



TP : Labyrinthe

2026-2027

Thibaud Lemanissier

PC Lycée Carnot



1. Principe du TP

L'objectif est de créer un labyrinthe et de donner la solution de ce labyrinthe :

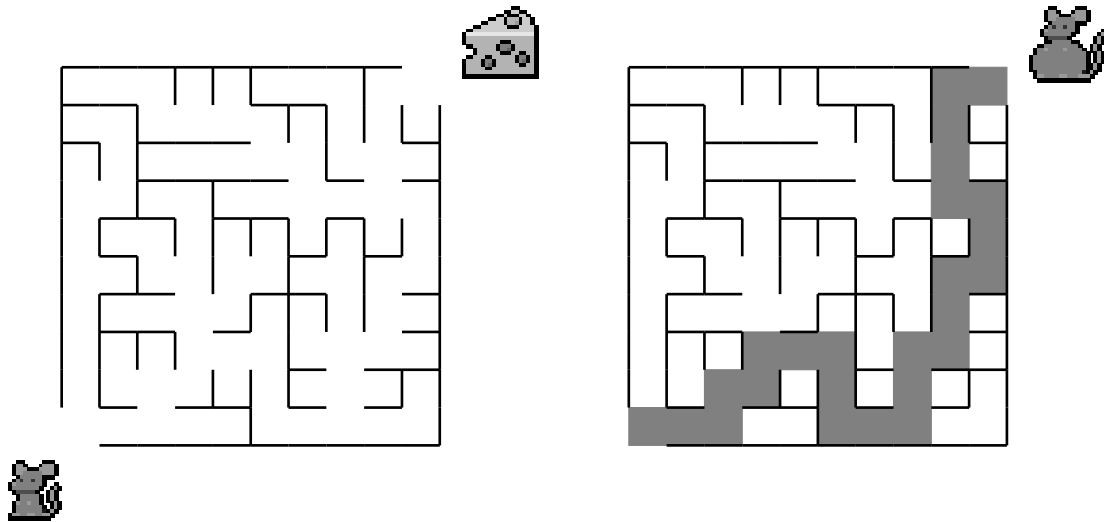


FIGURE 1 – À gauche un labyrinthe et à droite ce même labyrinthe avec la chemin le plus court de l'entrée à la sortie.

On rappelle qu'un **arbre** $\mathcal{G} = (\mathcal{S}, \mathcal{A})$ est un graphe vérifiant l'une des conditions équivalentes suivantes :

- (i) $\mathcal{G}$ est connexe et acyclique;
- (ii) $\mathcal{G}$ est connexe et $\text{card}(\mathcal{A}) = \text{card}(\mathcal{S}) - 1$;
- (iii) $\mathcal{G}$ est acyclique et $\text{card}(\mathcal{A}) = \text{card}(\mathcal{S}) - 1$.

Pour créer notre labyrinthe, nous allons créer un arbre dont la relation avec le labyrinthe est synthétisée dans la figure suivante :

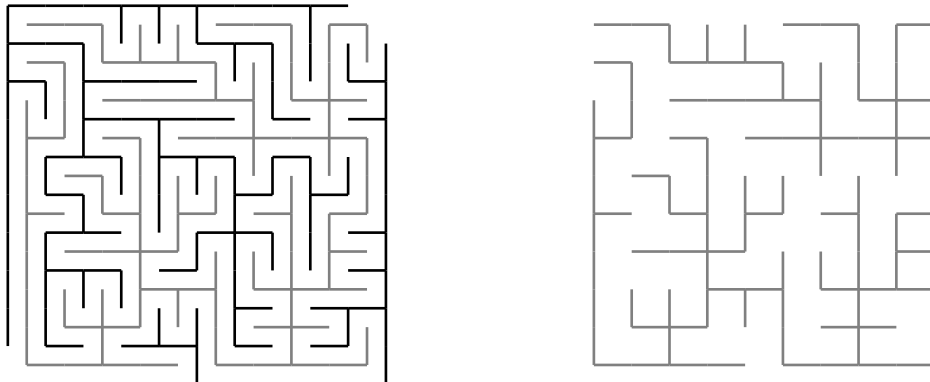


FIGURE 2 – À gauche une représentation du labyrinthe avec en gris l'arbre correspondant et à droite une représentation de l'arbre uniquement.

Pour que le labyrinthe soit intéressant, nous allons construire cet arbre de sorte à ce que le chemin soit long et il y ait régulièrement des embranchements.

2. Construction du Réseau

Pour construire cet arbre nous allons commencer par créer le graphe dont les sommets sont les couples $(i, j) \in \llbracket 0, n-1 \rrbracket^2$ et les sommets adjacents à (i, j) sont les sommets $(i-1, j), (i, j-1), (i+1, j), (i, j+1)$ à condition que les coordonnées de ces points soient inclus dans $\llbracket 0, n-1 \rrbracket^2$. Dans ce graphe et pour la suite du TP $(0, 0)$ désignera l'entrée et $(n-1, n-1)$ la sortie du labyrinthe et on notera `res` ce graphe.

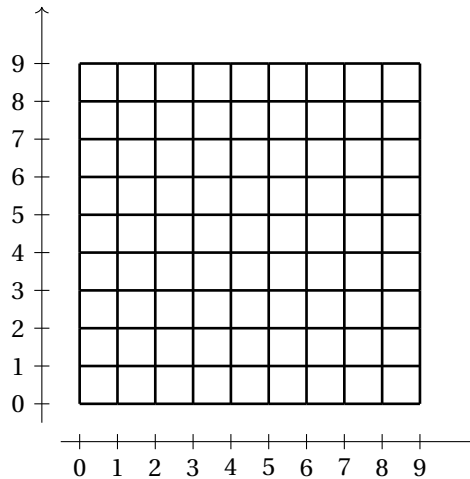


FIGURE 3 – Représentation du graphe.

1. Écrire une fonction `voisin(i, j, n)` qui prend en arguments les coordonnées d'un point (i, j) et la taille d'un côté du réseau n et qui renvoie la liste des sommets adjacents au sommet (i, j) dans le réseau. Attention au bord!
2. Utiliser la fonction précédente pour écrire une fonction `reseau(n)` qui prend en argument un entier n et qui renvoie le dictionnaire correspondant au réseau $n \times n$. Pour tout couple (i, j) , on utilisera la fonction `shuffle` (du module `numpy.random` importé avec l'alias `rd`) sur la liste des sommets adjacents (i, j) de sorte à ce qu'ils soient rangés aléatoirement.

Les premiers termes du dictionnaire vont donc être de la forme

$$\{(0,0) : [(0,1), (1,0)], (1,0) : [(1,1), (0,1), (2,0)], \dots\}.$$

3. Écrire une fonction `affichage(d)` qui prend en argument un dictionnaire représentant un graphe et qui affiche les arêtes correspondantes. On pourra utiliser la bibliothèque `matplotlib.pyplot` (importé avec l'alias `plt`).

Pour rappel, pour tracer un segment entre deux points $(x1, y1)$ et $(x2, y2)$, on peut écrire :

```
1 plt.plot([x1,x2], [y1,y2], "k") # "k" sert à tracer le segment en noir
```

Avant d'afficher le graphe (`plt.show()`), on ajoutera les deux lignes :

```
1 plt.axis('equal')
2 plt.axis('off')
```

pour avoir une représentation orthonormée et ne pas faire apparaître les axes.

3. Construction de l'arbre couvrant

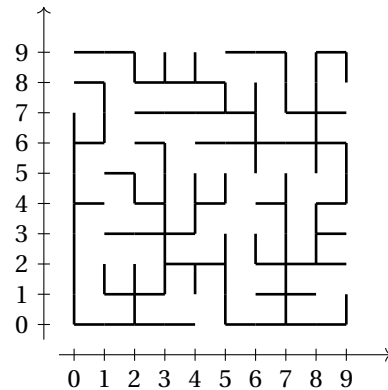
Nous allons maintenant pouvoir construire un arbre aléatoire couvrant de ce graphe, c'est-à-dire un sous graphe de ce graphe qui est un arbre et qui contient tous les sommets.

Plus précisément, nous allons construire un arbre **enraciné**, c'est-à-dire un graphe orienté acyclique tel que chaque sommet possède exactement un parent sauf un qui sera appelé la racine.

Un arbre enraciné sera codé dans un dictionnaire dont les clefs seront les sommets et la valeur associée à chaque clef sera le parent de ce sommet sauf pour la racine dont la valeur associée sera None.

Par exemple, l'arbre ci-contre correspond au dictionnaire :

$\{(0,0) : \text{None}, (1,0) : (0,0), (1,0) : (0,0), (2,0) : (1,0), (2,1) : (2,0), (1,1) : (2,1), \dots\}$



4. S'inspirer de la question 3 pour écrire une fonction `affichage_arbre(a)` qui prend en argument un dictionnaire représentant un arbre tel que décrit précédemment et qui affiche les arêtes correspondantes.

Pour construire cet arbre, nous allons procéder en deux temps :

- construire un chemin allant de la case $(0,0)$ jusqu'à la case $(n-1, n-1)$;
- construire le reste des branches de l'arbre en partant du plus proche de la racine.

La première étape sera réalisée par un parcours en profondeur c'est-à-dire on prend systématiquement le dernier de la file d'attente, la seconde sera réalisée par un parcours en largeur c'est-à-dire on prend systématiquement le premier de la file d'attente.

3.1. Construction de chemin de l'entrée à la sortie

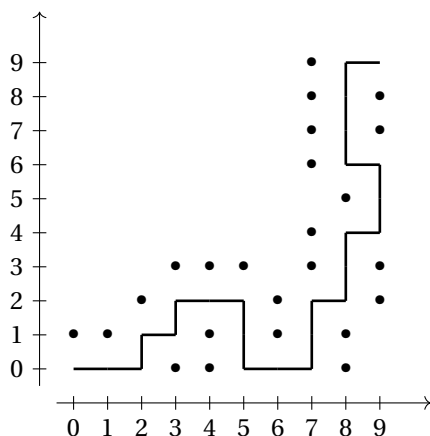
Pour construire le chemin entre l'entrée et la sortie, nous allons procéder à un parcours en profondeur du graphe en partant du sommet $(0,0)$ pour créer un arbre jusqu'à ce que `a` contienne le sommet $(n-1, n-1)$. On notera `L` la file d'attente qui contiendra les sommets qui n'ont pas encore été visités et qui sont adjacents à des sommets de `a`.

Plus précisément, nous allons procéder de la manière suivante :

- On construit un dictionnaire `a = {(0,0) : None}` qui va contenir et la liste des sommets qui sont adjacents à $(0,0)$, `L = res[(0,0)]` qui va servir de file d'attente des sommets à ajouter à l'arbre;
- On enlève `s` le dernier sommet de la file dans la file d'attente `L`, on ajoute les sommets adjacents à `s` qui ne sont ni dans `a` ni dans `L` à `L` et on ajoute la clef `s` à `a` avec pour valeur associée un des sommets de `a` qui lui est adjacent.

On répète le deuxième point jusqu'à ce que le dictionnaire `a` contienne le sommet $(n-1, n-1)$.

5. Expliquer pourquoi l'algorithme décrit précédemment s'arrête.
6. Compléter la fonction `chemin(res, n)` suivante qui prend en argument le graphe que renvoie la fonction `reseau` et un entier `n` et qui renvoie un arbre `a` correspondant à un chemin reliant l'entrée à la sortie ainsi que la file d'attente `L` qui contiendra les sommets qui n'ont pas encore été visités et qui sont adjacents à des sommets de `a`.



```

1 def chemin(res, n) :
2     a = ...
3     L = ...
4     while ... :
5         s = ...
6         for sp in res[s] :
7             if sp not in a and sp not in L :
8                 ...
9             if ... :
10                a[s] = sp
11     return ...

```

FIGURE 4 – Chemin reliant l'entrée à la sortie ainsi que les voisins sous forme de point.

3.2. Complétion de l'arbre et récupération de la solution

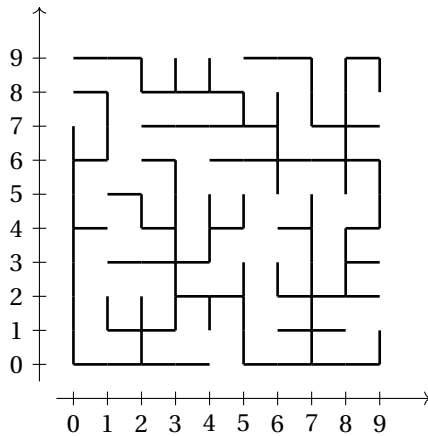
Pour compléter l'arbre, nous allons procéder à un parcours en largeur du graphe en utilisant la file d'attente L renvoyée par la fonction `chemin` pour compléter l'arbre à lui aussi renvoyé par la fonction `chemin` jusqu'à ce que L soit vide.

Plus précisément, nous allons procéder de la manière suivante :

- On récupère l'arbre a et la file d'attente L renvoyés par la fonction `chemin`;
- On enlève s le premier sommet de la file dans la file d'attente L , on ajoute les sommets adjacents à s qui ne sont ni dans a ni dans L à L et on ajoute la clef s à a avec pour valeur associé un des sommets de a qui lui est adjacent.

On réitère le deuxième point jusqu'à ce que la liste L soit vide.

7. Expliquer pourquoi l'algorithme décrit précédemment s'arrête.
8. Écrire une fonction `couvrant(n)` prenant en argument un entier n et qui applique l'algorithme décrit précédemment pour obtenir un arbre couvrant du graphe obtenu à partir de la fonction `reseau`.



```

1 def couvrant(n) :
2     res = ...
3     a, L = ...
4     while ... :
5         s = ...
6         for sp in res[s] :
7             if sp not in a and sp not in L :
8                 ...
9                 if ... :
10                    a[s] = sp
11     return ...

```

FIGURE 5 – Représentation de l'arbre couvrant.

Dans ce contexte, une solution du labyrinthe est un chemin de $(0,0)$ à $(n-1,n-1)$.

9. Écrire une fonction `solution(a)` qui prend en argument l'arbre couvrant obtenu à partir de la fonction précédente et qui renvoie un chemin de $(0,0)$ à $(n-1,n-1)$ sous la forme d'une liste.

4. Construction du labyrinthe

Nous allons maintenant pouvoir créer le labyrinthe associé à l'arbre a obtenu grâce à la fonction `couvrant`. Pour cela, nous allons créer un graphe non orienté (sous la forme d'un dictionnaire) `labyrinthe` dont les sommets sont les points de la forme $(i - 1/2, j - 1/2)$ avec $(i, j) \in \llbracket 0, n \rrbracket^2 \setminus \{(0,0), (n,n)\}$.

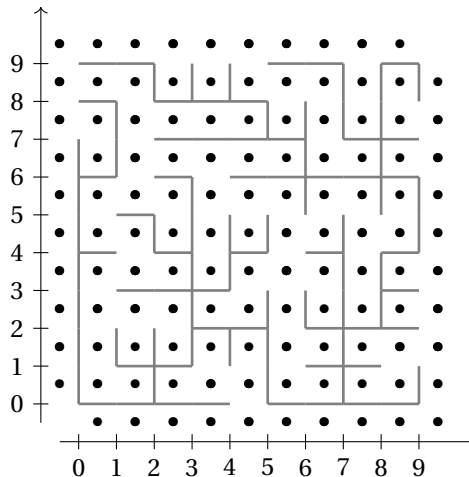


FIGURE 6 – L'arbre a obtenu à partir de la fonction `couvrant` et des sommets (les points) du graphe `labyrinthe`.

Les arêtes sont déterminées par les règles suivantes :

- Le sommet $(i - 1/2, j - 1/2)$ ne peut être adjacent qu'à l'un des sommets qui lui est le plus proche;
- Le sommet $(i - 1/2, j - 1/2)$ et $(i - 1/2, j + 1/2)$ sont adjacents dans le labyrinthe si, et seulement si, (i, j) n'est pas adjacent à $(i, j - 1)$;
- Le sommet $(i - 1/2, j - 1/2)$ et $(i + 1/2, j - 1/2)$ sont adjacents dans le labyrinthe si, et seulement si, (i, j) n'est pas adjacent à $(i - 1, j)$.

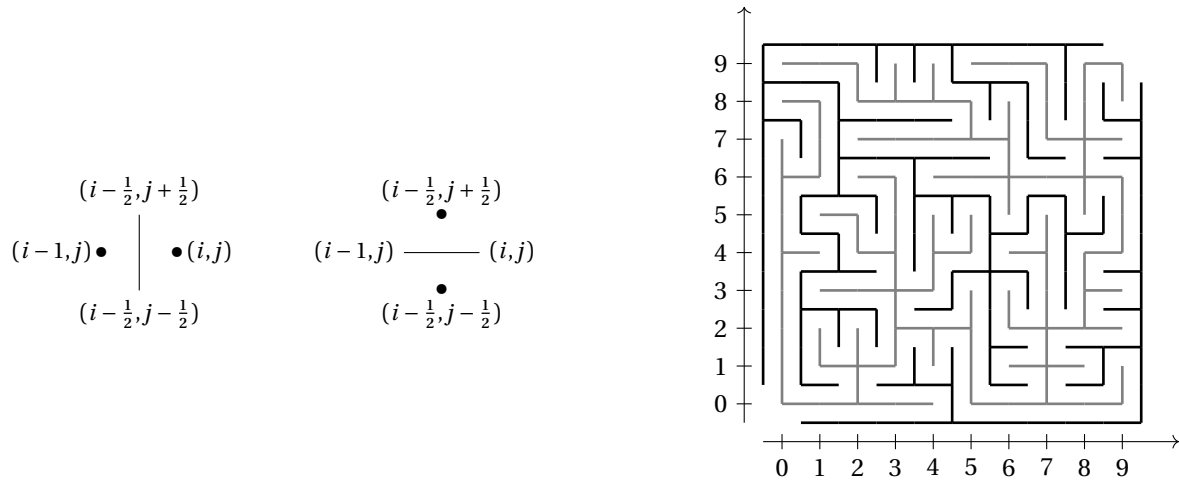


FIGURE 7 – Les cas possibles où les points désignent les sommets non reliés (à gauche) ainsi que le résultat du labyrinthe (en noir) superposé à l'arbre (en gris)

10. Écrire une fonction `labyrinthe(n)` qui prend en argument un entier `n` qui crée un arbre `a` à l'aide de la fonction `couvrant` et qui renvoie le dictionnaire `labyrinthe` décrit précédemment.
11. Compléter la fonction suivante pour qu'elle affiche la solution du labyrinthe à la manière de :

```

1 from matplotlib.patches import Rectangle
2
3 def affiche_sol(labyrinthe, a, n) :
4     fig = plt.figure()
5     ax = fig.add_subplot()
6     ... # tracer du labyrinthe
7     ...
8     ...
9     c = solution(a, n)
10    for e in c :
11        rec = Rectangle(...)
12        ax.add_patch(rec)
13    plt.axis("equal")
14    plt.axis("off")
15    plt.axis(xmin=-3/2, xmax=n+1/2, ymin=-3/2, ymax = n+1/2)
16    plt.show()

```

La fonction `Rectangle` prend en argument les coordonnées du sommet en bas à gauche du rectangle, la largeur et la hauteur du rectangle ajoutera l'argument optionnel `color = "gray"` pour fixer la couleur du rectangle affiché.

5. Rappels

5.1. Quelques opérations sur les listes

On rappelle quelques opérations sur les listes qui nous seront utiles pour le TP :

Instruction Python	Opération effectuée
<code>L = []</code>	création d'une liste vide
<code>L[i]</code>	accéder au <i>i</i> ème élément de la liste
<code>L[-1]</code>	accéder au dernier élément de la liste
<code>L[i]=x</code>	remplacer la valeur en position <i>i</i> de la liste par <i>x</i>
<code>len(L)</code>	renvoie le nombre d'entrée du dictionnaire
<code>L.append(x)</code>	ajouter l'élément <i>x</i> à la fin de la liste
<code>x = L.pop(i)</code>	enlève et stock dans <i>x</i> l'élément en <i>i</i> ème position de la liste
<code>x = L.pop()</code>	enlève et stock dans <i>x</i> le dernier élément de la liste

5.2. Quelques opérations sur les dictionnaires

Pour rappel, un dictionnaire est une structure de donnée qui se présente sous la forme suivante :

$$\text{dico} = \{x_1 : v_1, x_2 : v_2, \dots, x_n : v_n\}$$

Les éléments situés à gauche des « : » s'appellent les clefs, les éléments situés à droite s'appellent les valeurs. Les clefs peuvent être des entiers, des flottants, des chaînes de caractères, des *n*-uplet... plus généralement des objets non mutables.

Instruction Python	Opération effectuée
<code>d = {}</code>	création d'un dictionnaire vide
<code>dico[key]</code>	accéder à la valeur associée à la clef <i>key</i>
<code>dico[key]=value1</code>	si <i>key</i> est une clef de <i>dico</i> remplace la valeur associée par <i>value1</i> si <i>key</i> n'est pas une clef de <i>dico</i> ajoute cette clef avec la valeur <i>value1</i>
<code>len(dico)</code>	Renvoie le nombre d'entrée du dictionnaire
<code>dico.keys()</code>	Renvoie la collection des clefs du dictionnaire
<code>dico.values()</code>	Renvoie la collection des valeurs du dictionnaire
<code>dico.items()</code>	Renvoie la collection des couples clefs, valeurs du dictionnaire
<code>key in dico</code>	Booléen qui teste si <i>key</i> est une clef de <i>dico</i>
<code>del(dico[key])</code>	Supprime l'entrée correspondant à la clef <i>key</i>
<code>value = dico.pop(key)</code>	Supprime l'entrée correspondant à la clef <i>key</i> et stock la valeur correspondante dans <i>value</i>