TP : Graphes et labyrinthes

I. Fichier source

I.a. Récupération du fichier

Attention, lors de cette séance, vous écrirez votre code dans un nouveau fichier (et non à la suite du fichier source).

Commencez par récupérer le fichier source disponible sur le support en ligne du cours, et enregistrez-le sur le Bureau.

Créez un nouveau fichier au même endroit que le fichier source (toujours sur le Bureau).

I.b. Utilisation

Pour pouvoir utiliser les fonctions du fichier source dans votre nouveau fichier, il vous suffit de taper la ligne de code suivante au début de votre nouveau fichier :

```
#use "laby.ml"
```

En plus d'alléger votre fichier de travail, cela simplifie grandement l'évaluation des fonctions mises à votre disposition.

I.c. Fonctions disponibles

Voici un extrait des fonctions mises à votre disposition pour ce TP :

- `tracer_laby lab` vous permet de tracer le labyrinthe `lab`
- `creer_laby_avance l h p` crée un labyrinthe de largeur `l`, de hauteur `h` et avec un pourcentage `p` de murs (sans garantie sur la solvabilité du labyrinthe)
- `creer_laby_parfait l h` crée un labyrinthe parfait de largeur `l` et de hauteur `h`
- `creer_laby_presque_parfait l h p` crée un labyrinthe parfait de largeur `l` et de hauteur `h`, puis retire un pourcentage `p` de murs dans ce labyrinthe (afin qu'il y ait plus d'un chemin possible entre deux cases)
- `voisins lab i j` renvoie la liste des voisins de la case `(i,j)` dans le labyrinthe `lab`
- `resoudre_laby lab` lance le mode "résolution d'un labyrinthe"
- `resoudre_laby2 lab` lance le mode "résolution d'un labyrinthe", avec en plus la possibilité de chercher automatiquement le plus court chemin vers la sortie

Attention, si vous tentez d'utiliser la fonction `resoudre_laby2` sur un labyrinthe qui n'a pas de solution (ce qui peut arriver avec la fonction `creer_laby_avance`), votre programme risque de planter ou de tourner en boucle.

II. Cases accessibles

II.a. En un coup

Quelles sont les cases accessibles, dans le labyrinthe `lab`, à partir de la case `(i,j)`, en un coup ?

II.b. En au plus deux coups

Imaginez une fonction `cases_ccessibles_2_coups_au_plus` :

- prenant en argument un labyrinthe `lab` et les coordonnées `(i,j)` d'une case
- renvoyant une liste de coordonnées
- telle que `cases_accessibles_2_coups_au_plus lab (i,j)` renvoie la liste des cases accessibles dans le labyrinthe `lab`, en partant de la case `(i,j)`, en au plus deux coups.

Indice : vous pouvez utiliser la fonction `List.concat`, qui concatène plusieurs listes en une seule (`List.concat [ l1 ; l2 ; l3 |]` créée une nouvelle liste qui contient tous les éléments de `l1`, puis tous les éléments de `l2`, puis tous les éléments de `l3`).

II.c. En exactement deux coups

Imaginez une fonction `cases_accessibles_2_coups` :

- prenant en argument un labyrinthe `lab` et les coordonnées `(i,j)` d'une case
- renvoyant une liste de coordonnées
- telle que `cases_accessibles_2_coups lab (i,j)` renvoie la liste des cases accessibles dans le labyrinthe `lab`, en partant de la case `(i,j)`, en exactement deux coups.

Indice : vous pouvez commencer par implémenter une fonction `supprimer_elements` telle que `supprimer_elements l1 l2` renvoie une nouvelle liste constituée des éléments de `l1` n'apparaissant pas dans `l2`.

II.d. En autant de coups que nécessaire

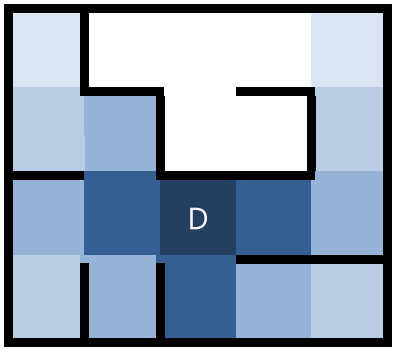
Imaginez une fonction `cases_accessibles` :

- prenant en argument un labyrinthe `lab` et les coordonnées `(i,j)` d'une case
- renvoyant une liste de coordonnées
- telle que `cases_accessibles lab (i,j)` renvoie la liste des cases accessibles dans le labyrinthe `lab`, en partant de la case `(i,j)`, et en autant de coups que nécessaire.

Indice 1 : Il pourra être pratique de gérer :

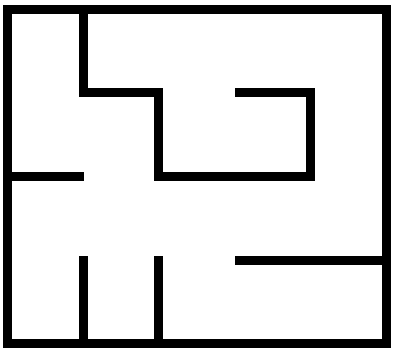
- une liste des cases à visiter
- une liste des cases déjà visitées (sous la forme d'un tableau à deux dimensions contenant des booléens)

Indice 2 : Inspirez-vous du dégradé du labyrinthe ci-contre pour l'ordre dans lequel vous devez traiter les cases (la lettre "D" marquant la case de départ).

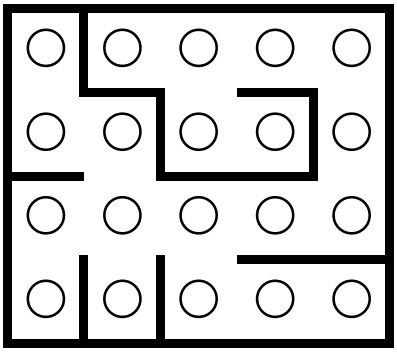


Indice 3 : Vous pouvez également essayer d'essayer de deviner ce que devra faire votre algorithme en vous entrainant sur l'exemple ci-dessous :

Labyrinthe



Cases déjà visitées



Cases à visiter

II.e. Test de faisabilité

Imaginez une fonction `est_accessible` :

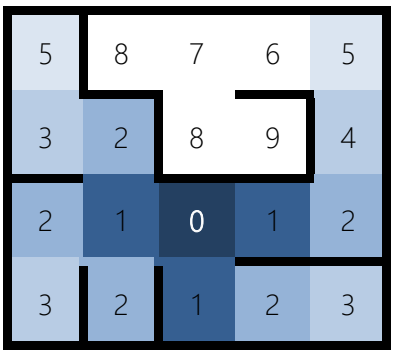
- prenant en argument un labyrinthe `lab` et les coordonnées `(i1,j1)` et `(i2,j2)` de deux cases de ce labyrinthe
- renvoyant un booléen
- telle que `est_accessible lab (i1,j1) (i2,j2)` renvoie `true` si la case `(i2,j2)` est accessible dans le labyrinthe `lab` en partant de la case `(i1,j1)`, et `false` sinon.

III. Distances

Dans cette partie, on se concentrera dans un premier temps sur le cas des labyrinthes parfaits : il existe donc un seul et unique chemin entre tout couple de cases.

Nous verrons ensuite comme gérer également le cas des labyrinthes presque parfaits.

III.a.Objectif



L'objectif de cette partie est de calculer une carte des distances, c'est-à-dire un tableau donnant, pour chacune des cases d'un labyrinthe, la distance qui la sépare d'une case de "départ".

Observez l'exemple ci-contre : que remarquez-vous ?

Comment pourriez-vous modifier le code imaginé dans la partie précédente pour créer cette carte des distances ?

III.b. Implémentation

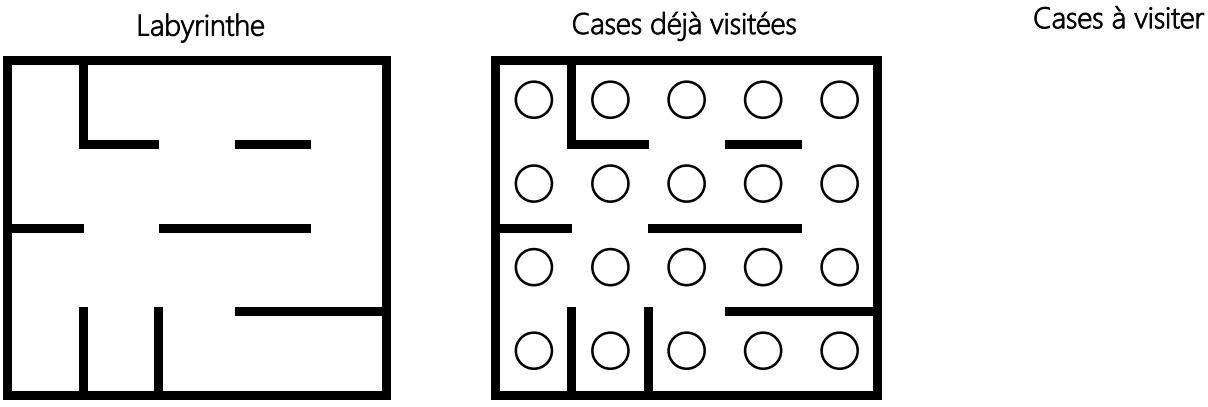
Imaginez une fonction `carte_des_distances` :

- prenant en argument un labyrinthe `lab` (qu'on suppose parfait) et les coordonnées `(i,j)` d'une case de départ
- renvoyant un tableau d'entiers à deux dimensions
- telle que `carte_des_distances lab (i,j)` renvoie la carte des distances du labyrinthe `lab` avec pour case de départ la case `(i,j)`.

Indice : pour créer un tableau d'entiers à deux dimensions, de largeur `l` de hauteur `h`, initialisé avec la valeur `v`, utilisez la ligne de code `creer_tableau l h v`

III.c. Cas des labyrinthes presque parfaits

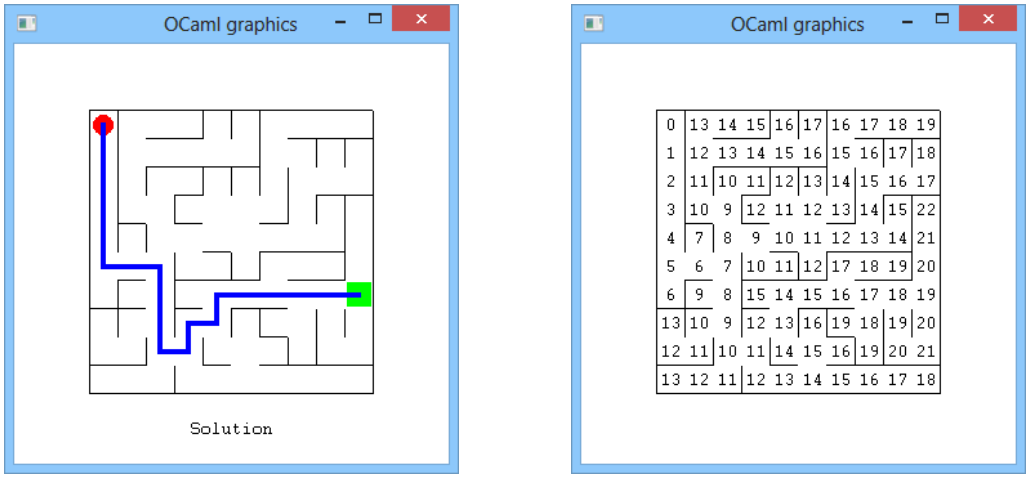
La fonction écrite au point précédent fonctionne-t-elle également pour les labyrinthes presque parfaits ? Peut-on traiter les cases à visiter dans n'importe quel ordre ?
Pour justifier votre réponse, vous pourrez vous inspirer de l'exemple ci-dessous :



Si besoin est, modifiez votre fonction `carte_des_distances` pour gérer également le cas des labyrinthes presque parfaits.

IV. Plus court chemin entre deux points

Observez la carte des distances et le plus court chemin ci-dessous. Que remarquez-vous ?



Comment pourrait-on modifier l'algorithme de calcul de la carte des distances pour obtenir le trajet le plus court entre deux points dans un labyrinthe ?

Imaginez une fonction `trouver_solution` :

- prenant en argument un labyrinthe `lab` (qu'on suppose parfait), les coordonnées `(i1,j1)` d'une case de départ et les coordonées `(i2,j2)` d'une case d'arrivée
- renvoyant une liste de coordonnées
- telle que `trouver_solution lab (i1,j1) (i2,j2)` renvoie la liste des cases à parcourir pour aller de la case `(i1,j1)` à la case `(i2,j2)` en un minimum de coups, dans le labyrinthe `lab`.

Indice 1 : Stocker pour chaque point `(i,j)` les coordonnées de son "prédécesseur" `(iP,jP)`, c'est-à-dire le point `(iP,jP)` à partir duquel on a atteint le point `(i,j)` pendant la phase d'exploration du labyrinthe.

Indice 2 : Une fois qu'on a obtenu les coordonnées de tous les prédécesseurs, il suffit de partir de l'arrivée et de remonter la "trace" que constituent ces prédécesseurs pour revenir jusqu'au départ.