Devoir maison n°2 - À rendre le 10 Janvier 2024
Structure UnionFind et applications

Notions abordées

- structure de données UnionFind, implémentation avec des ”arbres”
- génération aléatoire en C
- parcours de graphe implicite

Avant de commencer, télécharger sur cahier de prépa l'archive contenant le squelette du code. Après décompression, prendre connaissance des fichiers fournis et de leur organisation : on s'efforcera de séparer le code comme suit.

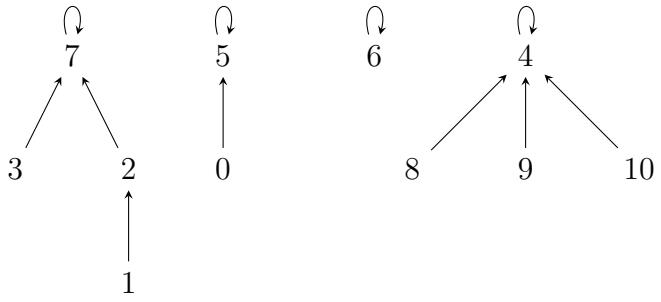
- les fichiers `.h` contiennent les déclarations de type et de fonctions ;
- les fichiers `.c` contiennent le code des fonctions déclarées dans le `.h`, mais pas de `main` ;
- les fichiers `tests_xxx.c` contiennent le code qui permet de tester els fonctions codées dans le `.c`, et notamment une fonction `main` qui appelle les tests voulus, ainsi leutr compilation produit un execéutable qu'il convient d'exécuter pour bien tester le code.

Il peut notamment être utile de regarder quelles commandes de compilations sont disponible dans le Makefile.

Exercice 1 : Implémentation d'une structure UnionFind en C

Dans cet exercice on se propose d'implémenter une structure UnionFinden C. On rappelle que cette structure a pour objectif la représentation d'une partition d'un ensemble fini S. Dans tout cet exercice on suppose que $S = [\![0, n-1]\!]$ pour un certain $n \in \mathbb{N}$.

Structure arborescente. On réutilise dans cet exercice l'idée fondamentale introduite en classe de représentation des classes d'équivalences au moyen d'une structure arborescente. À chaque élément de S on adjoint un élément de S qui est son ”père”. Ainsi pour chaque élément de S on peut aller visiter son père, puis le père de son père, etc. . .. Afin d'assurer qu'un tel processus termine, le père d'un élément ne peut être son descendant strict. Cependant le père d'un élément peut-être lui-même, auquel cas on dit que cet élément est une **racine**. L'ensemble S étant fini, le parcours de père en père depuis n'importe quel élément x conduit alors nécessairement à un élément racine : c'est le représentant de la classe de x. **Attention :** Les arbres manipulés dans ce DM n'ont pas la structures inductive usuelle des arbres. Si un arbre est souvent défini comme un nœud contenant deux fils qui sont eux-mêmes des arbres, ici un nœud contient un pointeur vers son père seulement, il n'a pas accès à ses fils, qui peuvent d'ailleurs être en nombre quelconque (0,1,2 ou plus...).



Ainsi illustration ci-dessus représente la partition de $[\![0, 10]\!]$ en 4 classes : $\{1, 2, 3, 7\}$, $\{0, 5\}$, $\{6\}$ et $\{4, 8, 9, 10\}$.

Représentant canonique. Cette représentation met de plus en avant un représentant canonique de chaque classe : la racine de l'arbre représentant la classe. Ainsi $\{1, 2, 3, 7\}$ admet 7 comme représentant canonique. De même $\{0, 5\}$ (resp. $\{6\}$, resp. $\{4, 8, 9, 10\}$) admet 5 (resp. 6, resp. 4) comme représentant canonique.

Rang d'un nœud. La remontée de père en père dans un arbre est d'autant plus coûteuse (dans le pire cas) que les arbres sont hauts. En vue de manipuler des arbres les moins hauts possibles, on souhaite connaître la hauteur des arbres impliqués lors d'une opération d'union. Pour cela, on conserve au niveau de chaque nœud une majoration de la hauteur du sous-arbre qu'il engendre. Plus précisément, on appelle **rang** d'un nœud x une majoration de la hauteur du sous-arbre enraciné x. c'est-à-dire l'arbre constitué des élément qui admettent x comme père ou grand père ou ancêtre. Dans l'exemple ci-avant, le nœud étiqueté par la valeur 4 admet 1 comme rang (mais aussi 42).

Types en C. Afin de stocker les informations de père et de rang pour un nœud de l'arbre, on définit le type structuré `uf_elem_t` ci-dessous.

```c
struct uf_elem_s {
  int             rank   ;
  int             elem   ;
  struct uf_elem_s* parent ;
};

typedef struct uf_elem_s* uf_elem_t;
```

Ainsi, dans un objet de type `struct uf_elem` représentant un nœud x sont bien stockées les informations suivantes :

- l'entier représenté par le nœud : `elem`,
- le père du nœud x (qui est lui même un nœud) : `parent`,
- le rang du nœud x : `rank`.

On définit alors le type `uf_partition_t` pour représenter les structures UnionFind comme tableaux de nœuds. ♣

```c
typedef uf_elem_t* uf_partition_t ;
```

♣. Vu l'hypothèse $S = [\![0, n[\![$, on aurait pu stocker directement le rang et le père de chaque élément $i \in S$ dans la case d'un tableau. Outre l'intérêt pédagogique de faire manipuler des pointeurs, cette structure avec des cellules pour les nœuds serait plus facilement adaptable à un ensemble S quelconque. Le lien entre un élément $s \in S$ et le nœud le représentant serait alors géré par un dictionnaire (implémenté avec une table de hachage par exemple). Le passage d'un élément à son nœud ne serait pas exactement en temps constant, mais puisqu'un nœud connaît directement le nœud représentant son père, il n'y aurait pas besoin de repasser par le dictionnaire à chaque étape, il suffirait de faire un appel au dictionnaire au début de l'opération trouver.

Matériel fourni. Les définitions de type et déclarations de fonctions sont fournies dans le fichier `uf.h`. Le fichier `uf.c`, qui contient déjà une fonction d'affichage, est à compléter avec les fonctions demandées dans la suite de cet exercice. Des jeux de tests sont fournis dans le fichier `tests_uf.c`. De plus un Makefile propose des commandes de compilations utiles pour ces fichiers.
Aussi l'exécution de la fonction `main` de de `uf_exemple.c` reproduite ci-dessous conduit à l'affichage en regard. □

```c
int main() {
  uf_partition_t ex = exemple();
  print_uf_partition_verbose(ex, 11);
}
```

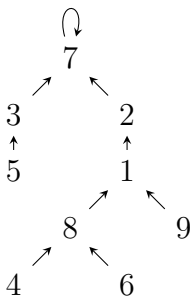
```
0 ~> 5
1 ~> 2 ~> 7
2 ~> 7
3 ~> 7
4
5
6
7
8 ~> 4
9 ~> 4
10 ~> 4
```

Q. 1 Définir une fonction `uf_partition_t uf_initialize(int size)` créant une structure Union-Find représentant la partition de $[\![0, \texttt{size} - 1]\!]$ en singletons. *On pourra s'inspirer de ce qui est fait dans la fonction `exemple` de `uf_exemple.c`.*

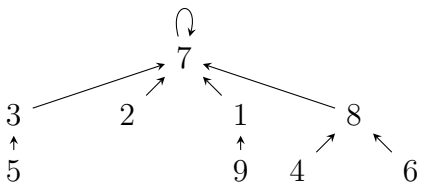
Q. 2 Définir une fonction `void uf_free(uf_partition_t tab, int size)` permettant la libération de tous les nœuds de la structure, ainsi que celle du tableau permettant de les stocker.

Q. 3 Définir une fonction `uf_elem_t uf_find_no(uf_elem_t x)` retournant le nœud du représentant de la classe de `x`. *Cette fonction ne doit pas modifier la structure.*

Compression de chemin. On remarque que la complexité algorithmique de la fonction `find` dépend linéairement de la longueur du chemin menant de l'élément dont on cherche un représentant à la racine de l'arbre représentant sa classe. Ainsi dans un but d'amortir le coût de parcours, on redéfinit la fonction `uf_find` de sorte qu'elle profite du parcours pour compresser le chemin d'un élément à son représentant. Par exemple dans la structure donnée ci-dessous, lors de la recherche du représentant de 8, on traverse les nœuds 8, 1 et 2, pour finalement trouver 7. On sait alors non seulement que l'entier 8 a 7 comme représentant, mais que c'est aussi le cas des entiers 1 et 2. On en profite donc pou attribuer 7 comme père à tous ces nœuds.



Avant raccourcissement du chemin $(8, 1, 2, 7)$



Après raccourcissement du chemin $(8, 1, 2, 7)$

Q. 4 Définir une fonction `uf_elem_t uf_find(uf_elem_t x)` retournant le nœud du représentant de la classe de `x`, et appliquant le raccourcissement de chemin aux nœuds croisés entre `x` et la racine. *On pourra utiliser **habilement** la récursivité.*

Union par rang. On a vu en cours, qu'il était possible d'unir deux classes de représentants respectifs x et y en attribuant y comme père à x, ou x comme père à y. Le choix entre ces deux possibilités va être guidé par les rangs des nœuds x et y. En effet l'objectif de la structure étant de minimiser

le coût de l'opération `uf_find`, on souhaite minimiser les longueurs des chemins vers la racine, et donc minimiser la hauteur des arbres de la structure. Ainsi, si le rang de x est strictement inférieur à celui de y, on préférera placer x sous y (mettre y comme père de x) que le contraire. Dans le cas où les rangs des nœuds x et y sont identiques, **on préférera placer x sous y**. On fera attention à mettre à jour convenablement les valeur des rangs des nœuds x et y. **Attention :** On remarque que le rang est bien une majoration de la hauteur de l'arbre et non la hauteur de l'arbre puisque la compression de chemin peut réduire la hauteur d'un arbre sans pour autant que l'information soit propagée dans les champs rang de tous les nœuds concernés.

Q. 5 Définir une fonction `void uf_union(uf_elem_t a, uf_elem_t b)` modifiant la structure de sorte que les classes de `a` et `b` sont réunies.

Exercice 2 : Génération de labyrinthe en C

Nous nous intéressons dans cet exercice à des labyrinthes dits parfaits. On proposera deux algorithmes de générations aléatoires de tels labyrinthes, avant de fournir un algorithme de résolution (c'est-à-dire un algorithme qui recherche le chemin de l'entrée à la sortie).

Labyrinthe parfait. Un labyrinthe est dit **parfait** dès lors qu'entre toute paire de cases, il existe un et un seul chemin de l'une à l'autre dans le labyrinthe. Autrement dit le graphe dont les sommets sont les cases du labyrinthe et dont les arêtes joignent deux cases ayant un côté commun sans mur est un arbre. Ci-dessous deux exemples de labyrinthes. Le labyrinthe 1a est un exemple de labyrinthe parfait : on peut vérifier que chaque case est accessible depuis n'importe quelle autre et qu'à chaque fois le chemin est unique. Le labyrinthe 1b n'est pas un labyrinthe parfait. La case ⬟ n'est accessible depuis aucune autre case, et il est possible d'aller de la case ● à la case ■ de deux manières différentes (chemin 1 en ⇢ et chemin 2 en →). Dans toute la suite nous considérerons que l'entrée du labyrinthe se trouve dans la case au nord ouest (de coordonnées $(0, 0)$) et la sortie dans le coin au sud est (de coordonnées $(\text{height} - 1, \text{width} - 1)$). On remarque donc que l'indexation des cases suit la convention matricielle : numéro de ligne $\times$ numéro de colonne ♣. Ainsi sur la figure 1a, on peut voir le chemin solution en →.

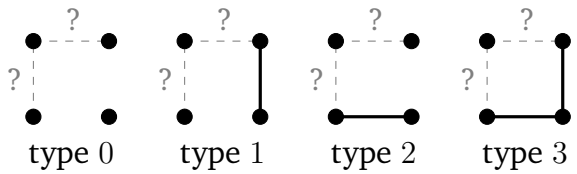
(a) Labyrinthe parfait (b) Labyrinthe non parfait

Figure 1 – Exemples de labyrinthes

Cases du labyrinthe. On remarque qu'un mur étant adjacent à 2 cases, si chaque case retenait l'absence ou la présence des 4 potentiels murs autour d'elle, la même information serait stockée deux fois, or une fois suffit ♡. Ainsi, chaque case du labyrinthe stocke l'information de présence ou absence de mur seulement pour sa face est et sa face sud ♠. Par convention, on peut supposer que

♣. Avec bien sûr, comme souvent en informatique, une indexation à partir 0
♡. d'autant plus qu'on peut accéder en temps constant à une case voisine pour récupérer l'information.
♠. les côtés à droite et en bas sur le dessin

les cases les plus à l'ouest du labyrinthe ont un mur sur leur face ouest, et celles les plus au nord un mur sur leur face nord. En effet, ces murs ne changent en rien l'accessibilité entre les cases du labyrinthe, ils font partie de l'enceinte extérieure, qu'on suppose murée par convention (et sur les dessins).
Une case porte alors deux murs potentiels, il y a donc 4 configurations possibles. Le dessin ci-dessous les résume et précise l'entier utilisé pour représenter chacune en C. Un trait pointillé représente une information stockée dans une autre case (celle à l'ouest ou au nord), un trait plein indique la présence d'un mur, une absence de trait l'absence de mur.



On dira de deux cases de coordonnées (i, j) et (i', j') qu'elles sont voisines si $|i - i'| = 1$ et $j = j'$ ou si $|j - j'| = 1$ et $i = i'$, autrement dit si elles sont adjacentes, c'est-à-dire si elles ont un côté commun. Ainsi une case a au plus 4 voisines, pas 8 (un sommet en commun ne suffit pas ici).

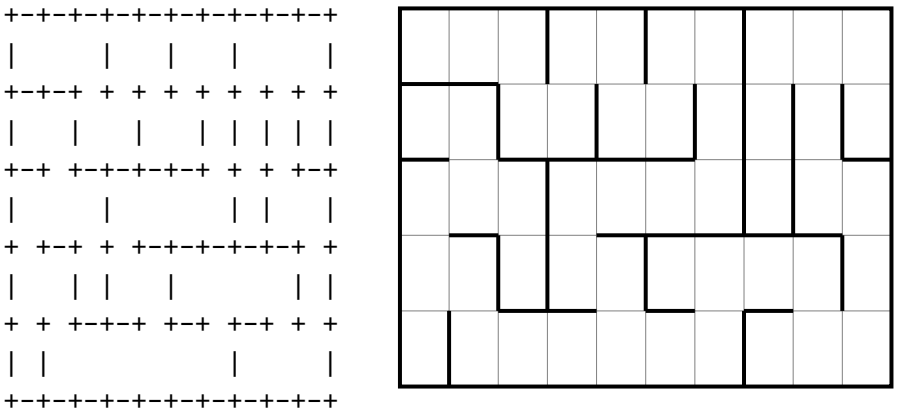
Représentation en mémoire du labyrinthe. Finalement un labyrinthe est la donnée d'une hauteur h, d'une largeur w et d'un tableau de $h \times w$ entiers stockant le type de chaque case du labyrinthe. Bien qu'on représente graphiquement les labyrinthes en deux dimensions, et qu'on utilise parfois une double indexation ligne $\times$ colonne pour identifier une case, on stockera dans notre implémentation le type des cases dans un tableau unidimensionnel. On utilise alors l'indexation "classique" des couples d'entiers de $[\![0, h-1]\!] \times [\![0, w-1]\!]$ dans $[\![0, hw-1]\!]$ définie par : $(i, j) \mapsto iw+j$. Autrement dit on "linéarise" le tableau 2D en le parcourant ligne par ligne, de haut en bas.

Matériel fourni. Les définitions de type et déclarations de fonctions sont fournies dans le fichier `laby.h`. Le fichier `laby.c`, qui est à compléter avec les fonctions demandées dans la suite de cet exercice, contient déjà quelques fonctions pour la manipulation de labyrinthes, permettant notamment :

- d'afficher un labyrinthe, avec ou sans chemin dessus ;
- de convertir les coordonnées dans un tableau bidimensionnel à l'indice dans un tableau unidimensionnel, et vice-versa ;
- de tester si un labyrinthe est plein ;
- de tester si une case existe bien dans un labyrinthe ;
- de tester s'il n'y a pas de mur entre deux cases d'un labyrinthe ;
- de casser dans un labyrinthe le mur entre deux cases du labyrinthe.

Les jeux de tests sont à implémenter dans le fichier `tests_laby.c`. De plus un Makefile propose des commandes de compilations utiles pour ces fichiers.
On pourra, par exemple l'appel `void draw_laby(laby_t laby)` pour `laby` le labyrinthe représente ci-dessous à droite, on obtient l'affichage ci-dessous à gauche.

Lien avec les graphes. On peut voir un labyrinthe comme un graphe : l'ensemble de ses sommets est l'ensemble des cases du labyrinthe, une arête est présente entre deux sommets si elles sont adjacentes (ou voisines) et s'il n'y pas de mur entre ces deux cases. Ainsi un chemin entre deux cases dans le labyrinthe correspond à une chaîne entre les deux sommets correspondant à ces deux cases dans le graphe. Un labyrinthe parfait est alors un labyrinthe dont le graphe associé est un arbre.



(a) Affichage produit (b) Labyrinthe représenté

FIGURE 2 – Exemples d'affichage avec la fonction `void draw_laby(laby_t laby)`

Il existe plusieurs types d'algorithmes générant des labyrinthes parfaits de dimensions données. Dans ce DM, nous en verrons deux, qui ont pour point commun de construire un labyrinthe parfait en cassant des murs à partir du labyrinthe **plein**, c'est-à-dire celui où tous les murs possibles sont présents. En terme de graphes, cela équivaut à partir du graphe vide (*i.e.* sans aucune arête), et ajouter des arêtes (entre des sommets correspondants à deux cases adjacentes seulement).

1. Création de labyrinthes parfaits au moyen d'un parcours en profondeur

Le premier algorithme de génération du labyrinthe parfait qu'on considère consiste à réaliser un parcours en profondeur du graphe de toutes les adjacences possibles (*i.e.* le graphe correspondant au labyrinthe sans mur, où chaque case est reliée à ses 4 voisines ♣). En effet, dans le cas d'un graphe connexe, l'arborescence associée à un parcours (en profondeur ou non), fournit un arbre couvrant, c'est-à-dire dans notre cas un labyrinthe parfait. Le choix de faire le parcours en profondeur est motivé par le fait que c'est le parcours obtenu naturellement par programmation récursive, simplifiant ainsi les structures de données à mettre en place pour le parcours. En terme de labyrinthe, cet algorithme se traduit comme suit.

- On crée le labyrinthe plein, et une structure d'ensemble de cases visitées vide.
- On explore récursivement le labyrinthe en maintenant à jour l'ensemble des cases visitées, en commençant par la case $(0, 0)$.
- Lors de la visite d'une case c, on traite ses 4 cases voisines dans un ordre aléatoire. Pour chacune d'elles, s'il s'agit d'une case non encore visitée, on détruit le mur la séparant de c, on la marque comme visitée, puis on explore récursivement le labyrinthe à partir de cette case.

L'ensemble des cases visitées pourra être implémenté par un tableau de booléens indiquant, pour chaque case, si oui ou non elle a été visitée.

Q. 1 Implémenter cet algorithme dans une fonction `void generate_laby(laby_t laby)` prenant en argument un labyrinthe plein (toutes les cases sont de type 3), et cassant des murs en suivant l'algorithme décrit ci-dessus. *On pourra définir une fonction récursive de signature* `void rec_generator(laby_t laby, bool* visited, int i, int j)` *prenant en argument l'état courant du labyrinthe, le tableau des visitées et la position courante de la visite.*

Q. 2 ★ Proposer une implémentation non récursive de cet algorithme ?

♣. sauf celles des bords qui ont moins de voisines

2. Création de labyrinthes parfaits au moyen d'une structure UnionFind

Pour fabriquer un labyrinthe parfait de hauteur h et de largeur w, il suffit de partir d'un labyrinthe plein, et de lui ôter $wh-1$ murs séparant deux ensembles de cases ne communiquant pas par ailleurs. En effet un labyrinthe décrit une relation d'équivalence sur les cases : deux cases sont équivalentes si et seulement s'il existe un chemin de l'une à l'autre (ou réciproquement), et donc une partition de ses cases : la partition en classe d'équivalence. En particulier le labyrinthe plein correspond à une relation vide et à la partition en singletons, qui a donc autant de classes que le labyrinthe a de cases, soit wh, et à l'opposé, un labyrinthe parfait correspond à la partition où toutes les cases sont réunies dans une même classe.

Dans la figure ci-dessous, vous trouverez une représentation de ces classes d'équivalence au cours des 11 étapes de la construction du labyrinthe à 12 cases de la figure 1a. Les cases appartenant à la même classe d'équivalence sont représentées dans la même couleur (et sont délimitées par les traits noirs épais représentant les murs). Une étape de construction revient donc à enlever un mur entre deux cases de couleurs différentes.

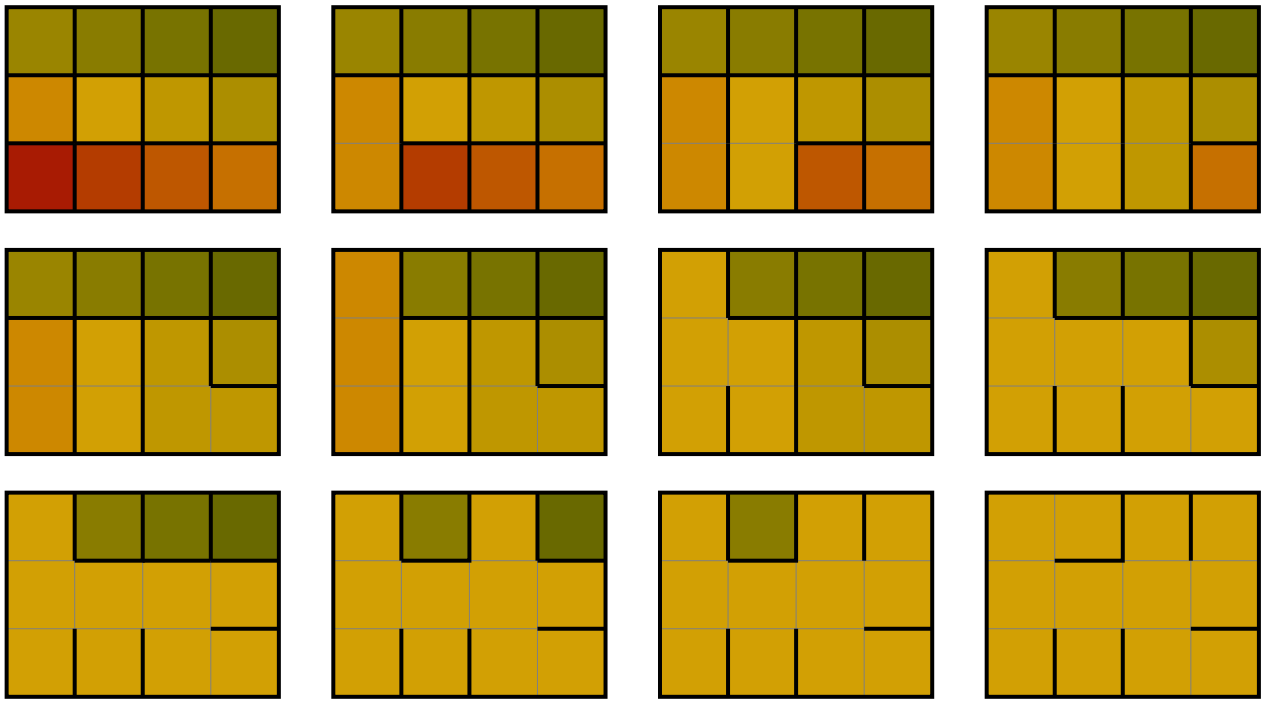


FIGURE 3 – Création du labyrinthe 1a

Le partitionnement des cases en classes d'équivalences sera représenté au moyen de la structure UnionFind définie dans l'exercice 1.
On décrit ci-dessous un algorithme de génération de labyrinthe basé sur les remarques précédentes.

- On calcule un tableau contenant tous les murs du labyrinthe plein, on le mélange.
- On crée P une structure UnionFind partitionnant les cases du labyrinthe en singletons.
- Tant qu'on n'a pas cassé $wh-1$ murs, on prend un nouveau mur selon l'ordre du tableau. S'il sépare deux cases ne se trouvant pas dans la même classe de la partition P, alors on casse ce mur et on réunit alors les des classes de ces cases dans P.

Un mur pourra être représenté par 4 entiers : les coordonnées (i, j) des deux cases se trouvant de part et d'autre du mur. Puisque le mur d'enceinte est considéré muré par convention, les murs au sud des cases de la dernière ligne ne sont pas à considérer ici, de même que les murs à l'est des cases de la dernière colonne, ainsi, pour un labyrinthe de dimensions $h \times w$, il y a $(h-1)w + (w-1)h$ soit $2wh - w - h$ murs à considérer. Ignorer les murs d'enceinte permet aussi d'assurer que les cases

de part et d'autre d'un mur sont toutes les deux des cases du labyrinthe, *i.e.* de coordonnées dans $[\![0, h[\![\times [\![0, w[\![$.

Q. 3 Définir un type `mur_t` et une fonction `mur_t* tab_murs_laby_plein(laby_t laby)` retournant le tableau de tous les murs du labyrinthe plein `laby` qui est passé en argument.

Afin de mélanger le tableau des murs, on se propose d'utiliser le mélange de Fisher-Yates, que l'on décrit ci-dessous.

Algorithme 1 : Mélange de Fisher-Yates

Entrée : Un tableau T de taille n
Sortie : Le tableau T est mélangé en place

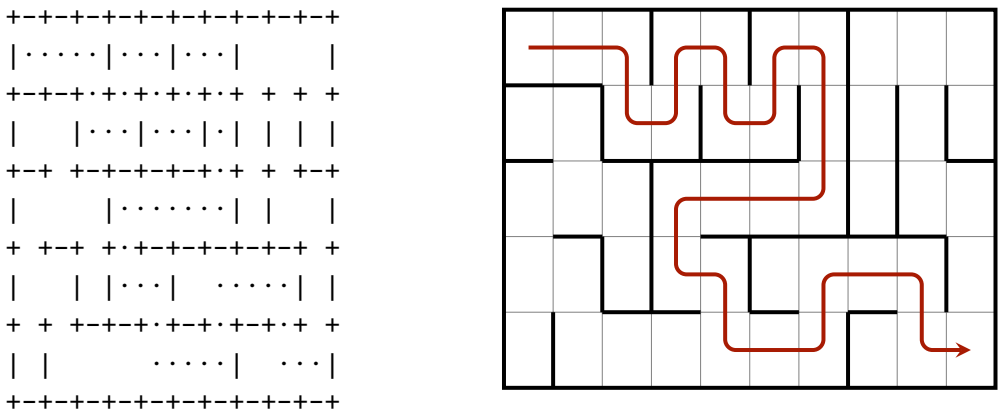
1 **pour** $i = n - 1$ **à** 1 **faire**
2 $\quad j \leftarrow \mathcal{U}([\![0, i]\!])$; (*Choix aléatoire uniforme dans $[\![0, i]\!]$*)
3 $\quad$ Échanger $T[j]$ et $T[i]$;

Q. 4 Définir une fonction `void melange_liste_murs(mur* murs)` mélangeant la liste `murs` au moyen de l'algorithme de Fisher-Yates.

Q. 5 Définir une fonction qui implémente l'algorithme de génération de labyrinthe parfait présenté ci-avant.

3. Exploration du labyrinthe

Dans cet exercice on s'intéresse à la recherche d'un chemin dans le labyrinthe, on rappelle que l'entrée du labyrinthe se trouve dans la case en haut à gauche (de coordonnées $(0, 0)$) et la sortie dans le coin en bas à droite (de coordonnées $(\text{hauteur} - 1, \text{largeur} - 1)$).
On précise que `compagnon_laby.c` fournit une fonction `void draw_laby_with_visited(laby_t laby, bool* ouverts)` permettant de représenter un labyrinthe, mais aussi les cases ouvertes selon `ouverts` en affichant un point en leur centre. Un exemple d'un tel affichage en ligne de commande est fourni ci-dessous.



(a) Affichage produit (b) Labyrinthe et chemin représentés

FIGURE 4 – Exemples d'affichage avec la fonction `draw_laby_with_visited`

Q. 6 Définir une fonction `bool* solve_labyrinthe(laby_t laby)` qui cherche un chemin de l'entrée vers la sortie du labyrinthe, au moyen d'un parcours en profondeur du labyrinthe. Le chemin est renvoyé sous la forme d'un tableau de booléens qui indique les cases présentes sur le chemin. *On pourra définir une fonction récursive* `bool rec_solver(laby_t laby, bool*`

`visited, int i, int j)` *prenant en argument le labyrinthe, le tableau des visités et la position courante de la visite du labyrinthe.*

4. Réparation d'un labyrinthe

Dans cet exercice on suppose qu'on part d'un labyrinthe initialement construit, et qu'on cherche à le rendre parfait. Cet exercice est libre : on ne fournit pas d'algorithme à suivre, vous êtes libre de définir le votre. On pourra par exemple définir un algorithme procédant par étapes. La première étape consiste à ajouter des murs pour rendre le graphe sous-jacent acyclique, et la deuxième phase à casser des murs pour le rendre connexe sans perdre le caractère acyclique.

Q. 7 Définir une fonction `void repare(laby_t laby)` qui transforme `laby` (*i.e.* ajoute des murs et supprime des murs) en un labyrinthe parfait. *On ne veut pas créer un labyrinthe l_f sans rapport avec le labyrinthe initial l_i, au contraire pour deux cases a et b, s'il existait des chemins reliant a et b dans l_i, celui qui existe dans l_f doit être l'un d'eux. De plus on évitera de créer des murs qu'on supprime ensuite...*

5. Extensions possibles

En bonus, une des idées ci-dessous peut être développée. Attention les pistes proposées mènent à des développements de longueur et de difficulté variables.

- Modifier la fonction d'exploration du labyrinthe pour qu'elle ne renvoie pas seulement un tableau indiquant les cases sur le chemin retenu, mais plutôt une liste des cases de ce chemin. Connaître les cases dans l'ordre pourrait permettre d'afficher le chemin pas à pas : d'abord la première case seulement, puis les deux premières, et ainsi de suite jusqu'à atteindre la sortie.
- Prendre en compte des coûts de construction des murs. Ainsi on cherche un labyrinthe parfait de coût minimal (ce qui correspond à chercher un arbre couvrant de poids maximal). On peut pour cela adapter l'algorithme de Kruskal, ce qui nécessite en particulier d'implémenter un tri. De même que le graphe est ici implicite, le coût pourrait aussi être implicite, et être calculé à partir d'informations portées par les cases. On peut imaginer différents types de cases (ville, bois, château, campagne...) et décider d'une fonction de coût de construction d'un mur en fonction des types de cases qu'il sépare.
- Imaginer un labyrinthe torique : comme si le plateau n'avait pas de bords, ou plutôt que les bords sont recollés, comme dans le jeu snake : quand on arrive tout à gauche du labyrinthe et qu'on se déplace encore d'un cran vers la gauche, on arrive sur la case tout à droite à la même hauteur. De même si on est tout en haut et qu'on se déplace encore vers le haut, on arrive tout en bas. Cela revient à considérer non pas l'ensemble des cases comme $[\![0, h[\![\times [\![0, w[\![$, mais comme $\mathbb{Z}/h\mathbb{Z} \times \mathbb{Z}/w\mathbb{Z}$. Dans cette hypothèse les murs d'enceinte ne sont plus montés par convention : il faut décider si on mure ou non ces frontières entre deux cases adjacentes pour la nouvelle relation d'adjacence considérée.

Générer un nombre aléatoire en C

Pour générer un nombre pseudo-aléatoire en C, on utilise la fonction `rand`. Celle-ci est codée dans la librairie standard, qu'il faut donc inclure. Cette fonction est un générateur, à chaque appel au cours d'une exécution elle donne un nouvel entier, la suite des entiers ainsi obtenus étant une suite de nombres pseudo-aléatoires. Cependant, utilisée telle quelle, cette fonction a le même comportement à chaque exécution. En commentant la ligne 6 du programme ci-contre avant compilation, on peut observer qu'il s'exécute toujours de la même manière.

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<time.h>
4
5  int main(){
6    srand(time(NULL));
7    for(int i=0;i<=20;i++){
8      printf("%d\n", rand());
9    }
10   return 0;
11 }
```

Pour pallier ce problème, il faut initialiser cette suite avec une nouvelle valeur à chaque exécution. Pour cela, on utilise généralement la fonction `time` de la librairie `time.h` qui renvoie le nombre de secondes écoulées depuis le 01/01/1970 à 00h00mm00s.

Les entiers générés simulent une loi uniforme sur $[\![0, N]\!]$ avec $N =$ `RAND_MAX`, où `RAND_MAX` est une constante supérieure à $2^{15} - 1$ qu'on peut utiliser (et afficher par curiosité). Pour obtenir des entiers de $[\![0, p-1]\!]$, on pourra prendre les valeurs obtenues modulo p, et pour obtenir des entiers de $[\![a, b]\!]$, on pourra prendre les valeurs obtenues modulo $p = b - a + 1$ puis leur ajouter a, même si ces opérations ne préservent pas tout à fait l'uniformité du tirage.

Exercice Écrire un programme qui affiche la proportion d'entiers plus petits que `RAND_MAX`$/2$ pour 1000 tirages effectués avec la fonction `rand`.