2) ; // cellules ayant des coordonnées paires =0, les autres 1

        }
}
```

Ainsi on obtient un total de (co/2+co%2)(li/2+li%2) cellules ayant pour valeur 0

12 janvier 2018

Je suis parti sur une autre base et ai repensé tout le générateur de labyrinthe. J'ai imaginé traviller avec les cellules adjascentes des cellules adjascnetes considérées. J'ai codé une fonction qui me renvoie pour chaque cellules adjascentes à la cellule considérée, en fonction des valeurs de ces propres cellules adjascentes si elle est exploitable ou pas. Il va falloir que je retravaille dessus, j'ai encore quelques soucis de sortie de tableau…

A	B	C	D	E	
1					
2					
3					
4					

En jaune clair et moyen : les cellules adjascentes à la cellule initiale (en noire)
En jaune moyen et foncé : les cellules adjascentes à une des cellule adjacente à la cellule noire.
Ainsi on peut aller en D2 si :
- D2 n'est pas encore visitée
- si D1 && ((C1 et C2) ou (E1 et E2)) n'ont pas été visitées (donc sont encore des murs)
Et ainsi dessuite, pour chaque cellule en jaune clair/moyen

20 janvier 2018

Je travaille sur cet algo depuis un petit moment et notamment sur ma fonction trop complexe à debuguer (beaucoup de test logiques sur la présence des cellules adjascences au rang 2 en fonction de leur existance). L'aléatoire n'arrange pas les choses pour faire les différents tests.
Le labyrinthe est cependant généré à peu pres correctement.
Le principe est
- Je genere un tableau de i par j que je remplis de 1
- Je choisis une cellule au hasard que je passe à -1
- Je cherches les possiblités de passage en fonction des cellules adjascentes de chaque cellule adjascente existante de la cellule considérée. (C'est justement cette fonction que je veux simplifier) et code cet possibilités sur un entier entre 0 et 15
- Je décode cet entier, choisis aléatoirement un des chemins possible et stocke i et j (les indices) dans un tableau
- J'incrémente ou décrémente i ou j et passe la nouvelle cellule à -1 , puis on recommence
-Si la cellule n'a plus de nouveau chemins possible on la passe à 0 et on se rend à la cellule stockée dans le tableau indiqué précédement.

-On s’arrete donc quand tout le tableau est vide (c’est à dire qu’il n’y a pplus la possiblité de créer de noveaux chemins
- Apres quoi on relie si ce n’est pas fait la cellule [0,0] au labyrinthe créé et de meme pour la cellule [li-1, co-1]

Les différentes fonctions que j’ai créé pour réaliser ces étapes sont expliquées en commentaire dans le code.

J’ai beau retourner ce probleme de simplification de l’adjascence dans tous les sens, il me parrait pas si évident à résoudre. L’année avance et je n’ai pas spécialement l’envie ni le temps de refondre completement mon projet.

27 janvier 2018

Je vais laisser le générateur en l’état pour un moment (vu qu’il marche mais que c’est juste la complexité du labyrinthe généré qui ne me plais pas outre mesure) pour attaquer l’automate qui va le parcourir. Aussi je transmet mon rapport de projet (ou plutot journal de bord de projet).

Réalisation de l’automate

25 mars 2018

Je me remets dans le projet. Avec le recul je corrige quelques fonctions.
Je me rends compte que ma fonction « suivant » recoit trop de parametres inutiles. Je fais le ménage
Pour le solveur de labyrinthe, je vais dans un premier temps tenter une adaptation de mon code de génération.
En effet, en modifiant ma fonction « adjascent », je pense pouvoir résoudre le labyrinthe.
Initialement, cette fonction génère aléatoirement une case de départ et va creuser le labyrinthe selon les résultats de la fonction « ch_possible » (fonction qui, comme je l’ai dit précédemment est à corriger). L’idée va être de shunter l’appel à cette fonction et de supprimer les paramètres d’aléatoire que j’y avais inclus pour la génération. Le principe de codage des déplacement restera le même (à priori)

24 avril 2018
J’ai un peu de vacances et vais enfin pouvoir avancer sur ce projet qui traine depuis quelques temps…
Depuis le dernier rapport sur ce journal, j’ai monté une première ébauche d’une fonction « sortir1 ».
Comme je l’avais précisé, elle est largement inspirée de « adjascent » l’appel à la fonction « ch_possible » a été remplacé par un groupe de tests conditionnels sur les valeurs des cases suivantes si elles existent. Pour pouvoir revenir en arriere (mémoire des positions précédentes) j’ai modifier la valeur de associée à la cellule (de 0 à 8) afin de pouvoir distinguer les cellules visitées totalement des cellules où il reste des chemins exploitables.

30 avril 2018

J’ai bien avancé cette semaine et juge ma fonction « sortir1 » prête
Maintenant on peut suivre le tracé du parcours ! J’ai d’ailleurs appliqué la même chose à la création du labyrinthe

Voici donc la fonction soritr :

```
sortir1(int li, int co,int **tab)
{
   int i=0,j=0;
   int cpt=(li*co);
   int *chem=malloc(cpt*2*sizeof(int));
   int ichem=0; //indice pout chem[]
   while (!(i==li-1 && j==co-1))
   {
      int nbadj=0;
      tab[i][j]=-1; //la cellule corrante passe en statut visitée
      int adj=0;
      if (i+1<li && tab[i+1][j]==0)
         adj+=1;
      if (j+1<co && tab[i][j+1]==0)
         adj+=2;
      if (i-1>=0 && tab[i-1][j]==0)
         adj+=4;
      if (j-1>=0 && tab[i][j-1]==0)
         adj+=8;

      int tmp[3]={0,0,0};//tmp représente le tableau des valeurs possibles
```

```
        chem[ichem]=i; //récupération de la position de i
        chem[++ichem]=j; //récupération de la position de j
        ++ichem;

        // decodage adj
        if(adj-8>=0)
        {
           tmp[nbadj++]=8;
           adj-=8;
        }
        if(adj-4>=0)
        {
           tmp[nbadj++]=4;
           adj-=4;
        }
        if(adj-2>=0)
        {
           tmp[nbadj++]=2;
           adj-=2;
        }
        if(adj-1>=0)
        {
           tmp[nbadj++]=1;
           adj-=1;
        }
        if(nbadj>0) //si des chemins n'ont pas été totalement explorés
        {
           //si il ne reste qu'une cellule possible, on défini directement que tous les parcours sont
explorés
     //adj permet de récupérer la position atteignable(1+2+4+8)
           suivant(tmp[nbadj-1],&i,&j);
        }
        else
        {
           tab[i][j]=8; //si on ne peut plus visiter on fout la case à 8
           if (ichem>3)// si on est bloqué, on remonte la file
           {
              ichem--;
              j=chem[ichem-=2];
              i=chem[--ichem];
           }
           else ichem=-1;
        }
        usleep(30000);
        printf("\e[1;1H\e[2J"); //effacer l'ecran
        disp(li,co,tab); //bien penser à gerer le 8 comme " " pour que l'affichage des étapes
intermédiaires soit correct
     }
     tab[i][j]=-1; // la derniere valeur est effectivement une cellule faisant partie du chemin
     //pour garder un tableau de sortie conforme, on remplace toutes les valeurs à 8 par 0
     for (int a =0;a<li;a++)
        for(int b=0;b<co;b++)
           if(tab[a][b]==8)
              tab[a][b]=0;
     printf("\e[1;1H\e[2J");
     printf("Un chemin a été trouvé !\n");
     disp(li,co,tab);
```

```
    sleep(10);
}
```

Dans un premier temps, je n'avais pas créé la valeur « 8 » pour le cas du retour arriere et avais utilisé -1. Or cela complexifiait inutilement le chemin final de la résolution. Alors, j'ai eu plusieurs choix qui se sont posés : rester au plus proche de l'énoncé et créer une bete copie de mon tableau où je remplace les valeurs 8 par des 0, soit rajouter une condition dans ma fonction d'affichage : interpréter les 8 comme des espaces. Ce que j'ai choisi.
Ainsi, les valeurs qui seront à 8 apparaitront comme des « », soit des passages ne faisant pas parti du chemin.

2 mai 2018

Ces quelques jours m'ont permis de quasiment finaliser le projet.
J'ai revu ma fonction sortir1 et ai créé une fonction « sortir2 » utilisant sensiblement le meme fonctionnement, mais dans laquelle j'implémente le tableau des déplacement tel qu'indiqué dans l'énoncé (ce qui n'était pas fait pour la version 1).
J'ai ajouté la sauvegarde et l'ouverture de fichiers labyrinthe ainsi qu'un menu permettant de choisi les différentes options.
Du coup, je vérifie aussi (très sommairement) la validité du labyrinthe.
J'ai également grandement commenté le code, histoire de s'y retrouver un minimum. Entre le tableau du labyrinthe, celui qui permet de retrouver les valeurs précédentes de i et j, le tableau des déplacements….

J'ai tenté de créer une 3e version où j'ai voulu supprimer chem et ichem ainsi que mon tableau tmp des chemins possibles, cependant avec la journée de codage dans les pattes, j'ai eu des soucis pour récupérer les anciennes valeurs de i et j dans le cas du retour arriere. Peut etre j'aurais dû me servir du tableau des deplacements… Quoi qu'il en soit, j'ai supprimé cette version.

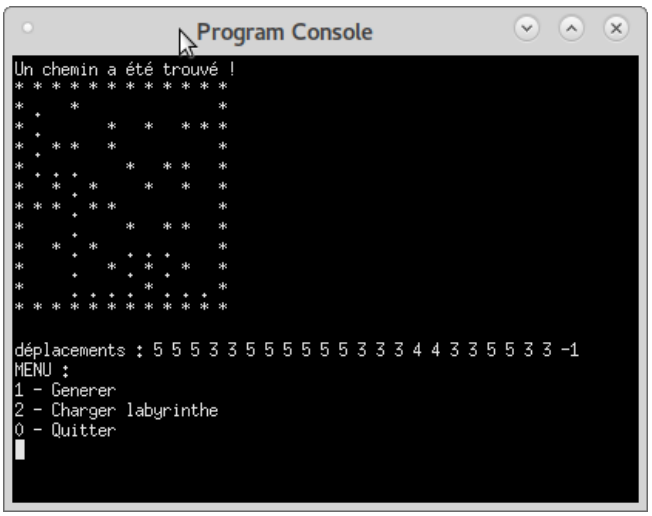
Je vais maintenant créer differents jeux de priorités pour les différentes directions possibles.
L'utilisateur définira ainsi si il souhaite utiliser un jeu de déplacments favorisant la découverte des cellules (long) ou si l'ordre des priorités de déplacement sera optimisé.
Nous savons que l'entrée le trouve dans l'angle supérieur droit et la sortie dans la'angle inférieur gauche. Alors pour optimiser les prioorités, on va choisir d'aller vers le bas, puis vers la droite, puis vers le haut et enfin vers la gauche.
Le chemin favorisant la découverte des cellules sera quant à lui réalisé avec les mêmes priorités dans l'ordre inverse.

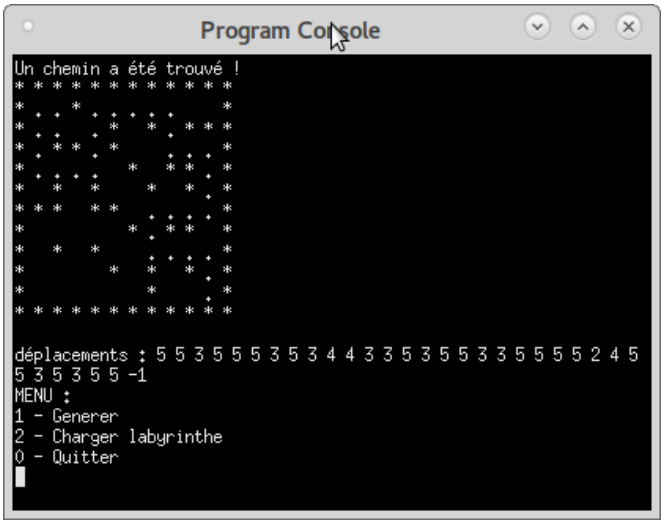
Afin de pouvoir tester le solveur (automate) avec des labyrinthes complexes, particuliers.. J'ai créé un menu permettant de choisir si l'utilisateur souhaite générer un labyrinthe ou en charger un stocké dans un fichier. Si l'utilisateur génère, celui-ci pourra enregistrer le labyrinthe généré. Il sera créé un fichier dans lequel la premiere ligne recevra les dimensions du tableau et les lignes suivantes contiendront le labyrinthe sous forme de 1 et de 0

5 mai 2018

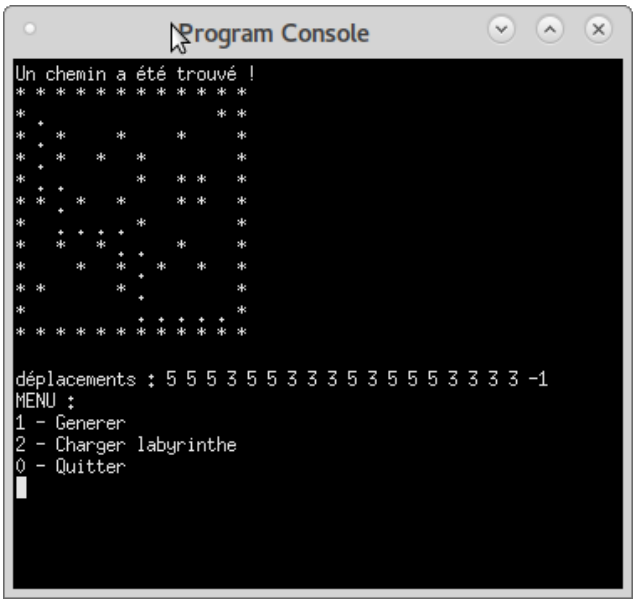
Enfin nous y sommes. Je finalise ce rapport et vais proposer des labyrinthes de test dans lesquels je vais comparer les résultats entre les 2 versions du solveur. Je vais faire un jeu de 4 labyrinthes et comparer les solutions trouvées.
Laby1 et Laby2 seront générés et exécutés
Laby3 sera un labyrinthe créé à la main qui contiendra des « pièges » (des murs circulaires, des faux chemins…)
Pour rester conforme à l'énoncé, ces tableaux seront de dimension 10*10 mais auraient très bien pu être rectangulaires (étant donné que mon code le permet)

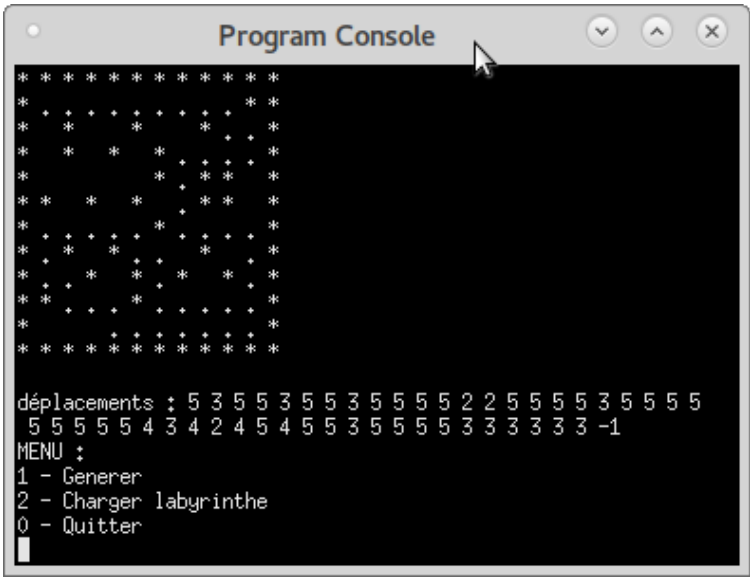
Laby1





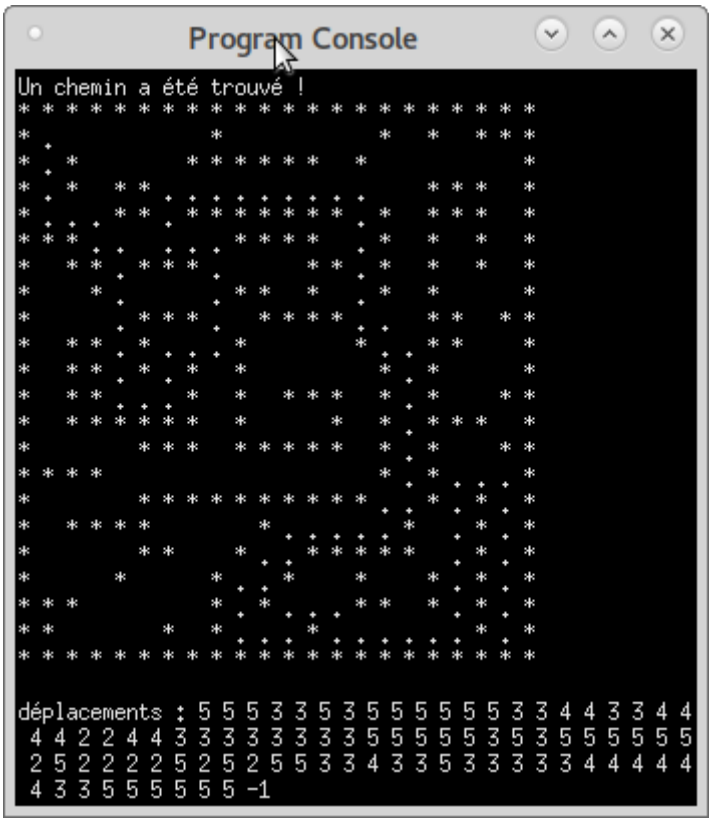
A gauche, le parcours rapide et à droite le parcours lent. On remarque clairement que le fait de favoriser la direction générale facilite grandement la résolution. Il en est de même avec Laby2 :

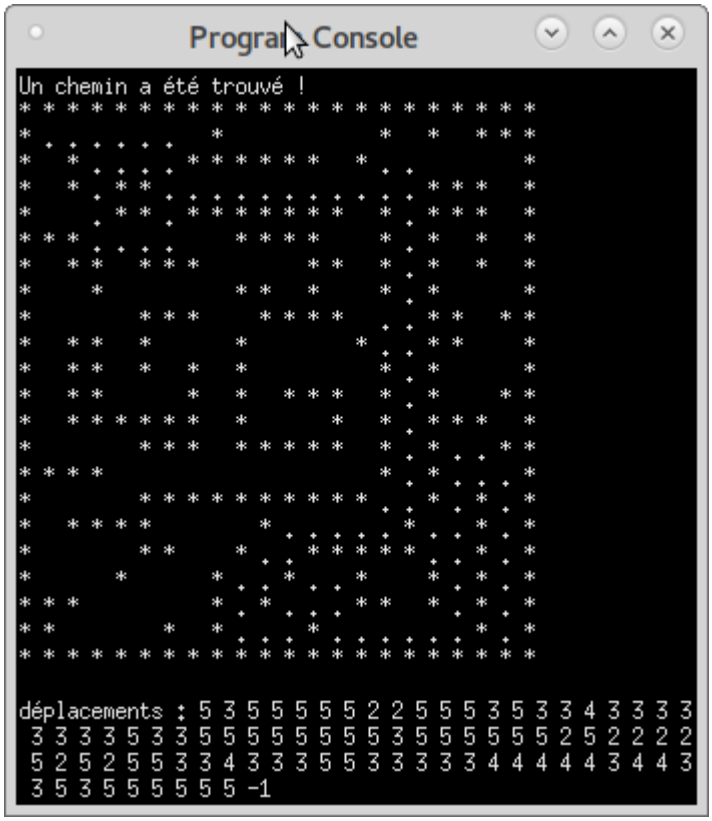




Pour Laby3, j'ai créé un labyrynthe de 20 par 20 reprenant les principaux pieges possibles : il y a des murs circulaires, des chemins larges, des retours arrieres, une obligation de passer au moins une fois dans chaque direction pour trouver la sortie...
Nous pouvons observer à l'exécution le parcours des 2 programmes. Le premier explore beaucoup plus (donc plus long pour trouver la sortie) cependant les résultats sont sensiblements identiques quant au parcours trouvé (dû au fait que j'ai volontairement créé un chemin unique sur la 2eme partie du labyrinthe ; la premiere comprenant des boucles et un « escargot ».

Ci-dessous, les exécutions, d'abord, la version « rapide », qui pour le coup est plus lente d'un point de vue temporel dans ce labyrinthe précis, puis la version « lente ».





Pour le 4eme, j'ai utilisé un générateur de labyrinthe parfait trouvé sur internet et ai créé mon fichier à partir de ce dernier. Il s'agit d'un labyrinthe de 59 lignes sur 99 colonnes.

Pour le coup, le parcours en profondeur est très profond !
J'ai bien envi de rajouter un tableau cases parcourues afin de me rendre réellement compte des différences.
Mais nous avons vu que chacune de ces version peut s'avérer plus efficace, donc meme si dans la majorité des cas la version rapide permet d'aboutir plus rapidement, dans certains cas il peut etre judicieux de modifier la priorité attibuée à chaque déplacement.
L'aléatoire est à exclure pour ce type de recherche de parcours car impliquerait une perte de temps énorme.
On sait d'où l'on vient, on sait où l'on va, donc l'idéal est de suivra la direction en général…(même si ici nous traitons un labyrinthe…)

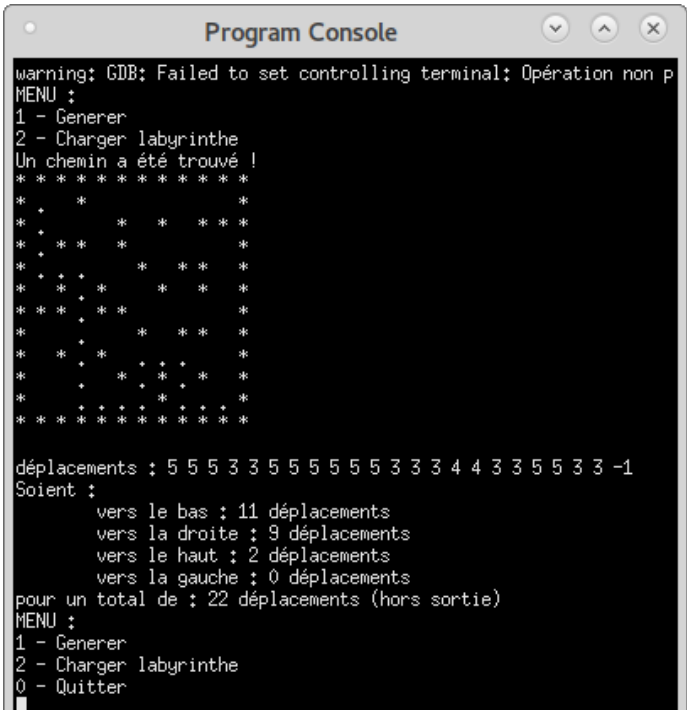
Je ne présenterai pas de copie écran de ce labyrinthe (elle serait illisible, cependant la résolution se déroule parfaitement dans les 2 cas).

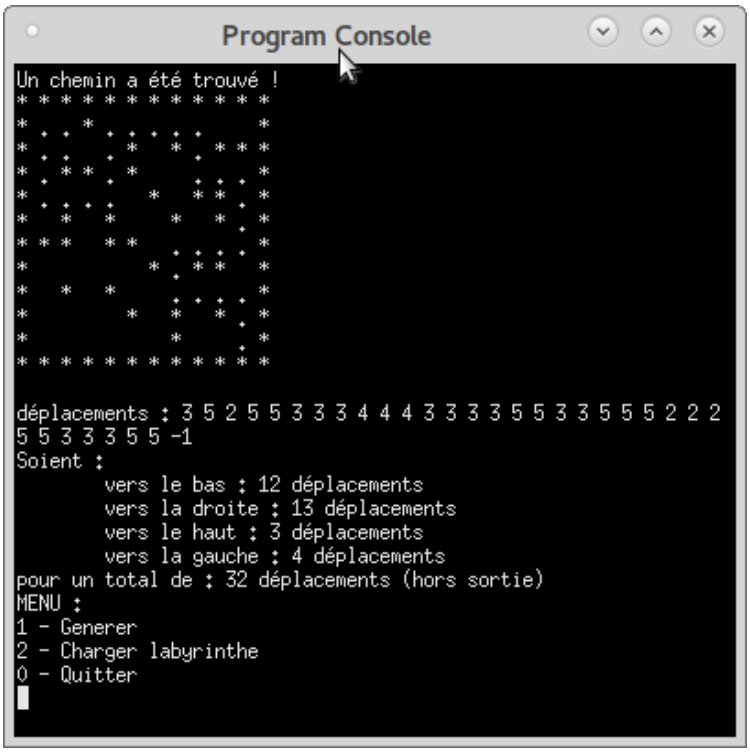
	Déplacements bas (5)	Déplacements doite (3)	Déplacements haut(4)	Déplacements gauche(2)	Total
Laby1, rapide	11	9	2	0	22 + 1(sortie)
Laby1, profond	18	10	3	1	32 + 1(sortie)
Laby2, rapide	9	9	0	0	18 + 1(sortie)
Laby2, profond	29	12	4	3	
Laby3, rapide					
Laby3, profond					
5b,3d,					

Et à ce stade, je m'apperçois d'une énormité dans mon code ! Ce 29 est totalement incohérent au vue du nombre de déplacemnt vers le haut. Cela signifierait que le tableau fait 26 de hauteur...

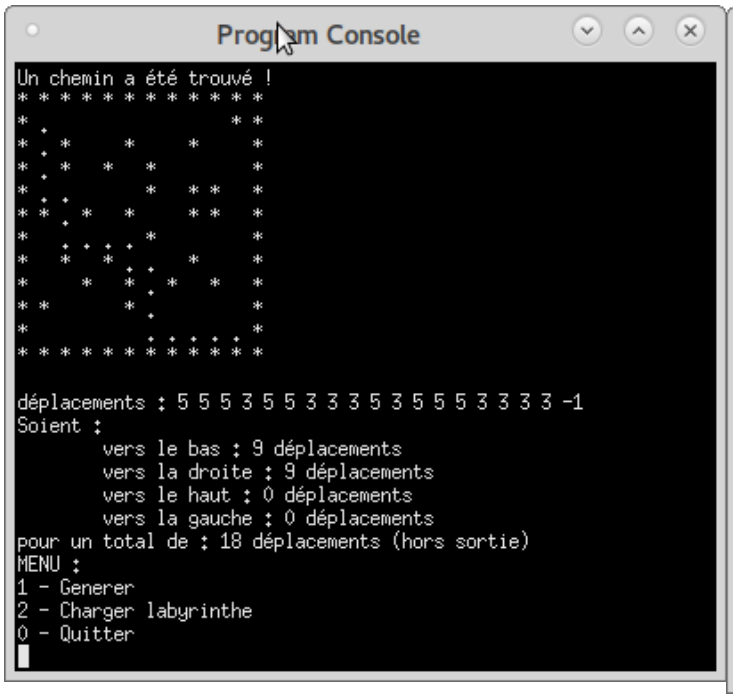
Mon code est réparé (une condition du switch/case posait souci sur la 2e forme de résolution pour l'implémentation du tableau des déplacements…). J'ai également ajouter le décompte par direction et vais donc présenter les grilles et le tableau avec les bonnes valeurs.

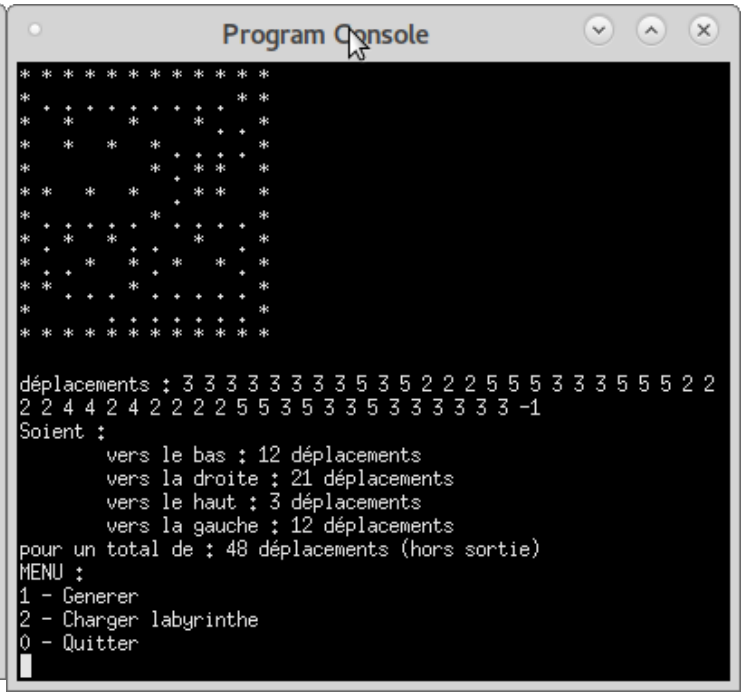
Laby1 :



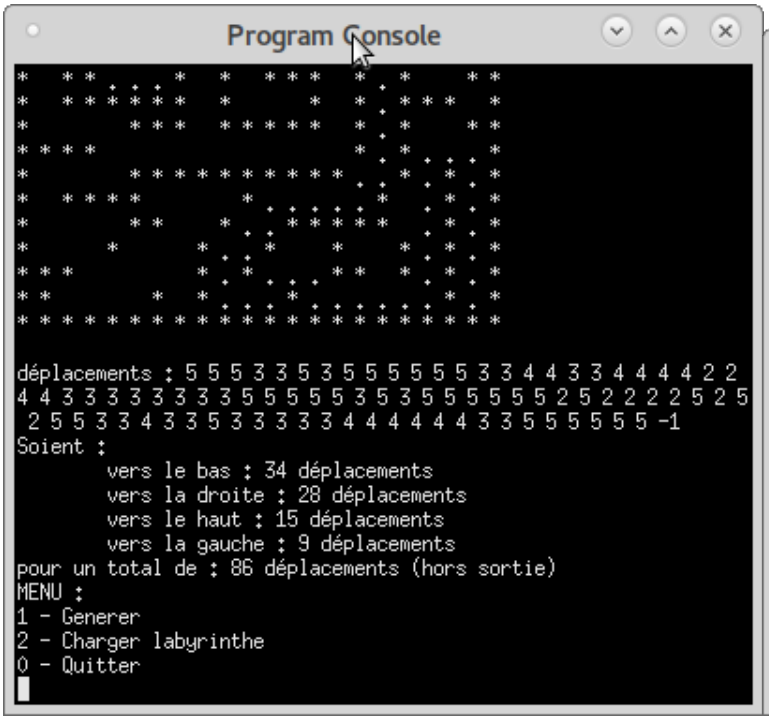


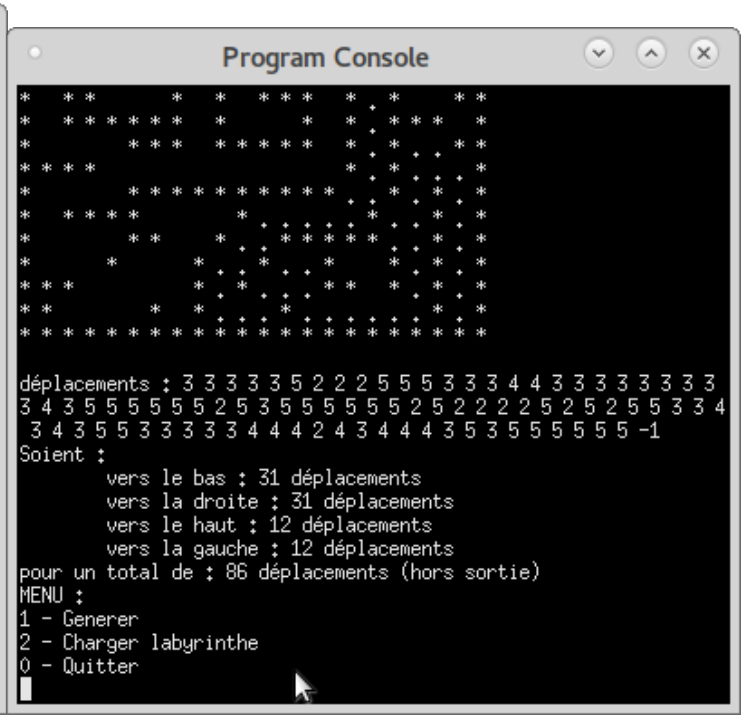
Laby2 :


Program Console
Un chemin a été trouvé !
déplacements : 5 5 5 3 3 5 5 3 3 3 5 3 5 5 5 3 3 3 3 -1
Soient :
vers le bas : 9 déplacements
vers la droite : 9 déplacements
vers le haut : 0 déplacements
vers la gauche : 0 déplacements
pour un total de : 18 déplacements (hors sortie)
MENU :
1 - Generer
2 - Charger labyrinthe
0 - Quitter


Program Console
déplacements : 3 3 3 3 3 3 3 3 5 3 5 2 2 2 5 5 5 3 3 3 5 5 5 2 2
2 2 4 2 4 2 2 2 2 5 5 3 5 3 3 5 3 3 3 3 3 3 -1
Soient :
vers le bas : 12 déplacements
vers la droite : 21 déplacements
vers le haut : 3 déplacements
vers la gauche : 12 déplacements
pour un total de : 48 déplacements (hors sortie)
MENU :
1 - Generer
2 - Charger labyrinthe
0 - Quitter

Laby 3 :


Program Console
déplacements : 5 5 5 3 3 5 3 5 5 5 5 5 5 3 3 4 4 3 3 4 4 4 4 2 2
4 4 3 3 3 3 3 3 3 3 5 5 5 5 5 3 5 3 5 5 5 5 5 5 2 5 2 2 2 2 5 2 5
2 5 5 3 3 4 3 3 5 3 3 3 3 3 4 4 4 4 4 4 3 3 5 5 5 5 5 5 -1
Soient :
vers le bas : 34 déplacements
vers la droite : 28 déplacements
vers le haut : 15 déplacements
vers la gauche : 9 déplacements
pour un total de : 86 déplacements (hors sortie)
MENU :
1 - Generer
2 - Charger labyrinthe
0 - Quitter


Program Console
déplacements : 3 3 3 3 3 5 2 2 2 5 5 5 3 3 3 4 4 3 3 3 3 3 3 3 3
3 4 3 5 5 5 5 5 5 2 5 3 5 5 5 5 5 5 5 2 5 2 2 2 2 5 2 5 2 5 5 3 3 4
3 4 3 5 5 3 3 3 3 3 4 4 4 2 4 3 4 4 4 3 5 3 5 5 5 5 5 5 -1
Soient :
vers le bas : 31 déplacements
vers la droite : 31 déplacements
vers le haut : 12 déplacements
vers la gauche : 12 déplacements
pour un total de : 86 déplacements (hors sortie)
MENU :
1 - Generer
2 - Charger labyrinthe
0 - Quitter

	Déplacements bas (5)	Déplacements doite (3)	Déplacements haut(4)	Déplacements gauche(2)	Total
Laby1, rapide	11	9	2	0	22 + 1(sortie)
Laby1, profond	12	13	3	4	32 + 1(sortie)
Laby2, rapide	9	9	0	0	18 + 1(sortie)
Laby2, profond	12	21	3	12	48 + 1 (sortie)
Laby3, rapide	34	28	15	9	86 + 1 (sortie)
Laby3, profond	31	31	12	12	86 + 1 (sortie)
Laby4 *	152	180	94	82	508+ 1 (sortie)
Répartition rapide / profond (hors laby4)	54 / 55	46 / 65	17 / 18	9 / 28	126 / 166

Il est évident que la hauteur du tableau fait toujours le nombre de déplacement vers le bas -nb déplacement vers le haut +1, et idem pour la longueur. C’est d’ailleurs une des propriétés qui m’a permis de m’apercevoir de mon erreur précédente.
Etant donné que Laby4 est un labyrinthe parfait, quelqeusoit le parcours réalisé dessus, le chemin déterminé sera unique. Aussi je ne le prends pas en compte dans les totaux.

Conclusion

Ce projet a été très intéressant à réaliser à différents niveaux :

Il m’a permis de réfléchir à différentes manières de générer un labyrinthe, en creusant, en murant, en agissant de minière aléatoire, linéaire (ou colonnaire…), adjacente... et à réfléchir aux différentes manières de les coder. J’ai d’emblée éliminé les versions linéaires car bien que paraissant les plus simple à mettre en place, me déplaisaient dans ma conception d’un labyrinthe. Aussi, je me suis concentré sur l’adjacence où il me semble qu’il y a de la matière à réfléchir. Avec le recul, peut-être un peu trop. En effet, les labyrinthes générés par mon appli fonctionnent mais restent éloignés de ce que je souhaitais obtenir.

Il a été intéressant de travailler par étape. Cela a permis de dégrossir le projet et de prendre de temps de se concentrer sur chaque partie. Cependant, de nombreuses notions n’ont été abordées que tardivement dans l’année (notamment les algorithmes sur les graphes). D’ailleurs, au moment où j’ai commencé à me pencher sur ce projet, la notion de graphes m’était encore inconnue.

Aussi, à refaire, il y a des chances que mon code s’en retrouve grandement modifié. En effet j’utiliserai certainement un structure de tableau à 2 dimensions d’objets cellule ou quelque chose qui s’en approche et utiliserai des algorithmes issus de la théorie des graphes.

Annexe : Code-source

(j’ai volontairement laissé la fonciton sortir1, bien qu’elle ne serve plus)

J’ai tenté de séparer mes fonction sur différentes pages

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

//Pour pouvoir sauvegarder le labyrinthe généré dans un fichier
void save_laby(int li, int co, int **tab)
{
    printf("Entrez un nom de fichier pour votre laby dans rep courrant ou entrez  un chemin
complet+nomde fichier\n");
    char chem[150];
    scanf("%s",&chem);
    printf("%c,chem");
    FILE *flaby=fopen(chem,"w"); //On va créer le fichier avec le nom saisi et l'ouvrir
    fprintf(flaby,"%d %d\n",li,co); //pour pas s'embeter, on place à la premiere ligne la valeur de li
et de co

    for(int i=0;i<li;i++) //ensuite chaque cellule du tableau va être recopiée dans le fichier
    {
        for(int j=0;j<co;j++)
            fprintf(flaby,"%d ",tab[i][j]);
        fprintf(flaby,"\n"); //ligne suivante
    }
    fclose(flaby); // on ferme le fichier
}


//Fonction qui va gérer l'affichage
void disp(int i, int j, int **tab)
{
    for (int l=0;l<=j+1;l++) //bordure haute du cadre
        printf ("* ");
    printf("\n");
    for(int k=0;k<i;k++)
    {
        printf ("* "); //bordure gauche du cadre
        for (int l=0;l<j;l++)
        {
        if (tab[k][l]==0 ||tab[k][l]==8){printf("  ");} // la valeur 8 a été rajoutée par rapport à la
fonction sortir1
        else if (tab[k][l]==-1){printf (". ");}//on affiche un espace ou * en fonction de la valeur
contenue à tab[k][l]
        else if (tab[k][l]==1){printf ("* ");}//on affiche un espace ou * en fonction de la valeur
contenue à tab[k][l]

        //printf("%d",tab[k][l]);
        //(tab[k][l]?printf("* ") :printf ("  "));//on affiche un espace ou * en fonction de la valeur
contenue à tab[k][l]
        }
        printf ("*\n"); //bordure droite du cadre
    }
    for (int l=0;l<=j+1;l++) // bordure basse du cadre
        printf ("* ");
    printf ("\n\n");
}

//passer à la cellule suivante du laby selon la valeur de x
suivant(int x,int *i, int *j)
{
    switch (x)
    {
        case 1:++(*i);break;
        case 2:++(*j);break;
        case 4:--(*i);break;
        case 8:--(*j);break;
    }
}
```

```
int ch_poss(int i,int j, int li, int co, int **tab) //recherche des déplacements possibles
{
        printf("\e[1;1H\e[2J");
        disp(li,co,tab);
        int ch=0;//il y a max 3 cases adjascentes possibles (on évite de faire des aller retours
inutiles on ne revient pas sur case visitée... ou pas par là)

//on desccend si :
//la casse existe i<li-1
//pas encore visitee (ou pas finie) !=0

    //BAS
    if (i<li-1 && tab[i+1][j]==1)
    {
    //cellules adjascentes à la destination hors bords
        if(i<li-2 && j<co-1 && j>0) //cas general
        {
                if(!(tab[i+1][j-1]<1 && ((tab[i][j-1]<1 &&tab[i][j]<1) || (tab[i+2][j-1]<1 && tab[i+2]
[j]<1)))) //groupement par 3 -> pas possible
                    if (!(tab[i+1][j+1]<1 && ((tab[i][j+1]<1 &&tab[i][j]<1) || (tab[i+2][j+1]<1 &&
tab[i+2][j]<1))))
                        ch+=1;
            //memes conditions si cell adj on reprend les memes conditions avec les valeurs hors
tableau à 1;
        }
        else if(i<li-2 && j==0) //bord gauche
        {
            if(!(tab[i+2][j]<1)) //groupement par 3 -> pas possible
                if (!(tab[i+1][j+1]<1 && (tab[i][j+1]<1  || (tab[i+2][j+1]<1 && tab[i+2][j]<1))))
                    ch+=1;
        }
        else if(i<li-2 && j==co-1) //bord droit
        {
            if(!(tab[i+1][j-1]<1 && tab[i][j-1]<1)) //groupement par 3 -> pas possible
                if (!(tab[i+2][j]<1))
                    ch+=1;
        }

        else if(i==li-2 && j<co-1 && j>0) //bord bas
        {
            if(!(tab[i+1][j-1]<1 && tab[i][j-1]<1 ) || !(tab[i+1][j+1]<1 && tab[i][j+1]<1 &&tab[i]
[j]<1))// &roupement par 3 -> pas possible
                ch+=1;
        }
        //bord haut inutile : on descend
        //angles
        else if(j==0 && i==li-2) //angle gauche
            {
                if (!(tab[i+1][j+1]>0 && tab[i][j+1]<0 ))
                    ch+=1;
            }
        else if(j==co-1 && i==li-2) // angle bas droite ok
        {
            if (!(tab[i+1][j-1]>0 && tab[i][j-1]<0 ))
                ch+=1;
        }
    } //FIN SI BAS


    //DROITE
    if (j<co-1 && tab[i][j+1]==1)
    {
        if(j<co-2 && i<li-1 && i>0) //cas general
        {
            if(!(tab[i-1][j+1]<1 && ((tab[i-1][j]<1 &&tab[i][j]<1) || (tab[i-1][j+2]<1 && tab[i]
[j+2]<1)))) //groupement par 3 -> pas possible
                if (!(tab[i+1][j+1]<1 && ((tab[i+1][j]<1 &&tab[i][j]<1) || (tab[i+1][j+2]<1 && tab[i]
[j+2]<1))))
                    ch+=2;
            //memes conditions si cell adj on reprend les memes conditions avec les valeurs hors
tableau à 1;
        }
```

```
        else if(j<co-2 && i==0) //bord gauche
        {
            if(!(tab[i][j+2]<1))//groupement par 3 -> pas possible
                if (!(tab[i+1][j+1]<1 && (tab[i+1][j]<1 || (tab[i+1][j+2]<1 && tab[i][j+2]<1))))
                    ch+=2;
        }
        else if(j<co-2 && i==li-1) //bord droit
        {
            if(!(tab[i-1][j+1]<1 && tab[i-1][j]<1 )) //groupement par 3 -> pas possible
                if (!(tab[i][j+2]<1))
                    ch+=2;
        }

        else if(j==co-2 && i<li-1 && i>0) //bord bas
        {
            if(!(tab[i-1][j+1]<1 && tab[i-1][j]<1 ) || !(tab[i+1][j+1]<1 && tab[i+1][j]<1))//
&roupement par 3 -> pas possible
                ch+=2;
        }
        //bord haut inutile : on descend
        //angles
        else if(i==0 && j==co-2)
        {
            if (!(tab[i+1][j+1]<1 && tab[i+1][j]<1 ))
                ch+=2;
        }
        else if(j==co-2 && i==li-1) //angle droit OK
        {
            if (!(tab[i-1][j+1]<1 && tab[i-1][j]<1))
                ch+=2;
        }
    } //FIN SI DROITE

    //HAUT
    if (i>0 && tab[i-1][j]==1)
    {
    //cellules adjascentes à la destination hors bords
        if(i>1 && j<co-1 && j>0) //cas general
        {
            if(!(tab[i-1][j-1]<1 && ((tab[i][j-1]<1 &&tab[i][j]<1) || (tab[i-2][j-1]<1 && tab[i-2]
[j]<1)))) //groupement par 3 -> pas possible
                if (!(tab[i-1][j+1]<1 && ((tab[i][j+1]<1 &&tab[i][j]<1) || (tab[i-2][j+1]<1 && tab[i-2]
[j]<1))))
                    ch+=4;
            //memes conditions si cell adj on reprend les memes conditions avec les valeurs hors
tableau à 1;
        }
        else if(i>1 && j==0) //bord gauche
        {
            if(!(tab[i-2][j]<1))//groupement par 3 -> pas possible
                if (!(tab[i-1][j+1]<1 && (tab[i][j+1]<1  || (tab[i-2][j+1]<1 && tab[i-2][j]<1))))
                    ch+=4;
        }
        else if(i>1 && j==co-1) //bord droit //OK
        {
            if(!(tab[i-1][j-1]<1 && tab[i][j-1]<1 && tab[i][j]<1)) //groupement par 3 -> pas possible
                if (!(tab[i-2][j]<1))
                    ch+=4;
        }
        else if(i==1 && j<co-1 && j>0) //bord haut
        {
            if(!(tab[i-1][j-1]<1 && tab[i][j-1]<1 &&tab[i][j]<1) || !(tab[i-1][j+1]<1 && tab[i][j+1]<1
&&tab[i][j]<1))// &roupement par 3 -> pas possible
                ch+=4;
        }
        //bord haut inutile : on descend
        //angles
        else if(j==0 && i==1)
            {
                if (!(tab[i-1][j+1]<1 && tab[i][j+1]<1 ))
                    ch+=4;
            }
```

```
        else if(j==co-1 && i==1) //angle SUP droit OK
        {
            if (!(tab[i-1][j-1]<1 && tab[i][j-1]<1 ))
                ch+=4;
        }
    } //FIN SI HAUT

//GAUCHE
    if (j>0 && tab[i][j-1]==1)
    {
        if(j>1 && i<li-1 && i>0) //cas general //ok
        {
            if(!(tab[i-1][j-1]<1 && ((tab[i-1][j]<1 &&tab[i][j]<1) || (tab[i-1][j-2]<1 && tab[i]
[j-2]<1)))) //groupement par 3 -> pas possible
                if (!(tab[i+1][j-1]<1 && ((tab[i+1][j]<1) || (tab[i+1][j-2]<1 && tab[i][j-2]<1))))
                    ch+=8;
        //memes conditions si cell adj on reprend les memes conditions avec les valeurs hors tableau à
1;
        }
        else if(j>1 && i==0) //bord HAUT
        {
          if(!(tab[i+1][j-1]<1 && tab[i+1][j]<1))//groupement par 3 -> pas possible
                 //   if (!(tab[i+1][j]<1))
                        ch+=8;
        }
        else if(j>1 && i==li-1) //bord bas OK
        {
            if(!(tab[i-1][j-1]<1 && tab[i-1][j]<1 )) //groupement par 3 -> pas possible
                if (!(tab[i][j-2]<1))
                    ch+=8;
        }

        else if(j==1 && i<li-1 && i>0) //bord gauche
        {    if(!( tab[i-1][j]<1  || (tab[i+1][j-1]<1 && tab[i+1][j]<1)))// &roupement par 3 -> pas
possible
                        ch+=8;
        }
        //bord haut inutile : on descend

        //angles A VERIF
        else if(i==0 && j==1)
        {
            if (!(tab[i+1][j-1]<1 && tab[i+1][j]<1 ))
                ch+=8;
        }
        else if(j==1 && i==li-1) //angle bas gauche ok
        {
            if (!(tab[i-1][j-1]<1 && tab[i-1][j]<1 ))
                ch+=8;
        }
    } //FIN SI GAUCHE
    return (ch); //si ch=0 la case suivante soit n'existe pas soit totalement visitée
}

//Initialisation du tableau
int** init(int li,int co)
{
    int **tab=NULL;
    tab=(int**)malloc(sizeof(int*)*li); //allocation espace mem pour stocker li pointeurs sur tableau
d'entiers
    for(int cpt=0;cpt<li;cpt++)
    {
        tab[cpt]=(int*)malloc(sizeof(int)*co); //allocation d'espace memoire pour stocker co entiers
        for (int it2=0;it2<co;it2++)
            tab[cpt][it2]=1;
    }
    return (tab);
}
```

```c
//etude des cellules adjacentes à la cellule courante
void adjacent(int li,int co,int **tab)
{
    srand(time(NULL));
    int i=rand()%li,j=rand()%co;
    int cpt=(li*co);
    int *chem=malloc(cpt*2*sizeof(int));
    int ichem=0; //indice pout chem[]
    //cell non visitées /mur : 1, cell partiellement tritée : -1, cell traitée : 0
    while ( ichem>=0)
    {
        int nbadj=0;
        tab[i][j]=-1; //la cellule corrante passe en statut visitée
        usleep(10000);//30000
        int adj=ch_poss(i,j,li,co,tab); //on cherche les directions possibles et stock dans adj
        int tmp[3]={0,0,0};//tmp représente le tableau des valeurs possibles
        chem[ichem]=i; //récupération de la position de i
        chem[++ichem]=j; //récupération de la position de j
        ++ichem;
        // decodage adj
        if(adj-8>=0)
        {
            tmp[nbadj++]=8;
            //++nbadj;
            adj-=8;
        }
        if(adj-4>=0)
        {
            tmp[nbadj++]=4;
            //++nbadj;
            adj-=4;
        }
        if(adj-2>=0)
        {
            tmp[nbadj++]=2;
            //++nbadj;
            adj-=2;
        }
        if(adj-1>=0)
        {
            tmp[nbadj++]=1;
            //++nbadj;
            adj-=1;
        }
        if(nbadj>0) //si des chemins n'ont pas été totalement explorés
        {
            //si il ne reste qu'une cellule possible, on défini directement que tous les parcours sont
explorés
     //adj permet de récupérer la position atteignable(1+2+4+8)
            switch(rand()%nbadj)
            {
                case 0:suivant(tmp[0],&i,&j);break;//i+2
                case 1:suivant(tmp[1],&i,&j);break;//j+2
                case 2:suivant(tmp[2],&i,&j);break;//i-2
                case 3:suivant(tmp[3],&i,&j);break;//j-2
                default:break;
            }
        }
        else
        {
            tab[i][j]=0; //si on ne peut plus visiter on fout la case à 0

            if (ichem>3)// si on est bloqué, on remonte la file
            {
                ichem--;
                j=chem[ichem-=2];
                i=chem[--ichem];
            }
            else ichem=-1;
        }
    }
}
```

```c
int** generer(int li, int co) //, int tab[li][co]
{
    int ** tab=init(li,co); //init du tableau : toutes les cellules sont à 1
    disp(li,co,tab); //azffichage du tableau initialisé
    int i=0,j=0;
    int *chemin=malloc(i*j*sizeof(int));
   // creer_cell(li,co, &i,&j,tab,chemin);
    adjacent(li,co,tab);

    while (li > 0 && tab[li-1][co-1]!=0 )
        tab[(--li)][co-1]=0;
    j=0;
    while (tab[0][j]!=0 && co-j>0)
        tab[0][j++]=0;
    printf("\n\n");
    return(tab);
}


sortir1(int li, int co,int **tab)
{
    int i=0,j=0;
    int cpt=(li*co);
    int *chem=malloc(cpt*2*sizeof(int));
    int ichem=0; //indice pout chem[]
//cell non visitées /mur : 1
//cell partiellement tritée : -1
//cell traitée : 0
//mur : 1
    while (!(i==li-1 && j==co-1))
    {
        int nbadj=0;
        tab[i][j]=-1; //la cellule corrante passe en statut visitée


        int adj=0;//ch_poss(i,j,li,co,tab); //on cherche les directions possibles et stock dans adj

        if (i+1<li && tab[i+1][j]==0)
            adj+=1;
        if (j+1<co && tab[i][j+1]==0)
            adj+=2;
        if (i-1>=0 && tab[i-1][j]==0)
            adj+=4;
        if (j-1>=0 && tab[i][j-1]==0)
            adj+=8;

        int tmp[3]={0,0,0};//tmp représente le tableau des valeurs possibles
        chem[ichem]=i; //récupération de la position de i
        chem[++ichem]=j; //récupération de la position de j
        ++ichem;
        // decodage adj
        if(adj-8>=0)
        {
            tmp[nbadj++]=8;
            adj-=8;
        }
        if(adj-4>=0)
        {
            tmp[nbadj++]=4;
            adj-=4;
        }
        if(adj-2>=0)
        {
            tmp[nbadj++]=2;
            adj-=2;
        }
        if(adj-1>=0)
        {
            tmp[nbadj++]=1;
            adj-=1;
```

```
        }
        if(nbadj>0) //si des chemins n'ont pas été totalement explorés
        {
            //si il ne reste qu'une cellule possible, on défini directement que tous les parcours sont
explorés
     //adj permet de récupérer la position atteignable(1+2+4+8)
            suivant(tmp[nbadj-1],&i,&j);
        }
        else
        {
            tab[i][j]=8; //si on ne peut plus visiter on fout la case à 8
            if (ichem>3)// si on est bloqué, on remonte la file
            {
                ichem--;
                j=chem[ichem-=2];
                i=chem[--ichem];
            }
            else ichem=-1;
        }
        usleep(50000);
        printf("\e[1;1H\e[2J"); //effacer l'ecran
        disp(li,co,tab); //bien penser à gerer le 8 comme " " pour que l'affichage des étapes
intermédiaires soit correct
    }
    tab[i][j]=-1; // la derniere valeur est effectivement une cellule faisant partie du chemin
    //pour garder un tableau de sortie conforme, on remplace toutes les valeurs à 8 par 0
    for (int a =0;a<li;a++)
        for(int b=0;b<co;b++)
            if(tab[a][b]==8)
                tab[a][b]=0;
    printf("\e[1;1H\e[2J");
    printf("Un chemin a été trouvé !\n");
    disp(li,co,tab);
    sleep(3);
}

int menu()
{
    int choix;
    printf("MENU :\n");
    printf("1 - Generer\n"); //ret int**
    printf("2 - Charger labyrinthe\n"); //ret int **
    printf("0 - Quitter\n");
    int ret;
    scanf("%d",&ret);
    return ret;
}

//leberation de ma méoire allouée via malloc
liberer(int li,int**tab)
{
    for (int a=0;a<li;a++)
        free(tab[a]);
    free(tab);
}

//chargement du laby à partir d'un fichier
int ** load_laby(int *li,int *co)
{
    printf("Entrez un nom de fichier pour votre laby dans rep courrant ou entrez  un chemin
complet+nomde fichier\n");
    char chem[150];
    scanf("%s",&chem);
    FILE *flaby=fopen(chem,"r");
    //on recupere les valeurs de li et de co dans le fichier et on initialise le tableau
    fscanf(flaby,"%d %d",&(*li),&(*co));
    int ** tab =init(*li,*co);
    //puis on remplit le tableau avec les valeurs
    for (int i=0; i<*li;i++)
        for (int j= 0; j<*co;j++)
            fscanf(flaby, "%d", &tab[i][j]);
    fclose(flaby);
```

```c
    return (tab);
}

//solveur de laby
void sortir2(int li, int co,int **tab,int*deplacement,int regle)
{
    int i=0,j=0;
    int cpt=(li*co);
    int *chem=malloc(cpt*2*sizeof(int));
    int ichem=0; //indice pout chem[]
    int i_depl=0; //indice pour deplacement[]
    while (ichem>=0 && !(i==li-1 && j==co-1)) //tant qu'on n'a pas parcouru tout ce qui était
parcorable et que la sortie n'a pas été trouvée
    {
        int nbadj=0;
        tab[i][j]=-1; //la cellule corrante passe en statut visitée
        int adj=0;//on cherche les directions possibles et stock dans adj
        if (i+1<li && tab[i+1][j]==0)
            adj+=1;
        if (j+1<co && tab[i][j+1]==0)
            adj+=2;
        if (i-1>=0 && tab[i-1][j]==0)
            adj+=4;
        if (j-1>=0 && tab[i][j-1]==0)
            adj+=8;

        int tmp[3]={0,0,0};//tmp représente le tableau des valeurs possibles
        chem[ichem]=i; //récupération de la position de i
        chem[++ichem]=j; //récupération de la position de j
        ++ichem;

        // decodage adj
        if(adj-8>=0)
        {
            tmp[nbadj++]=8;
            adj-=8;
        }
        if(adj-4>=0)
        {
            tmp[nbadj++]=4;
            adj-=4;
        }
        if(adj-2>=0)
        {
            tmp[nbadj++]=2;
            adj-=2;
        }
        if(adj-1>=0)
        {
            tmp[nbadj++]=1;
            adj-=1;
        }
        if(nbadj>0) //si des chemins n'ont pas été totalement explorés
        {
            if (regle==0)
            {
                //si il ne reste qu'une cellule possible, on défini directement que tous les parcours
sont explorés
                //adj permet de récupérer la position atteignable(1+2+4+8)
                switch(tmp[nbadj-1])
                {
                    case 1:
                        deplacement[i_depl++]=5;
                        break;
                    case 2:
                        deplacement[i_depl++]=3;
                        break;
                    case 4:
                        deplacement[i_depl++]=4;
                        break;
                    case 8:
                        deplacement[i_depl++]=2;
```

```
                        break;
                }
                suivant(tmp[nbadj-1],&i,&j);
            }
            if (regle==1)
            {
                switch(tmp[0])
                {
                    case 1:
                        deplacement[i_depl++]=5;
                        break;
                    case 2:
                        deplacement[i_depl++]=3;
                        break;
                    case 4:
                        deplacement[i_depl++]=4;
                        break;
                    case 8:
                        deplacement[i_depl++]=2;
                        break;
                }
                suivant(tmp[0],&i,&j); // on récupere la derniere case du tableau. on favorise le
premier élément entré soit le déplacement vers le haut, puis vers la gauche, puis la droite, puis le
bas
            }
        }
        else
        {
            tab[i][j]=8; //si on ne peut plus visiter on fout la case à 8, valeur arbitraire >0 et
neutre
            if (ichem>3)// si on est bloqué, on remonte la file
            {
                ichem--;
                j=chem[ichem-=2];
                i=chem[--ichem];
                deplacement[--i_depl] =-1; //ON PENSE AU NOUVEAU TABLEAU DE DEPLACEMENTS...
            }
            else ichem=-1;
        }
        usleep(20000);

        printf("\e[1;1H\e[2J"); //effacer l'ecran
        disp(li,co,tab); //bien penser à gerer le 8 comme " " pour que l'affichage des étapes
intermédiaires soit correct
    }
    if (ichem<0)
    {
        printf("Ce labyrinthe semble avoir un souci...\n");
        sleep(2);
        exit (1);
    }
    tab[i][j]=-1; // la derniere valeur est effectivement une cellule faisant partie du chemin
    deplacement[i_depl]=-1; //il s'agit de la case de sortie
    //pour restituer le tableau initial tel qu'il a été confié, on remplace toutes les valeurs à 8 par
0
    for (int a =0;a<li;a++)
        for(int b=0;b<co;b++)
            if(tab[a][b]==8)
                tab[a][b]=0;
    printf("\e[1;1H\e[2J");
    printf("Un chemin a été trouvé !\n");
    disp(li,co,tab);
}

int main()
{
    int men=menu();
    int li,co;
    int ** tableau=NULL;
    while(men>0)
    {
        if(men==1) //cas 1 - générer
```

```
        {
            printf("nombre de ligne");
            scanf("%d", &li);
            printf("nombre de colonnes");
            scanf("%d", &co);
            printf("%d lignes %d col \n\n",li,co);
            tableau=generer(li,co);
            printf ("\e[1;1H\e[2J");
            printf ("Voici le labyrinthe généré...\n");
            disp(li,co,tableau); // appel de la fonction, permettant l'afichage du tableau
            printf ("sauvearder ? 1 - oui, 0 - non, 2 - Regenerer\n");
            int rep;
            scanf("%d",&rep);
            if(rep==1)
                save_laby(li,co,tableau);
            if(rep!=2)
                men=3;
        }
        if(men==2)    //cas 2 charger labyrinthe
        {
            tableau=load_laby(&li,&co);
            printf("Votre labyrinthe de %dlignes par %d colonnes est chargé chargement ...\n",li,co);
            printf("\nVoici votre labyrinthe :\n");
            disp(li,co,tableau);
            men=3;
        }
        if (men==3)    //cas 3 résoudre laby
        {
            printf ("Résolution  du labyrinthe...\n");
            int etat[li*co]; //tableau pour stocker les deplacements
            int regle=0;
            printf("Règles de transitions : 0 : Rapide - 1 : En profondeur");
            scanf("%d", &regle);
            sortir2(li,co,tableau,etat,regle);
            printf("déplacements : ");
            int var2=0,var3=0,var4=0,var5=0;
            for (int i=0;etat[i]!=-1;i++)
            {
                printf("%d ",etat[i]);
                switch (etat[i])
                {
                    case 2:
                        var2++;break;
                    case 3:
                        var3++;break;
                    case 4:
                        var4++;break;
                    case 5:
                        var5++;break;
                }
            }
            printf("-1\n");
            printf("Soient :\n");
            printf("\tvers le bas : %d déplacements\n",var5);
            printf("\tvers la droite : %d déplacements\n",var3);
            printf("\tvers le haut : %d déplacements\n",var4);
            printf("\tvers la gauche : %d déplacements\n",var2);
            printf("pour un total de : %d déplacements (hors sortie)\n",var2+var3+var4+var5);

            liberer(li,tableau);
            men=menu();
        }
    }
    //menu=5
    return 0;
}
```