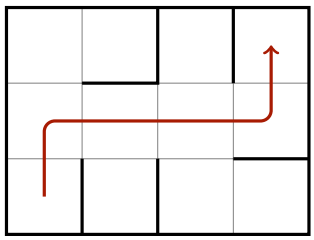
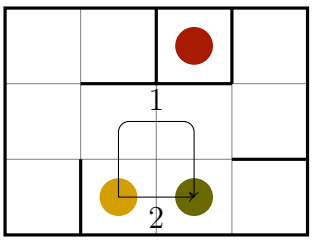
Exercice 1 : Création et exploration de labyrinthe

Dans cet exercice nous nous intéresserons à la création et à la résolution de labyrinthes dit *parfaits*. Un labyrinthe est dit parfait dès lors qu'entre toute paire de deux cases il existe un et un seul chemin dans le labyrinthe.
Ci-dessous deux exemples de labyrinthes. Le labyrinthe 1a est un exemple de labyrinthe parfait, vous pouvez 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 et chemin 2).
Dans toute la suite nous considérerons que l'entrée du labyrinthe se trouve dans la case en bas à gauche (coordonnée $(0, 0)$) et la sortie dans le coin en haut à droite (coordonnée $(\text{hauteur} - 1, \text{largeur} - 1)$). Ainsi sur la figure 1a, vous pouvez voir le chemin solution en $\longrightarrow$.



(a) Labyrinthe parfait



(b) Labyrinthe non parfait

FIGURE 1 – Exemples de labyrinthes

Dans `moodle` vous trouverez un bout de code définissant en OCaml quelques fonctions permettant la manipulation de labyrinthes. Vous trouverez notamment la définition d'une case comme contenant 2 booléens indiquant la présence ou l'absence de murs dans deux des quatre directions cardinales. Vous trouverez aussi la définition d'un type labyrinthe représentant un labyrinthe au moyen d'une matrice de case. On pourra dessiner un labyrinthe `laby` au moyen de l'appel à la fonction `draw_laby (laby: labyrinthe): unit`, on pourra appeler `draw_laby` avec les arguments optionnels suivants :

- `draw_laby ~visited:(Some(a)) laby` où `a` est une matrice de booléen de même taille que `laby` et indiquant si une case a été visitée. Le labyrinthe sera alors affiché en marquant d'un point rouge les cases déjà visitées.
- `draw_laby ~path:(Some(p)) laby` où `p` est une liste de coordonnées représentant un chemin dans le labyrinthe. Ce chemin sera représenté sur le labyrinthe.

Il existe plusieurs types d'algorithmes générant des labyrinthes parfaits. Nous en verrons deux dans ce TP, et si le temps le permet deux autres dans des TPs ultérieurs.

1. Boite à outils

Q. 1 Définir une fonction `init_full_laby (height: int) (width: int) : labyrinthe` fabriquant un labyrinthe complètement muré.

Q. 2 Définir une fonction `neighbours (i: int) (j: int) (laby: labyrinthe) : (int * int) list` produisant la liste des voisins (nord, sud, est, west) d'une case (i, j). On ne fera pas attention ici à la présence ou à l'absence de mur.

Q. 3 Définir une fonction `destroy_wall (i, j) (i', j') (laby: labyrinthe): unit` détruisant le mur se trouvant entre les cases `(i, j)` et `(i', j')`, on pourra supposer que ces deux cases sont adjacentes.

Q. 4 Définir une fonction `accessible (i, j) (i', j') (laby: labyrinthe): bool` permettant de tester l'existence d'un mur entre les cases `(i, j)` et `(i', j')`.

2. Création de labyrinthes parfaits version 1

Considérons un premier algorithme de génération du labyrinthe par exploration exhaustive du labyrinthe. L'algorithme est décrit ci-dessous.

- On crée un labyrinthe dont tous les murs sont fermés et dont aucune case n'a été visitée,
- On crée une pile (initialement vide) destinée à contenir l'ensemble des cases à traiter par l'algorithme.
- On choisit une case au hasard dans le labyrinthe. On modifie alors l'état de cette case pour indiquer qu'elle est visitée et on l'empile dans la pile des cases à traiter.
- Tant que la pile est non vide, on répète la suite des instructions ci-dessous.
  - On dépile la case au sommet de la pile.
  - On détermine la liste de ses cases adjacentes non visitées. Si elle a au moins une voisine non visitée, on rempile la case, on en choisit une voisine au hasard, on modifie son état pour indiquer qu'elle est désormais visitée, on "casse" le mur entre les deux cases et on empile la nouvelle case dans la pile des cases à traiter.

Q. 5 Implanter l'algorithme décrit ci-dessus dans une fonction `generate_laby2 (height: int) (width: int): labyrinthe`.

Q. 6 Dans un deuxième temps (comprendre après le TP, sur votre temps libre) : Implanter l'algorithme décrit ci-dessus en remplaçant l'utilisation de la pile par des appels récursifs.

3. Création de labyrinthes parfaits version 2

On propose une deuxième version de l'algorithme de génération de labyrinthe qui est basée sur une marche aléatoire. On initialise le labyrinthe avec tous les murs et ceux-ci sont détruits par des marches aléatoires successives.

- On maintient une liste des cases déjà visités, initialisées à une case choisie au hasard dans le labyrinthe.
- Tant qu'il reste des cases non encore visitée :
  - On choisit au hasard une case non encore visitée
  - On effectue une marche aléatoire, traversant les murs, ne repassant pas par elle-même dans le labyrinthe, jusqu'à tomber sur une case visitée.
  - On détruit tous les murs le long de la marche aléatoire, et on marque toutes ses cases comme visitées.

La figure 2 représente en rouge les cases déjà visités du labyrinthe en jaune la marche aléatoire qui est ajoutée au labyrinthe.
Afin de faire une marche aléatoire ne repassant pas par elle-même on pourra représenter la marche aléatoire par une matrice contenant des coordonnées ou `None`. Si une case contient une coordonnée c, cette case fait partie de la marche aléatoire et son prédécesseur dans la marche est la case c. Aussi pendant la marche aléatoire on vérifie qu'on ne passe que par des cases contenant `None` dans la matrice représentant la marche. Si ce n'est pas le cas on supprimera la boucle ainsi créée de la marche aléatoire. La première case de la marche pourra être repérée en stockant les coordonnées $(-1, -1)$ dans la matrice de la marche.

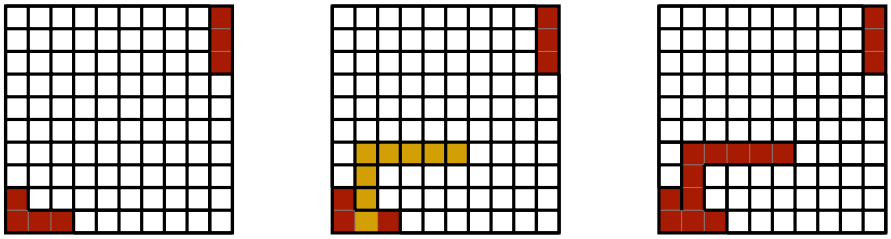


FIGURE 2 – Ajout d'une marche aléatoire

Q. 7 Implanter l'algorithme ci-dessus. On pourra commencer par implanter une fonction `walk (visited : bool array array) (laby: labyrinthe)` prenant en argument une matrice de booléen représentant les cases du labyrinthe (le deuxième argument) qui ont déjà été visitées, faisant une marche aléatoire sans boucle et cassant les murs de `laby` le long de la marche avant d'atteindre `visited`.

Exercice 2 : 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 bas à gauche (coordonnée $(0, 0)$) et la sortie dans le coin en haut à droite (coordonnée $(\text{hauteur} - 1, \text{largeur} - 1)$).

1. Recherche par pile

L'algorithme suivant utilise l'idée de l'algorithme de création de labyrinthe version 1. À la place de maintenir une pile des cases "ouvertes", on maintient une pile contenant des paires : cases "ouvertes" $\times$ chemin ayant mené à cette case. Par exemple la pile pourrait contenir : `((0, 0), [(0, 0)])` et `((0, 1), [(0, 1); (0, 0)])` indiquant que les cases `(0, 0)` et `(0, 1)` ont été visitées, mais aussi que le chemin suivi pour atteindre la case `(0, 1)` est : `[(0, 0); (0, 1)]` (on remarque que le chemin sera stocké à l'envers afin de pouvoir aisément empiler de nouveaux éléments).

- On crée une pile (initialement vide) des cases visitées.
- On empile les coordonnées de l'entrée du labyrinthe *ainsi que le chemin y menant depuis l'entrée*, et on marque la case comme visitée.
- Tant que la sortie n'a pas été trouvée, on dépile le sommet s de la pile, et on examine ses voisins accessibles et non encore visités :
  - S'il n'existe pas de tels voisin, on est dans une impasse, on recommence la boucle
  - Sinon on choisit aléatoirement un de ces voisins v,
    - Si ce voisin est la sortie, c'est gagné
    - Sinon on le marque comme visité et on empile s et v *en les appairant avec les bons chemins depuis l'entrée du labyrinthe*.

Q. 1 Implanter l'algorithme ci-dessus dans une fonction `solve_laby (laby: labyrinthe) : (int * int) list`

Q. 2 L'algorithme précédent effectue un parcours en profondeur. Modifier le pour faire un parcours en largeur du labyrinthe.