Projet informatique : Création de labyrinthe

IN 101 – ENSTA

`http://www.di.ens.fr/~pointche/enseignement/ensta/`

Résumé

Le but de ce projet est de créer une image représentant un labyrinthe possédant un chemin unique entre une entrée et une sortie et de nombreux chemins débouchant sur des impasses. Nous adopterons pour cela une technique simple, sous-optimale, mais dont la compréhension est très aisée.

1 Modalités de ce projet

Ce projet peut être effectué par binôme (2 personnes). Il sera évalué sur la base d'un compte-rendu envoyé par e-mail (à `David.Pointcheval@ens.fr`) composé

- des sources (veiller à la clarté des sources, et ne pas hésiter à commenter chaque étape) —à titre d'information, le module `construire.c` que vous avez à compléter nécessite l'ajout d'au plus **300 lignes**— ;
- d'un `Makefile` qui effectue la **compilation** de l'exécutable `labyrinthe` (voir la section 10.1 page 76 du polycopié, au sujet de l'utilitaire `make`, que vous avez intérêt à utiliser dès le début de vos travaux) ;
- d'un rapport, d'au plus 2 pages, au format électronique POSTSCRIPT (.ps) ou PDF Acrobat (.pdf) —tout autre format étant exclu— détaillant vos réflexions et les problèmes rencontrés, ainsi que ce que fait effectivement votre programme (fonctionnalités disponibles, parfaitement opérationnel, certaines restrictions, ...).

Ce compte-rendu doit être envoyé au plus tard le **11 mars 2005 à 12h00**.

Consulter la page `http://www.di.ens.fr/~pointche/enseignement/ensta1/projet` pour toute information supplémentaire, et notamment les morceaux de code qui vous sont fournis. Une liste de questions/réponses y sera également proposée.

2 Présentation du projet

2.1 Historique

Débutons tout d'abord par quelques définitions. Le «Labyrinthe» vient du grec laburinthos, qui signifie littéralement «palais des haches».

Les labyrinthes nous renvoient 4000 ans en arrière dans l'histoire. Ils prirent, pendant une longue période (3000 ans), la forme d'une courbe sinueuse sans jonctions. Il ne s'agissait pas de jeux mais d'un endroit dans lequel prenaient place les marches rituelles, les processions et les courses.

C'est dans la mythologie grecque que l'on trouve le labyrinthe le plus connu, celui qui enferma le Minotaure, fils de Knossos, tué par Thésée. Quelle que soit la véracité de ce mythe, on trouve le dessin d'un labyrinthe crétois à 7 anneaux sur des pièces de monnaie datant du 1er siècle avant J.C..

Les romains utilisaient souvent des labyrinthes représentés sur des mosaïques pour décorer le sol des pièces. On y retrouve d'ailleurs très souvent la personne de Thésée combattant le Minotaure, ou parfois des créatures semi-humaines de la mythologie.

Les invasions nordiques laissèrent en Angleterre de nombreuses traces de labyrinthes, réminiscence de ceux utilisés en Scandinavie et en Suède par les pêcheurs : une marche dans le labyrinthe assurait une bonne pêche et un retour sain et sauf d'une campagne de pêche dans les eaux terribles de la Mer du Nord. On retrouve aussi des dédicaces à la fertilité, le labyrinthe représentant alors le cordon ombilical.

Le 13^{e} siècle vit l'apparition en France, sous l'influence du christianisme, de labyrinthes représentant le chemin de la vie, utilisés comme dallage dans les cathédrales, souvent en forme de croix. Il n'est pas interdit de penser que certains de ces dessins furent en fait non pas créés mais copiés à partir d'anciennes formes, le contour en forme de croix masquant les origines païennes des dessins originaux.

La révolution industrielle du 19^{e} siècle permit la création de très beaux labyrinthes à l'aide de machines agricoles mais les conflits mondiaux du 20^{e} siècle les virent disparaître, le temps nécessaire à l'entretien de ces jardins sculptés n'étant plus vraiment disponible.

De nos jours, de nombreux labyrinthes existent, c'est même une passion qui prend de l'ampleur, notamment en Angleterre où l'on trouve les plus beaux labyrinthes de haies (cf. Fig. 1).

Fig. 1. Un exemple de labyrinthe.

2.2 Mathématique et labyrinthe

L'historique, tout comme la définition :

> **labyrinthe :** édifice composé d'un grand nombre de pièces disposées de telle manière qu'on en trouve que très difficilement l'issue.

ne nous aide pas vraiment pour l'élaboration d'un programme. Un labyrinthe est une surface connexe. De telles surfaces peuvent avoir des topologies différentes : simple, ou comportant des anneaux ou îlots. Les deux surfaces de la figure 2 ne sont topologiquement pas équivalentes.

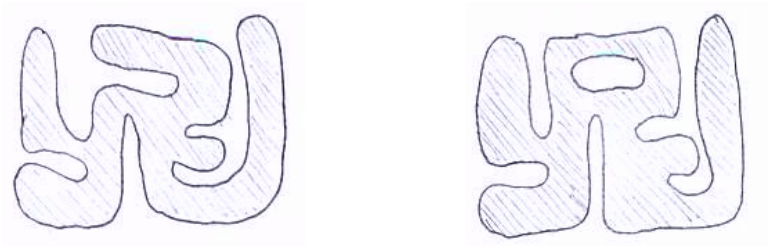

Fig. 2. A gauche un exemple de surface connexe simple et à droite une surface connexe avec îlot.

Cette différence, rapportée aux labyrinthes, conduit à deux espèces différentes : le labyrinthe «parfait» et le labyrinthe «imparfait» (cf. Fig. 3). La première espèce correspond à un chemin unique, passant par toutes les pièces. La seconde possède un chemin se recoupant qui peut éventuellement isoler des pièces. Il est tout à fait possible de décrire les labyrinthes par des

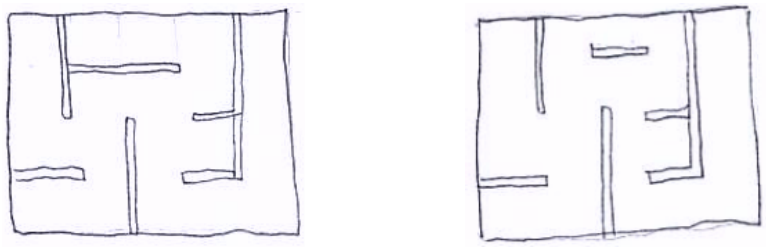

Fig. 3. A gauche un exemple de labyrinthe parfait, un seul chemin permet de relier un point à un autre. A droite un labyrinthe imparfait dans lequel il est possible d'emprunter plusieurs chemins pour aller d'un point à un autre. Cette figure est à rapprocher de la précédente (cf. Fig. 2) et les labyrinthes représentés sont en fait les répliques des formes utilisées.

arbres (cf. Fig. 4). Dans le cas d'un labyrinthe parfait il suffit de choisir un point au hasard et de construire un arbre dans lequel un noeud représente un point ou une pièce du labyrinthe et chaque branche un chemin pour aller dans la (les) pièce(s) voisine(s).

2.3 Méthodologie de création

Afin de construire un labyrinthe, la structure d'arbre nous invite à partir de l'entrée qui sera donc la racine de l'arbre et à construire un certain nombre de branches jusqu'à la sortie. Nous devons nous interdire de pouvoir passer d'une branche à une autre autrement qu'en remontant au noeud (à la pièce) qui relie ces branches et nous devons de plus ne pas laisser de pièces complètement isolées. Nous devons aussi conférer à l'ensemble un caractère aléatoire afin que le graphisme soit un peu artistique.

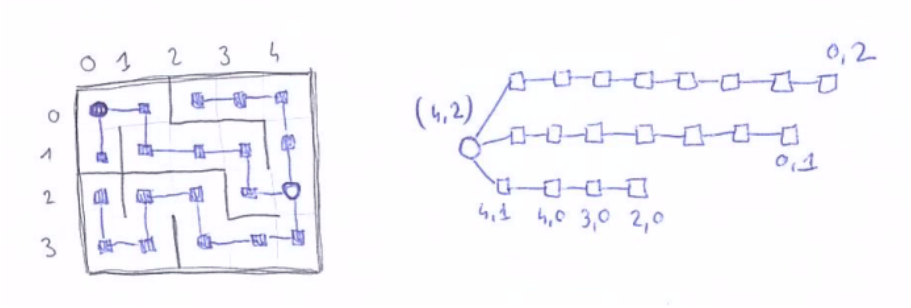

Fig. 4. Le labyrinthe parfait se traduit par un arbre dont la racine peut être choisi indifféremment parmi les pièces. Le labyrinthe imparfait verrait certaines branches reliées entre elles, perdant ainsi la structure d'arbre au profit de celle d'un graphe.

Nous pourrions donc partir d'un tableau et tracer les différents chemins par des marches aléatoires mais nous devrions alors analyser finement la figure obtenue pour tracer les murs afin d'éviter toute connexion entre chemin (pas de passage d'une branche à l'autre). Il est en fait beaucoup plus simple de partir d'un tableau représentant les différentes pièces séparées les unes des autres par un mur. Chaque pièce sera donc une cellule séparée de ses quatre voisines (nord, sud, est et ouest) par un mur. Le tracé des chemins consistera alors à abattre les murs en prenant soin d'éviter les interconnexions.

2.4 Structure informatique

Une cellule du labyrinthe peut être représentée de différentes manières. Elle peut comporter des champs donnant la présence ou l'absence des murs qui l'entourent, ou posséder un seul champ dont le codage permettra de retrouver cette absence ou cette présence. En effet si l'on adopte une représentation binaire et que l'on choisit de coder sur le bit de poids faible le mur Nord, puis le mur Ouest, puis le mur Sud et enfin le mur Est et si l'on choisit de dire que la valeur binaire «1» signifie la présence de mur, nous obtenons un mot de 4 bits comme le présente la

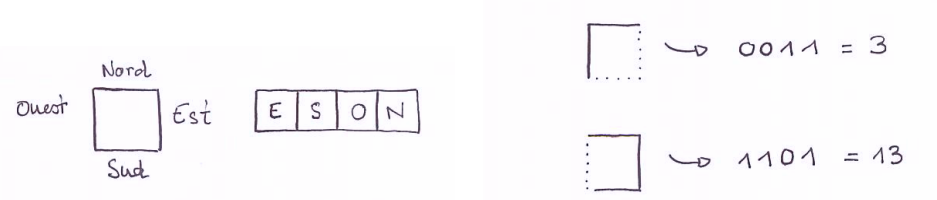

Fig. 5. Chaque cellule, ou pièce, du labyrinthe possède 4 murs. La présence ou l'absence de mur est codée sur un mot de 4 bits dans l'ordre présenté sur la figure de gauche, c'est à dire en partant du bit de poids faible vers le bit de poids fort : Nord Ouest Sud et Est. Deux exemples de codes binaires sont présentés dans la figure de droite.

figure 5. Nous pouvons donc représenter l'entité de base du labyrinthe de la manière suivante :

```
typedef struct {
  int compteur;
  unsigned char mur;
} Cellule;
```

Le champ `compteur` sera notre fil d'Arianne et nous permettra d'ordonner par la suite les cellules par ordre de visite et ainsi d'afficher le chemin menant de l'entrée à la sortie du labyrinthe. Nous verrons de plus que ce champ `compteur` nous permettra de différencier les cellules qui auront déjà reçu la visite de la fonction de recherche de celles n'ayant pas été visitées.

Le labyrinthe prendra la forme d'une structure incluant un tableau de cellules (tableau bidimensionnel), le nombre de cellules en largeur et en hauteur, l'épaisseur d'un mur (en pixels, nous verrons pourquoi après) ainsi que les dimensions d'une cellule (en pixels elles aussi). Nous trouverons aussi un champ permettant de représenter le labyrinthe sous la forme d'une image. On aboutit à la structure suivante :

```
typedef struct {
  int largeur;
  int hauteur;
  int epais_mur;
  int larg_cel;
  int haut_cel;
  Cellule **cel;
  gray **dessin;
} Labyrinthe;
```

2.5 Dessiner le labyrinthe

Une fois le labyrinthe créé, *i.e.* le tableau de cellules rempli, nous devons obtenir un affichage graphique. Le moyen le plus simple est de créer une image dans laquelle le fond sera blanc et les murs encore présents seront noirs. La librairie **libpgm** permet de manipuler les images très facilement. Une image est un tableau bidimensionnel que l'on obtient ainsi :

```
gray **pix;

pix = pgm_allocarray(largeur_en_pixels, hauteur_en_pixels);
```

Les images sont donc des tableaux bidimensionnels et les valeurs de ces tableaux sont comprises entre «0» (le noir) et «255» (le blanc). Chaque pixel est référencé comme présenté dans la figure 6. La sauvegarde de l'image ainsi créée au format `pgm` se fait très facilement au moyen de

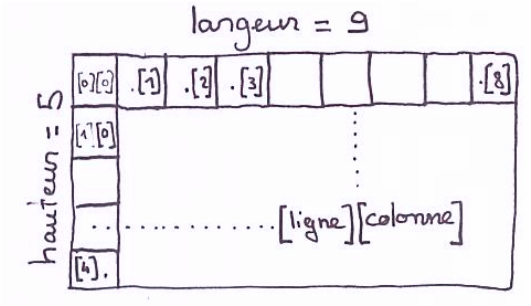

Fig. 6. Les pixels de l'images sont organisés dans un tableau bidimensionnel. Le pixel situé en haut et à gauche de l'image possède les coordonnées $(0, 0)$, et celui situé en bas à droite les coordonnées $(H - 1, L - 1)$ si L est la largeur de l'image (nombre de colonnes) et H la hauteur (nombre de lignes).

la fonction `pgm_writepgm()` qui s'utilise comme suit :

```
FILE *fp;
gray **pix;

pix = pgm_allocarray(largeur_en_pixels, hauteur_en_pixels);
...
pgm_writepgm(fp,pix,largeur_en_pixels, hauteur_en_pixels,255,0);
fclose(fp);
pgm_freearray(pix,hauteur_en_pixels);
```

La largeur et la hauteur de l'image dépendent bien sûr des valeurs définissant le labyrinthe, à savoir de `largeur`, `hauteur`, `epais_mur`, `larg_cel` et `haut_cel`. Le calcul est très simple comme le montre la figure 7. Attention, pour que le dessin soit agréable à regarder il est bon que les murs soient vraiment mitoyens et non en double !

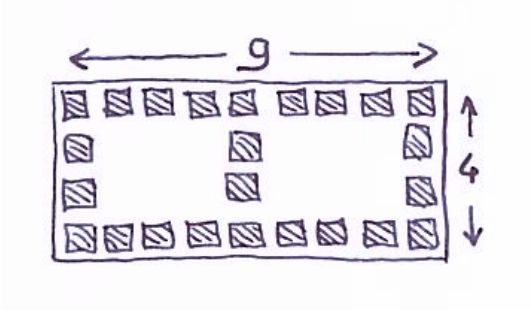

Fig. 7. Un labyrinthe de largeur 2 et de hauteur 1 dont les cellules possèdent une largeur de 3 et une hauteur de 2 et dont les murs sont d'épaisseur 1 se traduit par une image de 9 pixels de large et de 4 pixels de haut.

2.6 Algorithme de création

La méthode générale de construction ayant été définie, nous allons maintenant détailler les différentes étapes de construction du labyrinthe.

2.6.1 Allocation du labyrinthe

Dans un premier temps nous devons procéder à l'allocation des différentes structures qui nous permettent de manipuler le labyrinthe. Nous passerons les arguments nécessaires au déroulement du programme sur la ligne de commande, à savoir la largeur et la hauteur (en termes de pièce) du labyrinthe, l'épaisseur d'un mur en pixels (pour le dessin) ainsi que la hauteur et la largeur d'une cellule en pixels (toujours pour le dessin).

La fonction d'allocation suit le prototype suivant :

```
Labyrinthe *projet_alloue_labyrinthe(int h,int l,
                                     int epais_mur,
                                     int h_cel, int l_cel);
```

Cette fonction renvoie un pointeur sur une structure `Labyrinthe` dans laquelle les champs `largeur`, `hauteur`, `epais_mur`, `larg_cel`, `haut_cel` proviendront des valeurs passées en paramètres. La fonction alloue de plus le tableau bidimensionnel de cellules en utilisant la fonction `malloc()` classique de la bibliothèque `C` :

`cel[indice_ligne][indice_colonne]` représente la cellule située sur la ligne `indice_ligne` et la colonne `indice_colonne`.

Le champ `dessin` est alloué quant à lui en utilisant la fonction de la librairie `libpmg` présentée précédemment.

A l'issue de cette fonction, nous aurons donc à notre disposition une structure `Labyrinthe` dont les champs auront été alloués. Le tableau de cellules doit alors être initialisé.

Nous fournissons par la même occasion une fonction permettant de libérer les différentes structures (voir le fichier `alloc.h`).

2.6.2 Initialisation du labyrinthe

Nous séparons volontairement l'étape d'allocation en mémoire de l'étape d'initialisation des cellules. Ceci nous permettra de procéder à des variantes dans l'initialisation pour agrémenter le labyrinthe (variantes présentées dans la partie extension).

Nous devons débuter notre recherche de chemin par un labyrinthe dont toutes les pièces sont closes. Il faudra donc, pour chaque cellule du tableau, veiller à :

- initialiser le champ `compteur` à 0, les cellules sont ainsi toutes marquées comme n'ayant jamais été visitées,
- placer le champ `mur` à la valeur 15, *i.e.* valeur signifiant que tous les murs sont présents.

Nous profiterons de cette fonction d'initialisation pour remplir notre image avec une couleur de fond. On rappelle à ce propos que la valeur 0 représente le noir et la valeur 255 le blanc. Le prototype de la fonction d'initialisation est le suivant :

```
void projet_init_labyrinthe(Labyrinthe *lab);
```

Toutes ces fonctions d'allocation et d'initialisation vous sont fournies (dans les fichiers `alloc.h` et `alloc.c`). Les deux paragraphes suivants précisent ce que vous avez à programmer, en complétant le fichier `construire.c`.

2.6.3 Création du chemin principal

La création du chemin va se faire en plusieurs étapes toutes organisées de la même façon :

- prendre un point de départ, *i.,e.* une cellule non visitée,
- avancer dans le labyrinthe en brisant des murs pour construire un passage,
- s'arrêter lorsque l'on trouve le point final ou une impasse (selon l'étape).

Nous nous fixons dans un premier temps le point de départ. Il s'agira, afin de simplifier les choses, de la cellule se trouvant au milieu du bord gauche de l'enceinte du labyrinthe :

`cel[hauteur/2][0]`.

La sortie se trouvera quelque part sur le mur de droite de l'enceinte, dès que notre algorithme visitera une cellule de ce mur nous pourrons considérer que nous avons réalisé le tracé principal (cf. Fig. 8). Nous commencerons tout naturellement par briser le mur Ouest de la cellule de départ en plaçant donc le champ `mur` de ladite cellule à la valeur binaire `1101` soit `13` (cf. Fig. 5).

Nous rentrons ensuite dans la boucle de construction de chemin. A partir d'une cellule nous allons choisir de manière aléatoire une cellule voisine non visitée et nous allons briser les **deux murs** qui les séparent, à savoir le mur de la cellule où nous sommes et le mur mitoyen de la cellule où nous voulons aller comme présenté dans la figure 9. Notre destination peut être choisie en tirant aléatoirement une direction, soit un nombre entre 1 et 4, représentant respectivement le Nord, l'Ouest, le Sud ou l'Est. La boucle s'achève soit lorsque nous sommes arrivés sur le mur d'enceinte de droite (le chemin principal est terminé), soit lorsque nous arrivons dans une

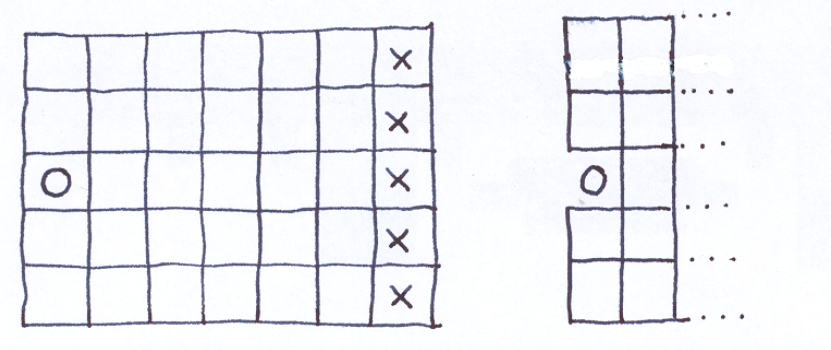

Fig. 8. La cellule de départ se trouve au milieu du mur d'enceinte de gauche. Elle est marquée d'un rond dans la figure. La sortie du labyrinthe se trouvera quant à elle n'importe où sur le mur d'enceinte de droite. Les cellules correspondantes sont marquées d'une croix dans la figure.

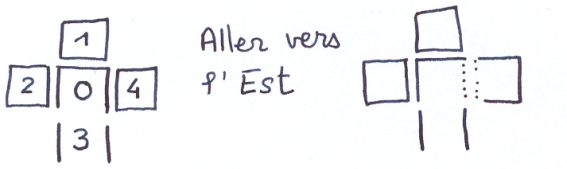

Fig. 9. La cellule courante porte l'indice 0. Ses quatre voisines portent les indices de 1 à 4. La cellule d'indice 3 a déjà été visitée comme le prouve la destruction des murs mitoyens. Les seules directions possibles pour continuer à progresser sont donc 1,2 et 4. Le choix aléatoire donne une direction Est, on brise donc les murs mitoyens entre la cellule courante et sa voisine de droite.

impasse comme cela est présenté dans la figure 10. Si le chemin principal est terminé nous passerons à une nouvelle étape. Nous allons déjà voir comment procéder lorsque l'on aboutit à une impasse.

Nous allons rechercher parmi toutes les cellules non visitées une cellule possédant une voisine visitée. Cette recherche est menée de manière exhaustive jusqu'à trouver cette cellule. Nous la choisirons alors comme nouveau point de départ en brisant les murs mitoyens entre elle et sa voisine occupée et nous entrerons de nouveau dans la boucle de construction de chemin : choix aléatoire d'une voisine libre, et progression jusqu'à la sortie ou jusqu'à une impasse. La recherche de cette cellule peut être menée de manière aléatoire afin de conférer un caractère assez tortueux au chemin permettant de franchir la labyrinthe. Dans les extensions, une méthode de recherche aléatoire est proposée.

Nous avons résumé cette première étape dans la figure 11 sous la forme de blocs fonctionnels.

2.7 Création des chemins de traverse

A l'issue de la création du chemin principal, même si celui-ci comporte plusieurs branches, il restera des cellules non visitées dans le labyrinthe. Le but étant de perdre les vaillants chevaliers qui s'aventureront dans les méandres de ce cauchemare, nous allons créer des chemins annexes pour visiter toutes les cellules du labyrinthe. La création de ces chemins annexes est très simple :

1. chercher une cellule non visitée voisine d'une cellule visitée,

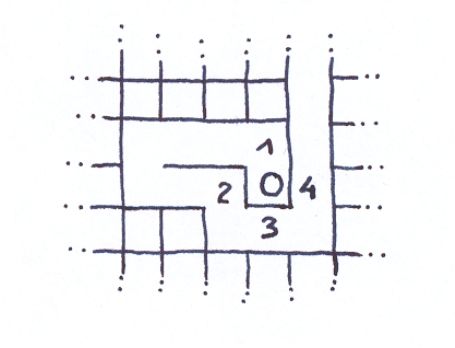

Fig. 10. La cellule courante porte l'indice 0. Ses quatre voisines portent les indices de 1 à 4. Elles ont toutes été visitées, le chemin ne peut pas être prolongé sous peine de briser la structure d'arbre du labyrinthe parfait.

2. briser les murs mitoyens entre ces deux cellules et se placer sur cette cellule,
3. progresser dans le labyrinthe comme pour la création du chemin principal jusqu'à aboutir à une impasse (interdiction de franchir le mur d'enceinte de droite car nous avons déjà une sortie !),
4. revenir à la recherche d'une cellule non visitée jusqu'à épuisement de toutes les cellules libres.

Nous pouvons aisément vérifier que nous avons respecté la structure d'arbre du labyrinthe parfait puisqu'à aucun moment nous n'autorisons un déplacement de la cellule courante vers une cellule déjà visitée.

2.8 Recommandations

2.8.1 Décomposition fonctionnelle

Le projet s'articule sommairement en quatre parties (dont 2 vous sont déjà fournies) :

1. allocation de la structure et initialisation des différents tableaux (**fourni**) ;
2. création du chemin principal (**à compléter**) ;
3. création des chemins de traverse (**à compléter**) ;
4. dessin du labyrinthe et sauvegarde de l'image (**fourni**).

Le dessin du labyrinthe utilise une image au format `PGM`. La librairie permettant de manipuler cette structure est déjà présente sur votre système. Pour compiler vos programmes vous devrez penser à utiliser la ligne de commande suivante (après avoir produit les fichiers objets de vos différents fichiers sources) :

```
gcc -lm -lpgm -o labyrinthe alloc.o construire.o dessin.o main.o
```

en prenant soin de bien éditer les liens avec la librairie mathématique (`libm.so` soit `-lm`) et la librairie d'images (`libpgm.so` soit `-lpgm`)

Les modules fournis permettent déjà une compilation complète, ainsi qu'une exécution partielle (le laryrinthe construit ne contient aucun chemin). Les «alertes» ou «warnings» à la compilation signalent juste la déclaration de fonctions non entièrement décrites : celles que vous avez à compléter !

Vous manipulez des tableaux, soyez humbles, faites des tests sur les indices que vous calculez, vous vous rendrez ainsi vite compte que vous accédez aux cellules du labyrinthe de votre voisin, et c'est, *a priori*, interdit par le système d'exploitation.

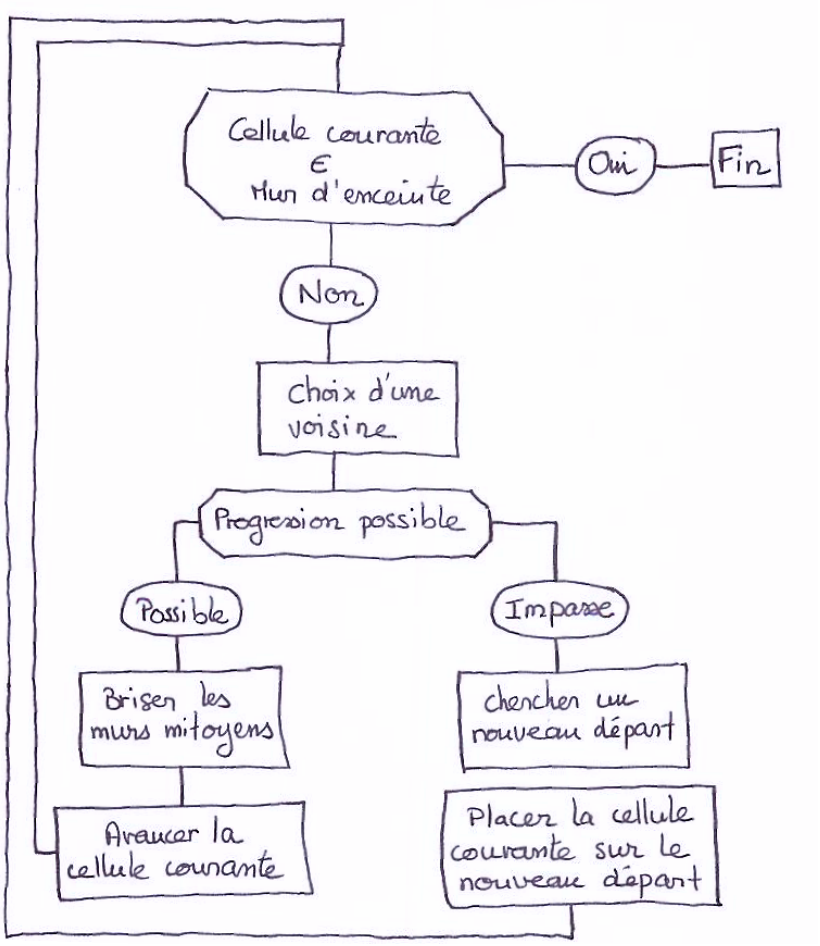

Fig. 11. Détail des blocs fonctionnels et de leur enchaînement dans l'étape de création du chemin principal.

3 Fonctions utiles

Outre les nombreuse fonctions qui vous sont déja fournies, parmi celles que vous avez à compléter, certaines sont pratiquement incontournables.

Tout d'abord, nous avons la fonction permettant de savoir si une voisine, dans une direction donnée, est libre :

```
void Test_Voisine_Libre(Labyrinthe *lab,
                        int x, int y, int direction);
```

Cette fonction retourne la valeur 1 si la voisine de la cellule de coordonnées (x, y) dans la direction `dir` est libre, et 0 sinon. On rappelle que les directions sont codées selon la suite horaire inverse en partant du Nord, soit Nord $\rightarrow$ 0, Ouest $\rightarrow$ 1, Sud $\rightarrow$ 2 et Est $\rightarrow$ 3. Le retour de la valeur 1 quand ladite voisine est libre n'est pas fait au hasard, en effet en répétant cette opération sur toutes les voisines et en sommant les résultats on obtient ainsi le nombre de cellules voisines libres.

Nous trouvons ensuite la fonction permettant de briser les murs mitoyens à deux cellules :

```
void Briser_Mur_Mitoyen(Labyrinthe *lab,
                        int x, int y, int direction);
```

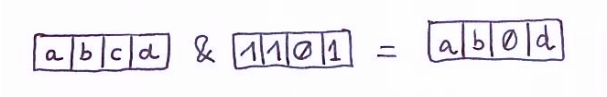

Fig. 12. Utilisation du mot binaire 1101 (soit la valeur 13) pour placer à 0 le mur Ouest d'une cellule.

Cette fonction doit placer le champ `mur` de la cellule de coordonnées x, y (x étant le numéro de colonne et y le numéro de ligne) de façon à ce que le mur se trouvant dans la direction `direction` devienne absent (bit correspondant à 0) sans toucher aux autres murs. On rappelle à ce propos l'opérateur bit à bit `&` qui permet de réaliser cette opération comme cela est montré dans la figure 12.

Nous trouvons ensuite la fonction qui permet de choisir une voisine libre :

```
int Choix_Voisine(Labyrinthe *lab, int x, int y);
```

Cette fonction prend en arguments les coordonnées de la cellule courante et retourne la direction dans laquelle il faut aller pour se placer sur la cellule libre voisine. Une direction inférieure à zéro peut signifier qu'il est impossible de prolonger le chemin en cours de construction. Cette fonction doit contenir un appel à une fonction de génération d'aléa (`rand()` ou `random()` —consultez les pages de manuel pour voir comment les utiliser, et procédez à quelques tests).

Cette fonction doit tout d'abord prendre connaissance du nombre de voisines libres disponibles. Soit N_l ce nombre. On tire alors au hasard un nombre compris entre 1 et N_l inclus. Soit N_c ce nombre. Ensuite il suffit de trouver la $N_c{}^{\text{e}}$ voisine libre disponible et donc la direction qu'il faut prendre pour y accéder.

4 Extensions multiples, variées et diverses

4.1 Adressage aléatoire

Lorsque l'on recherche des cellules libres pour le chemin principal ou les chemins de traverse, il peut être utile de disposer d'un balayage aléatoire du tableau de cellules. La réalisation est très simple. On construit tout d'abord un tableau monodimensionnel dont le nombre d'éléments est égal N_{cel}, le nombre de cellules. Dans ce tableau on place les nombres de 0 à $N_{cel} - 1$ par une boucle :

```
for(i=0;i<laby->largeur*laby->hauteur;i++) {
  tab[i] = i;
}
```

Ensuite, à l'aide d'une double boucle on mélange tous ces éléments de manière aléatoire :

```
for(j=0;j<laby->largeur*laby->hauteur;j++) {
  for(i=0;i<laby->largeur*laby->hauteur;i++) {
    if (random()<MAX_RAND/2) {
      tmp = tab[i];
      tab[i] = tab[j];
      tab[j] = tmp;
    }
  }
}
```

L'utilisation de ce tableau est ensuite assez simple car on passe d'une valeur du tableau aux coordonnées d'une cellule par soit une division entière soit un modulo :

```
x_cel = tab[k] % laby->largeur;
y_cel = tab[k] / laby->largeur;
```

Ceci vous permettra d'avoir un chemin principal assez alambiqué !

4.2 La solution s'il vous plait !

Comme tout bon créateur d'un problème vous devez être en mesure d'en posséder la solution ! Le tracé du chemin traversant le labyrinthe est assez simple à réaliser, et automatiquement tracé pour peu que vous ayiez pris soin de maintenir à jour le champ `compteur` des cellules et de l'incrémenter à chaque fois. En partant de la sortie (la seule cellule du mur d'enceinte de droite ne possédant pas de mur Est), la fonction `Trouver_Voisine_Precedente` (dans `dessin.c`) recherche parmi les voisines pour lesquelles le passage est possible, celle possédant une valeur de compteur inférieure à la valeur de la cellule courante. De proche en proche on remonte ainsi jusqu'à l'entrée. Puis on place un petit caillou noir dans chacune des cellules au fur et à mesure.

4.3 Labyrinthe à motif

Comme vous avez dû le remarquer, le champ `mur` contient huit bits et nous n'en utilisons que quatre. Nous pourrions par exemple utiliser un bit supplémentaire pour remplir certaines cellules de béton armé et ainsi en empêcher l'accès. Ceci permettrait de créer des labyrinthe dans lesquels le tracé des chemins s'organiserait autour d'une figure prédéfinie. Pour cela il suffirait de créer à l'aide d'un logiciel de dessin comme `The Gimp` une image dont la taille serait celle du labyrinthe (attention cette fois on compte en nombre de cellules !) et qui posséderait un fond

blanc (les cellules libres) et un dessin noir (les cellules en béton). Lors de l'étape d'initialisation nous pourrions rajouter le chargement de cette image :

```
FILE *fp;
gray **pix;
int colsP,rowsP,maxvalP;
int i,j;
  fp =fopen(‘‘monfichier.pgm’’,’’r’’);
  pix = pgm_readpgm(fp,&colsP,&rowsP,&maxvalP);
  fclose(fp);

  for(j=0;j<rowsP;j++) {
    for(i=0;i<colsP;i++) {
      if (pix[j][i] == maxvalP) {
        lab->cel[j][i].mur = 16;
      } else {
        lab->cel[j][i].mur = 0;
      }
    }
  }
```

Nous avons ainsi placé dans le bit numéro 5 (celui qui est juste avant les 4 bits de poids faibles consacrés à la description des murs) le fait qu'une cellule est libre d'être visitée (`mur = 16`) ou qu'au contraire elle soit impossible à visiter (`mur = 0`).

Lors du dessin du labyrinthe (il s'agit alors de construire une autre image !) nous pourrons remplir ces cellules bétonnées avec la couleur des murs et ménager ainsi un joli dessin.

4.4 Labyrinthe à étage

Nous pouvons tout aussi naturellement définir une cellule du labyrinthe comme étant un escalier de communication entre l'étage courant et un étage inférieur, cette cellule serait alors la porte d'entrée du nouveau labyrinthe (principe du jeu à niveau). Il faudra pour cela modifier légèrement l'algorithme de création du chemin principal (la fin est une cellule unique).

5 Exemples

Voici pour finir quelques exemples de labyrinthes... sans la solution ! !

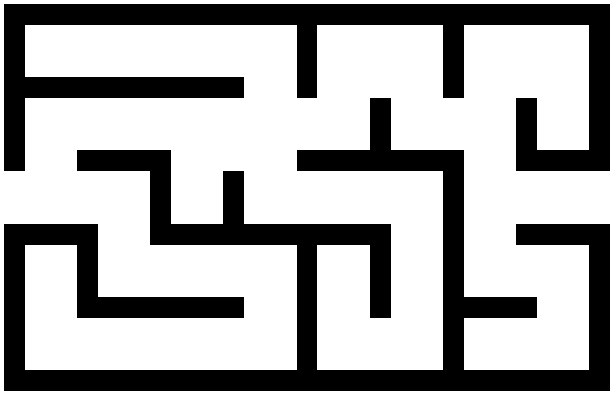

Fig. 13. Un labyrinthe très simple !

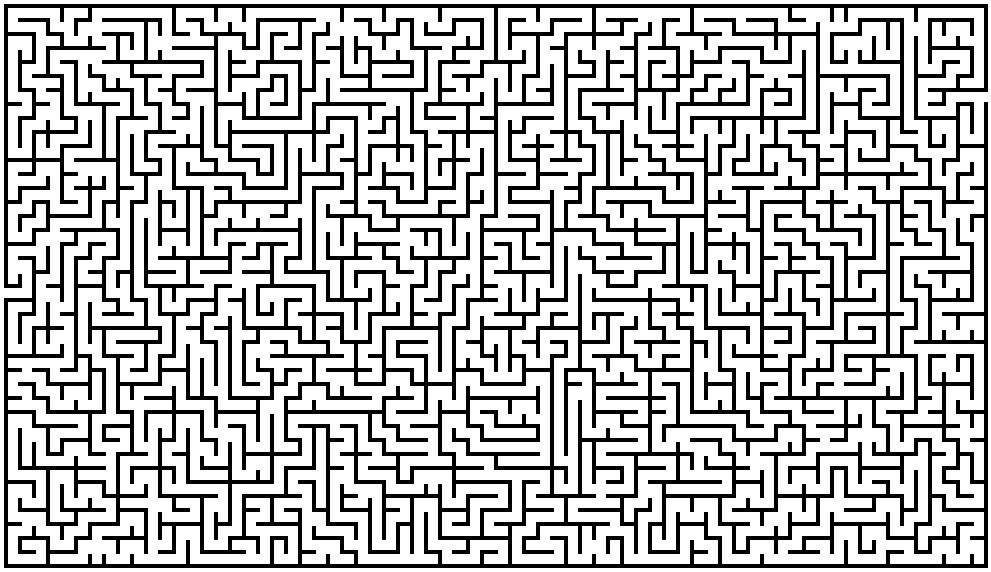

Fig. 14. Un labyrinthe plus difficile, et sans la solution !