

Algorithmes sur les graphes

I Définitions et fonctions élémentaires

I.1 Définitions et vocabulaire

Un *graphe* $G = (S, A)$ est constitué d'un ensemble S de *sommets* et d'un ensemble A d'*arêtes*. Une arête a est un élément de $S \times S$, c'est à dire $a = \{x, y\}$ avec $x \in S$ et $y \in S$. Dans toute la suite, on note $n = |S|$ le nombre de nœuds et $m = |A|$ le nombre d'arêtes. La définition précédente se transpose naturellement en une représentation graphique dans laquelle chaque nœud est un point du plan et chaque arête une ligne joignant deux nœuds.

Dans la définition, il est sous-entendu qu'une paire ne prend pas en considération l'ordre dans lequel les éléments x et y sont écrits : $a = \{x, y\}$ est synonyme de $a = \{y, x\}$. On dit dans ce cas que le graphe n'est pas orienté. Mais il existe aussi des *graphes orientés* dans lesquels A n'est pas formé de paires mais de *couples* (x, y) d'éléments de S . On dit alors que (x, y) est un *arc* et on le représente avec une flèche allant de x vers y . Dans ce cas, il ne faut pas confondre (x, y) et (y, x) qui serait représenté par une flèche en sens inverse.

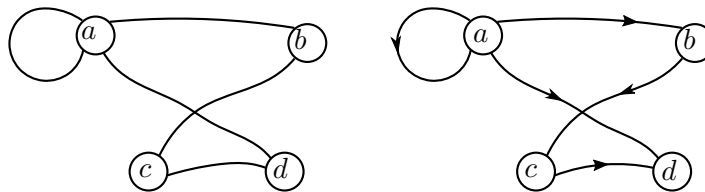


FIGURE 1 – Deux graphes possédant 4 sommets : $S = \{a, b, c, d\}$, $n = 4$. À gauche, le graphe est non orienté et possède 5 arêtes : $A = \{\{a, a\}, \{a, b\}, \{a, c\}, \{a, d\}, \{b, c\}, \{c, d\}\}$; à droite, le graphe est orienté et possède 5 arcs $A = \{(a, a), (a, b), (a, c), (a, d), (c, b), (c, d)\}$.

Le vocabulaire diffère légèrement selon que l'on parle de graphes orientés ou de graphes non orientés, mais nous ne chercherons pas la rigueur sémantique et nous travaillerons surtout avec des graphes non orientés.

- Deux sommets liés par une arête sont dits *adjacents*.
- *boucle* : arête de type $\{a, a\}$
- *graphe simple* : graphe sans boucle
- *degré* $d(x)$ d'un sommet x : nombre d'arêtes ayant x pour extrémité. Pour un graphe orienté, on peut compter séparément
 - le nombre d'arêtes qui arrivent sur x , on le nomme degré entrant et on le note $d^-(x)$;
 - le nombre d'arêtes qui sortent de x , on le nomme degré sortant et on le note $d^+(x)$.
- *chemin* (ou *chaîne*) : suite de la forme $x_0, a_0, x_1, a_1, \dots, x_{k-1}, a_{k-1}, x_k$, avec $x_i \in S$ et $a_i \in A$, dans laquelle l'arête a_i va du nœud x_i au nœud x_{i+1} . Le nombre k d'arêtes est appelé *longueur de la chaîne*. En pratique, on peut se contenter d'indiquer la suite $x_0, x_1, \dots, x_k$ des nœuds traversés, à condition qu'ils soient bien reliés les uns aux autres par des arêtes.
- Une chaîne est dite *simple* si elle ne passe pas deux fois par la même arête.
- On dit qu'une chaîne est *fermée* lorsque $x_k = x_0$.
- On appelle *cycle* (ou *circuit*) une chaîne simple et fermée.
- Un graphe est dit *connexe* s'il existe une chaîne entre toute paire de sommets.
- On appelle *arbre* un graphe connexe ne comportant pas de cycle.

De très nombreux problèmes informatiques peuvent être représentés par des graphes et il existe une foule d'algorithmes associés. Sont prévus au programme de PCSI : le parcours d'un graphe en largeur ou en profondeur, la recherche de connexité, et pour les graphes pondérés, l'algorithme de Dijkstra et l'algorithme A^* .

I.2 Représentation informatique

Il existe différentes manières de représenter un graphe en machine. Tout d'abord, les sommets sont généralement codés sous forme d'entiers : dans les exemples ci-dessus, on attribue l'indice 0 à a , l'indice 1 à b , etc, et on travaille avec ces indices plutôt qu'avec les lettres auxquelles ils correspondent. La méthode naïve qui consisterait à manipuler directement les ensembles S et A , sous forme de deux listes par exemple, n'est jamais utilisée pour des raisons d'efficacité. On lui préfère d'autres représentations, dont deux figurent à votre programme.

1. Représentation d'un graphe par une matrice d'adjacence

On définit une matrice M de taille $n \times n$ dont l'élément M_{ij} vaut 1 si les nœuds x_i et x_j sont liés par une arête, et 0 sinon. Cette matrice est symétrique pour un graphe non orienté. Pour un graphe orientée, $M_{ij} = 1$ lorsque qu'il existe un arc de i vers j . On peut typer cette matrice comme un tableau de `numpy`. La complexité spatiale d'une telle représentation est en $O(n^2)$ ce qui n'est pas toujours très efficace. En particulier, pour les graphes comportant beaucoup de nœuds et peu d'arêtes, on stocke inutilement une grande quantité de zéros.

2. Représentation d'un graphe par des listes d'adjacence

On décrit le graphe en se donnant pour chaque nœud la liste des nœuds auxquels il est relié par une arête (ou un arc). On construit donc une liste L de n listes, la liste $L[i]$ contenant les indices des nœuds adjacents au nœud x_i . La complexité spatiale d'une telle représentation est en $O(m)$. Plutôt que d'utiliser une liste, on peut aussi utiliser un dictionnaire dont les clés sont les nœuds et dont les valeurs sont les listes des voisins associés à une clé.

Pour voir si vous avez compris, écrire les matrices d'adjacence et les listes d'adjacence associées aux graphes dessinés sur la figure (1) (avec une liste de listes ou un dictionnaire de listes).

I.3 Codage de fonctions élémentaires

1. Écrire une fonction `listes_vers_matrice(L)` qui fait passer des listes d'adjacence à la matrice d'adjacence d'un graphe. La matrice renvoyée est codée comme un tableau de `numpy` bidimensionnel contenant des entiers. Initialiser d'abord un tableau de zéros par `np.zeros(dimensions, dtype = int)`. Cette fonction doit fonctionner pour des graphes orientés et non orientés.

2. Écrire une fonction `matrice_vers_listes(M)` qui fait le travail inverse. Quelle est la complexité asymptotique de ces deux fonctions ?

3. Écrire une fonction `matrice_vers_dico(M)` qui renvoie le dictionnaire représentant un graphe de matrice d'adjacence M .

4. Écrire une fonction `listes_vers_dico(L)` qui fait passer de la liste de listes d'adjacence au dictionnaire de listes d'adjacence.

5. Écrire une fonction `est_chaine(suite, M)` qui prend en argument une suite d'entiers représentant des nœuds et la matrice d'adjacence M d'un graphe. Elle renvoie un booléen qui indique s'il existe ou non une chaîne formée des nœuds énumérés dans cette suite, dans l'ordre. Il peut être commode d'utiliser ici un codage récursif.

Vous pouvez exécuter ces fonctions sur le graphe de la figure 2 (liste `L_andre` dans le fichier fourni) ou sur d'autres.

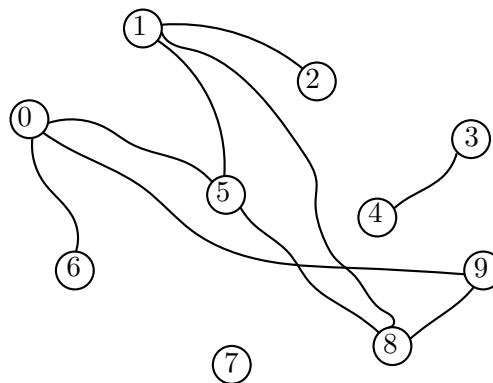


FIGURE 2 – Un graphe G_{andre} sur lequel exécuter les fonctions

II Coloration de graphe

La coloration d'un graphe consiste à colorier chacun de ses sommets de telle manière que deux nœuds adjacents quelconques présentent des couleurs distinctes. On ignore *a priori* le nombre minimal de couleurs nécessaires à ce travail, appelé *nombre chromatique* du graphe. On peut démontrer qu'il est inférieur ou égal au plus haut degré des sommets du graphe, augmenté de un.

La coloration de graphe constitue un cadre théorique permettant de traiter de nombreuses situations concrètes dans lesquelles interviennent des incompatibilités ou des conflits : planification de tâches s'excluant mutuellement, attribution de fréquences à des antennes relais en évitant les interférences, etc. Voici un exemple.

Stockage de produits chimiques

Un laboratoire possède 8 produits chimiques dont certains, s'ils sont mis en contact avec d'autres, font courir un risque d'explosion ou de dégagement de gaz toxiques. Pour les stocker en toute sécurité, on doit donc les ranger de manière que deux produits incompatibles ne soient pas dans la même armoire. Le problème est de savoir combien d'armoires sont nécessaires et comment il faut y répartir les produits.

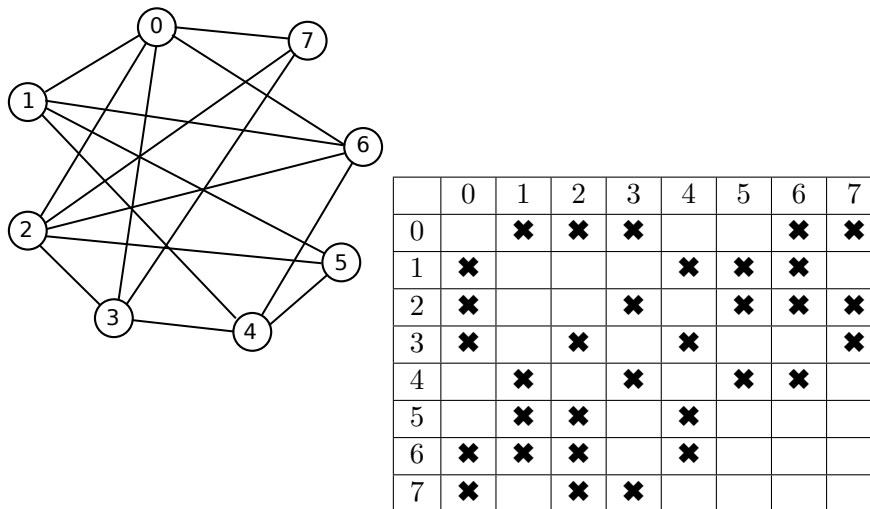


FIGURE 3 – Graphe d'incompatibilité de produits chimiques

Les incompatibilités entre produits sont indiquées dans le tableau de la figure 3. Considérons le graphe dont les sommets correspondent aux produits chimiques, numérotés de 0 à 7, et dont le tableau ci-dessus s'apparente à la matrice d'adjacence : on place une arête entre deux sommets lorsque le stockage conjoint des deux produits chimiques associés est interdit. On obtient ainsi le *graphe d'incompatibilité* de la figure 3. Maintenant, associons une couleur à chaque armoire : la première armoire est bleue, la seconde rouge, etc. Déterminer le rangement des produits dans les armoires équivaut à colorier chacun des sommets du graphe d'incompatibilité. Par exemple, colorier le sommet 0 en bleu équivaut à ranger le produit 0 dans la première armoire.

Algorithme glouton

Les couleurs sont codées comme des entiers positifs 0, 1, 2, etc. Une manière naïve de colorier un graphe consiste à utiliser l'algorithme suivant. On traite les nœuds un par un, dans un ordre quelconque. Pour chacun d'eux, on essaye toutes les couleurs dans l'ordre 0, 1, 2, etc, et on retient la première couleur qui n'entre pas en conflit avec ses voisins déjà coloriés. Il s'agit d'un *algorithme glouton* : on choisit sur l'instant une couleur favorable, puis on ne revient plus sur ce choix même si on s'aperçoit plus tard que ce n'était pas optimal.

1. Appliquer à la main l'algorithme sur le graphe de la figure 4.
2. Lors du déroulement de l'algorithme, la couleur de chacun des sommets du graphe sera indiquée dans la liste `Couleur` : `Couleur[i]` est la couleur du sommet i . On considère une couleur particulière c et une liste de sommets `liste_de_sommets`. Coder la fonction `est_deja_prise(c, liste_de_sommets, Couleur)` qui renvoie un booléen indiquant si un des sommets énumérés dans cette liste possède la couleur c .
3. Coder l'algorithme glouton dans une fonction `coloration_glouton(L)` prenant en paramètre la liste de listes d'adjacence du graphe d'incompatibilité. Cette fonction parcourt les sommets dans l'ordre 0, 1, 2, ... et renvoie la liste des couleurs à attribuer dans le même ordre.

4. Quelle est dans le pire des cas la complexité asymptotique de cette fonction ?
5. Appliquer cet algorithme à quelques exemples.

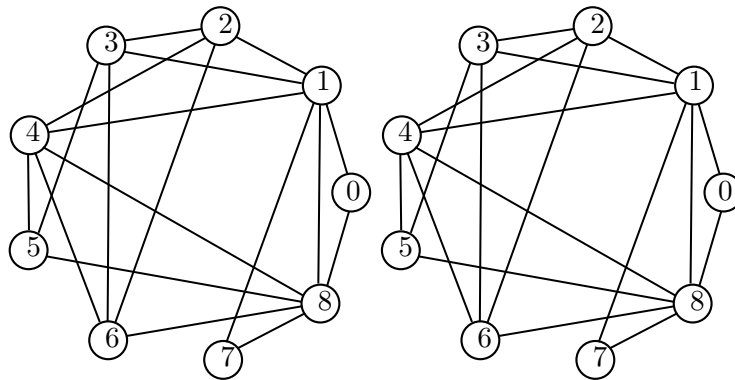


FIGURE 4 – Application manuelle des deux algorithmes de coloration sur un même graphe (désigné par `graphe_1` dans le fichier).

Algorithme de Welsh et Powell

L'algorithme naïf n'est pas optimal dans la mesure où il utilise souvent plus de couleurs que nécessaire. Un algorithme amélioré, mais pas non plus optimal, est celui de Welsh et Powell. Il repose sur l'idée que, plutôt que de traiter les sommets dans un ordre quelconque, il vaudrait mieux traiter en priorité ceux dont le degré est le plus élevé. Il procède donc selon les étapes suivantes.

- i. Déterminer le degré de chaque sommet.
- ii. Trier les sommets par degrés décroissants.
- iii. Parcourir la liste des sommets ainsi triés et attribuer à chacun d'eux la première couleur, s'il n'est pas adjacent à un sommet déjà colorié avec cette même couleur.
- iv. Tant qu'il reste des sommets non coloriés, prendre une nouvelle couleur et parcourir à nouveau la liste comme dans l'étape précédente pour les traiter.

Voici les étapes du travail à effectuer.

1. Appliquer l'algorithme à la main sur l'exemple de la figure (4).
2. Écrire une fonction `degre(i, L)` qui renvoie le degré du sommet `i` dans un graphe de listes d'adjacence `L`.
3. Écrire une fonction `second_element(liste)` qui renvoie le second élément d'une liste ou d'un tuple. Cette fonction semble triviale mais vous verrez son utilité plus loin. Utiliser le mot-clé `assert` pour garantir que la fonction s'exécutera uniquement si la liste a au moins deux éléments.
4. Coder enfin la fonction `welsh_powell(L)` qui renvoie la liste des couleurs à attribuer à chacun des nœuds du graphe.

On utilisera une liste `liste_sommets_degrees` de la forme `[(0, deg(0)), (1, deg(1)), (2, deg(2)), ...]` dont chaque élément est un tuple formé d'un nœud et de son degré. Nous n'allons pas coder l'étape de tri mais utiliser paresseusement la méthode `sort` sur les listes en précisant que le tri s'effectue par ordre décroissant selon le second élément de chaque tuple : `liste_sommets_degrees.sort(key = second_element, reverse = True)`.

5. Appliquer la fonction `welsh_powell` aux mêmes exemples que l'algorithme naïf et comparer les résultats.

III Graphes pondérés et arbre couvrant minimal

III.1 Pondération des arêtes et représentation informatique

Considérons les deux graphes de la figure 5, l'un orienté et l'autre non. À côté de chaque arête nous avons placé un nombre qui représente son « *poids* » (ou sa *valeur*) ; le graphe est dit *pondéré* (ou *valué*) et nous noterons ℓ_{ij} le poids de l'arête joignant deux sommets x_i et x_j . Cette valeur représente par exemple la distance entre deux villes sur une carte routière ou le temps de transport entre deux stations d'un réseau de communications. Nous nous limiterons dans la suite à des exemples pour lesquels toutes les valeurs sont positives ou nulles. Dans ce contexte,

on peut encore appeler ℓ_{ij} « longueur de l'arête $\{x_i, x_j\}$ ». Comme plus haut, nous pouvons adopter divers types de représentations informatiques d'un graphe pondéré.

- **représentation par une matrice d'adjacence**

Elle est analogue à celle définie dans la première partie mais, lorsqu'il existe une arête entre deux nœuds i et j , $M_{ij} = \ell_{ij}$. Lorsqu'il n'existe pas d'arête entre i et j , on pose $M_{ij} = \infty$.

- **représentation par des listes d'adjacence**

La liste $L[i]$ est maintenant une liste de tuples (x_j, ℓ_{ij}) énumérant les voisins x_j de x_i avec pour chacun la longueur ℓ_{ij} de l'arête qui le relie à x_i . Par exemple, le graphe de la partie gauche figure 5 est représenté par

```
1 L_adj_4 = [[(1, 3), (2, 10)], [(0, 3), (2, 6), (3, 2), (4, 7), (5, 4)], [(0,
    10), (1, 6), (5, 3)], [(1, 2), (4, 1)], [(1, 7), (3, 1), (5, 2)], [(1, 4),
    (2, 3), (4, 2)]]
```

- **Représentation par un dictionnaire de dictionnaires d'adjacence**

Les clés sont les nœuds du graphe, et les valeurs sont elles-mêmes des dictionnaires. Le sous-dictionnaire associé au nœud i a pour clés les nœuds j adjacents à i et pour valeurs les longueurs ℓ_{ij} . Par exemple, le graphe de la partie gauche figure 5 est représenté par

```
1 G_adj_4 = {0: {1: 3, 2: 10}, 1: {0: 3, 2: 6, 3: 2, 4: 7, 5: 4},
    2: {0: 10, 1: 6, 5: 3}, 3: {1: 2, 4: 1},
    4: {1: 7, 3: 1, 5: 2}, 5: {1: 4, 2: 3, 4: 2}}
```

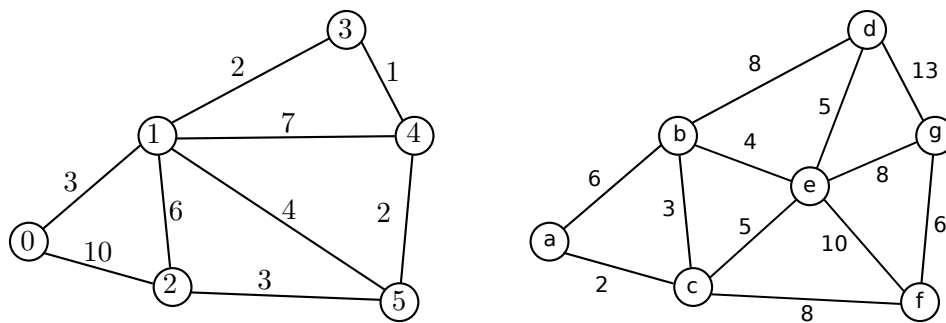


FIGURE 5 – Exemples de graphes valués. À gauche : graphe de liste L_adj_4 . À droite : graphe G_p224 .

À propos des graphes pondérés, vous avez étudié en PCSI l'algorithme de Dijkstra qui permet de trouver le plus court chemin d'un nœud donné à un autre nœud quelconque. Nous allons nous pencher cette année sur un autre problème : celui de l'arbre couvrant minimal.

III.2 Arbre couvrant minimal

Reprenons le graphe de la partie gauche de la figure 5 en imaginant qu'il s'agit d'un réseau routier et que la valeur de chaque arête correspond au prix de construction d'une route. Notre but est ici de minimiser le coût total du réseau en ne construisant que les routes strictement nécessaires relier deux villes quelconques en passant éventuellement par des villes intermédiaires. En langage mathématique, nous cherchons un sous-graphe connexe $G' = (S, A')$ de $G = (S, A)$, possédant les mêmes sommets et un sous-ensemble $A' \subset A$ de ses arêtes, tel que $\sum_{a \in A'} l_a$ soit minimale. On démontre que ce sous-graphe est forcément un arbre (c'est à dire un graphe connexe sans cycle). On dit donc que l'on recherche *un arbre couvrant minimal* (ACM en abrégé) de G . Nous admettons qu'il comporte $m' = n - 1$ arêtes (on rappelle que $n = |S|$ est le nombre de sommets).

Il serait très malhabile de rechercher G' par examen exhaustif de tous les sous-graphes de G car ils sont en général trop nombreux. Les paragraphes ci-dessous présentent deux algorithmes assez efficaces permettant d'obtenir l'ACM.

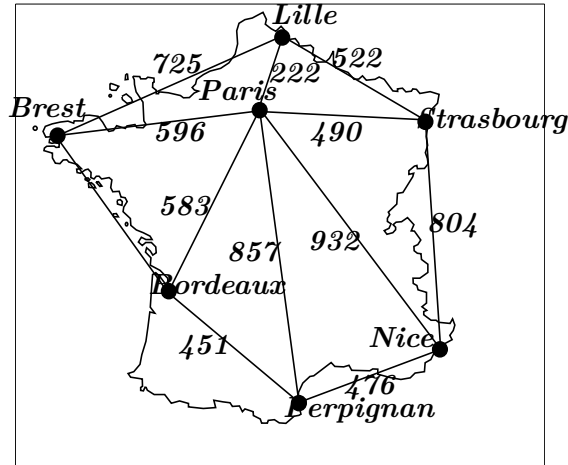
III.3 Algorithme de Kruskal

L'algorithme de Kruskal consiste à construire pas à pas G' selon les étapes suivantes.

- On établit une liste de toutes les arêtes du graphe avec leur poids.

- On trie cette liste d'arêtes par poids croissants.
- On part d'un graphe ne comportant aucune arête : $G' = (S, A')$ avec $A' = \emptyset$. On considère une par une les arêtes, de la plus courte vers la plus longue. Chacune est sélectionnée, c'est à dire inscrite dans A' , à condition qu'elle ne crée pas de cycle. L'ensemble A' grandit ainsi peu à peu et on continue jusqu'à ce qu'il contienne $n - 1$ arêtes. Nous pouvons exécuter à la main cet algorithme sur le graphe des villes de France.

L'algorithme de Kruskal est un glouton. Malgré sa gloutonnerie, on peut démontrer qu'il conduit à un optimum, c'est à dire qu'il fournit en effet un arbre couvrant *minimal*. La démonstration utilise un invariant de boucle difficile à manipuler et nous ne l'étudierons pas. Avant de coder l'algorithme, on l'exécute à la main sur l'exemple de villes de France.



Codage

On se limite à des graphes non orientés et on choisit la représentation d'un graphe par un dictionnaire.

1. Pour coder l'algorithme de Kruskal, on a besoin de la liste des arêtes du graphe. Écrire la fonction `liste_aretes(G)` qui prend en paramètre le dictionnaire d'adjacence d'un graphe et renvoie la liste de ses arêtes. Une arête est représentée un triplet de la forme (x_i, x_j, ℓ_{ij}) typé comme tuple. On prendra garde à n'écrire qu'une seule fois chaque arête.

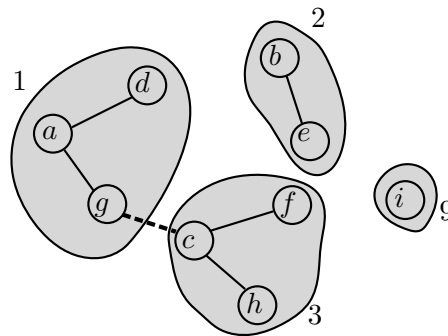


FIGURE 6 – gestion de la connexité dans l'algorithme de Kruskal

2. Avant d'insérer une arête dans A' , il faut s'assurer que cela ne créera pas de cycle dans le graphe. Les deux extrémités x_i et x_j de cette arête doivent donc se trouver dans des composantes connexes distinctes de G' en cours de construction, c'est à dire ne pas être déjà reliées par un chemin. Pour s'en assurer, on attribue à chaque nœud un entier que j'appelle « marqueur » et qui désigne le numéro de la composante connexe à laquelle il appartient. Au départ, lorsque A' est vide, tous les nœuds portent des marqueurs distincts (chacun d'eux est isolé sur un îlot). Dès que deux nœuds sont mis en relation par une arête, les composantes connexes auxquelles ils appartiennent fusionnent. On donne donc à tous les nœuds de ces deux composantes connexes le même marqueur, arbitrairement choisi égal à celui de la première extrémité x_i de l'arête qui les connecte. Sur la figure 6 par exemple, on a quatre composantes connexes portant les marqueurs 1, 2, 3 et 9. Si on place l'arête (c, g) (en pointillés), les trois sommets a , d et g se voient attribuer le marqueur 3. Les marqueurs des sommets sont stockés dans un dictionnaire : la valeur `marqueur[x]` est le marqueur du sommet x .

Écrire la fonction `introduit_un_cycle_ou_fusion(arete, marqueur)` qui prend une arête (x_i, x_j, ℓ_{ij}) et le dictionnaire des marqueurs. Elle analyse l'appartenance éventuelle des deux extrémités d'une arête à une même composante connexe et renvoie la réponse dans un booléen. Dans le cas négatif, on fusionne les deux composantes connexes en mettant à jour le dictionnaire `marqueur`. On rappelle qu'un dictionnaire est une variable mutable est qu'il peut être modifié lors de l'exécution d'une fonction dans laquelle il intervient comme paramètre.

3. Coder la fonction `indice_2(T)` qui renvoie l'élément d'indice 2 d'un tuple `T`. On s'assurera avec `assert` que la fonction ne s'exécute que si `T` comporte au moins 3 éléments.

4. Écrire la fonction `Kruskal(G)` qui prend comme paramètre le dictionnaire d'adjacence d'un graphe connexe valué et qui renvoie la liste des arêtes constituant un arbre couvrant minimal. Pour trier la liste d'arêtes, utiliser la méthode `sort` avec le mot clé `key` qui permet d'effectuer le tri selon les valeurs d'une fonction appliquée à chaque élément de la liste (ici, la longueur de chaque arête).

5. La fonction précédente renvoie un résultat qui suffit en principe. Par souci d'homogénéité avec le paragraphe suivant, coder la fonction `Kruskal_ACM(G)` qui renvoie le dictionnaire représentant, en tant que graphe, un arbre couvrant minimal de `G`.

6. Seulement si vous êtes en avance, écrivez la fonction `poits_total(G)` qui renvoie le poids total des arêtes d'un graphe G (représenté par un dictionnaire d'adjacence), puis utiliser cette fonction pour calculer le poids des ACM sur les exemples proposés.

III.4 Algorithme de Prim

L'algorithme de Prim poursuit le même objectif que celui de Kruskal : trouver un arbre couvrant minimal dans un graphe pondéré G . Son principe est le suivant. On choisit l'un quelconque des sommets du graphe comme premier sommet de l'ACM à construire (ce sera la « racine »). Parmi tous les autres sommets, on en choisit un connecté à la racine par une arête et situé à une distance minimale, et on l'ajoute à l'ACM en cours de construction (avec l'arête correspondante). L'arbre contient maintenant deux nœuds et une arête. Puis on choisit à nouveau, parmi les nœuds pas encore présents dans l'ACM et connectés à l'un des deux premiers, un sommet situé à distance minimale, et on l'incorpore à l'arbre (avec l'arête correspondante). L'arbre contient maintenant trois sommets et deux arêtes. On itère ce mécanisme jusqu'à ce que tous les sommets soient atteints. Exécutons manuellement cet algorithme sur le graphe `G_p224` de la figure (5) en choisissant arbitrairement le sommet a comme racine. Sur la feuille, repasser en couleur les arêtes choisies tour à tour puis calculer le poids de l'arbre finalement obtenu.

Comme l'algorithme de Kruskal, celui de Prim est glouton puisqu'il choisit irrévocablement ce qui est le plus avantageux à chaque étape. On démontre néanmoins, et nous l'admettrons, qu'il conduit en effet à un arbre couvrant *minimal*. Sa mise en œuvre rappelle l'algorithme de Dijkstra et elle utilise les données ci-dessous.

- un ensemble de booléens de `T`, tel que `T[x]` indique si le sommet x a déjà été sélectionné ou non pour faire partie de l'ACM ;
- un ensemble de distances : `dist[x]` représente la distance du nœud x (situé hors de l'ACM) à l'ACM en cours de construction. Si x n'est pas encore connecté à l'arbre, cette distance est égale à ∞ . Si x est connecté à l'arbre, c'est la plus courte des arêtes reliant x à un nœud de l'arbre.
- un ensemble de « prédécesseurs ». L'ajout d'un nœud y à l'arbre a lieu lorsqu'il est le plus proche de l'arbre, connecté par une arête à un nœud x déjà présent dans l'arbre. On dit que x est le prédécesseur de y et on pose `pred[y] = x`. On convient que la racine n'a pas de prédécesseur. En pratique, on affecte provisoirement `pred[y] = x` dès que la distance de y à x prend une valeur finie, signifiant par là que x est un prédécesseur potentiel de y . Si nécessaire, on mettra à jour `pred[y]` si on trouve un nœud x' qui en est plus proche.

Le détail de l'algorithme est le suivant.

1. Initialiser tous les booléens `T` à `False`, toutes les distances `dist` à ∞ et tous les prédécesseurs `pred` à `None`. Cela exprime le fait qu'au départ, aucun nœud n'a été sélectionné.
2. Choisir arbitrairement un sommet racine t (en pratique, nous laisserons l'humain le choisir à l'appel de la fonction).
3. Mettre à jour `T`, `pred` et `dist` pour traduire le fait que l'ACM contient t .
4. Tant qu'il reste des sommets non inclus dans l'ACM :
 - chercher un sommet m à distance minimale de ceux déjà présents dans l'ACM ;

- noter ce sommet comme atteint (il entre dans l'ACM) ;
- pour tous les sommets s non encore atteints et connectés à m : si $\ell_{s,m} < \text{dist}[s]$, remplacer $\text{dist}[s]$ par $\ell_{s,m}$ et remplacer $\text{pred}[s]$ par m .

Nous choisissons de représenter T , pred et dist par trois dictionnaires.

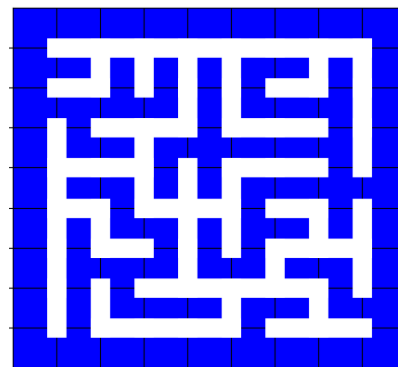
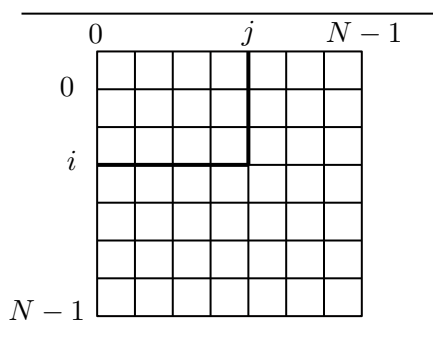
1. On se place à une étape quelconque du déroulement de l'algorithme. Coder la fonction `il_en_reste(G, T)` qui renvoie un booléen indiquant s'il reste, dans le graphe défini par le dictionnaire G , au moins un sommet pas encore sélectionné.
2. On se place à une étape quelconque du déroulement de l'algorithme. Coder la fonction `le_plus_proche(G, T, dist)` qui renvoie la clé désignant, dans le graphe de dico G , le sommet non encore sélectionné le plus proche de la partie de l'ACM déjà construite.
3. Coder la fonction `Prim_pred(G, racine)` qui prend en paramètre le dictionnaire représentant un graphe ainsi que le nœud choisi arbitrairement comme racine par l'utilisateur (lorsqu'il appellera la fonction). Elle applique l'algorithme de Prim et renvoie le dictionnaire des prédécesseurs.
4. Coder la fonction `Prim_ACM(G, racine)` qui, en utilisant la fonction précédente, calcule et renvoie le dictionnaire représentant un ACM de G . On rappelle qu'il s'agit d'un dictionnaire dont les valeurs sont elles-mêmes des sous-dictionnaires. En vue de l'application au labyrinthe, cette fonction renvoie aussi le dictionnaire des prédécesseurs.
5. Exécuter la fonction sur les graphes `G_villes`, `G_adj_4` et `G_p224`.
6. Seulement si vous êtes en avance, écrivez la fonction `poits_total(G)` qui renvoie le poids total des arêtes d'un graphe G , puis utiliser cette fonction pour calculer le poids des ACM sur les exemples proposés.

III.5 Application de l'ACM : construction d'un labyrinthe parfait

Un labyrinthe peut être considéré comme un graphe dans lequel chaque nœud est connecté ou non à chacun de ses voisins dans le plan, selon qu'il en est séparé par un couloir ou par un mur infranchissable. On souhaite ici construire aléatoirement des labyrinthes parfaits, c'est à dire dans lesquels il existe un unique chemin conduisant d'un nœud à un autre nœud quelconque. Un labyrinthe parfait est un arbre (un graphe connexe sans cycle) et on peut commodément en produire aléatoirement en utilisant le concept d'arbre couvrant minimal.

Pour cela, on considère tout d'abord un réseau à maille carrée comportant N lignes et N colonnes. Chaque nœud est relié à ses quatre voisins dans le plan, sauf sur les bords où il n'y a que trois voisins et dans les coins où il n'y en a que deux. Puis on attribue à chaque arête un poids aléatoire et enfin on extrait de ce graphe un arbre couvrant minimal : c'est le labyrinthe voulu ! Il est important de comprendre que, dans cet exemple, chaque sommet du graphe est repéré par deux entiers i et j , et non pas par un simple numéro comme précédemment. Dans le cas $N = 2$, le dictionnaire définissant le graphe se présente donc sous la forme illustrée par l'exemple ci-dessous.

```
1 G = { (0, 0): {(0, 1): 0.93, (1, 0): 0.55}, (0, 1): {(0, 0): 0.88, (1, 1): 0.76},
      (1, 0): {(0, 0): 0.27, (1, 1): 0.74}, (1, 1): {(0, 1): 0.11, (1, 0): 0.73} }
```



1. Coder la fonction `reseau_carre_alea(N)` qui renvoie, sous forme de dictionnaire, le graphe d'un réseau carré aléatoire dans lequel chaque arête porte un poids aléatoire dans $[0, 1[$. On utilise pour cela `rd.random()` du module `random` importé avec le préfixe `rd`.
2. Coder la fonction `construit_laby(N, racine)` qui construit, selon la méthode exposée plus haut, un labyrinthe de taille $N \times N$. La fonction renvoie le dictionnaire-graphe constituant le labyrinthe (les poids aléatoires, bien

qu'inutiles, sont conservés). Si vous utilisez Prim, renvoyer aussi le dictionnaire des prédécesseurs (qui sera utile pour la suite).

3. Pour le plaisir, produire quelques labyrinthes et les représenter avec la fonction `dessine_laby(G_laby, N)` fournie par le professeur.

La suite ne peut être pas être traitée si on a utilisé l'algorithme de Kruskal car on ne dispose alors pas des prédécesseurs.

4. Se déplacer d'un point A à un point B dans le labyrinthe est en général un casse-tête, mais pas pour nous ! En effet, tel l'architecte mythologique Dédale, nous connaissons la manière dont le labyrinthe a été construit et il nous est facile de déterminer le chemin menant de sa racine à un autre nœud quelconque. Il suffit pour cela d'utiliser les prédécesseurs : en remontant de proche en proche à partir du point à rejoindre, nous obtenons l'unique chemin venant de la racine. Coder la fonction `reconstruit_soluce(racine, arrivee, pred_laby)` qui renvoie la liste des nœuds à parcourir du point de départ `racine` jusqu'au point d'arrivée `arrivee`.
5. Choisir un point d'arrivée, utiliser la fonction précédente pour trouver le chemin qui y conduit depuis la racine.
6. Représenter le chemin de la racine jusqu'à l'arrivée grâce à la fonction `trace_soluce(soluce)` fournie par le professeur.

Résolution récursive du labyrinthe

7. Nous nous plaçons maintenant du point de vue d'un joueur *qui ne connaît pas le tableau des prédécesseurs*. Il cherche l'unique chemin menant d'un nœud `depart` à un nœud `arrivee`. Ce problème s'apparente à au parcourt d'un graphe et nous allons le mener de manière récursive, ce qui revient à faire un *parcours en profondeur d'abord*. L'algorithme est le suivant.

- a) Si le point de départ et le point d'arrivée sont identiques, l'affaire est réglée !
- b) Sinon, on regarde à quels points on peut accéder en se déplaçant d'une seule case depuis le point de départ.
- c) S'il n'existe aucun point voisin accessible, on est dans une impasse et il faut revenir à la dernière bifurcation et explorer une autre voie.
- d) S'il existe des voisins accessibles, on les essaie un par un et pour chacun d'eux,
 - on condamne le passage qui y mène (et le passage en sens inverse), afin de ne pas l'explorer inutilement une seconde fois ;
 - on tente d'atteindre l'arrivée en considérant ce voisin comme un nouveau point de départ ;
 - si ce voisin conduit à la solution, le chemin cherché est formé du point de départ et du chemin commençant sur ce voisin.
- e) si aucun des points voisins ne permet d'atteindre l'arrivée, le point de départ est une impasse.

La phrase soulignée suggère naturellement un codage récursif. Dans une vision imagée, on peut imaginer qu'à partir d'un point de départ, on envoie un explorateur vers chacune des directions possibles et que chacun d'eux reviendra en s'écriant « chef, j'ai trouvé la sortie ! » ou au contraire « chef, vous m'avez envoyé dans une impasse ! ». Ayant avancé d'une case depuis qu'on lui a confié sa mission d'exploration, chaque explorateur est pris de paresse. Il s'arrête et demande à d'autres explorateurs d'avancer plus profondément dans le labyrinthe pour faire le travail à sa place.

La fonction récursive à coder renvoie un booléen indiquant si la recherche a été couronnée de succès (ou si elle a conduit à une impasse), et la liste des nœuds formant le chemin jusqu'à l'arrivée. Pour condamner un passage, on attribue la longueur fictive -1 à l'arrête correspondante. Compléter le code ci-dessous.

```

1 def recherche_chemin_rec(depart, arrivee, G_lab) :
    if depart == arrivee :
        return ..., ...
5
    noeuds_accessibles = [s for s in G_lab[depart] if G_lab[depart][s] >= 0]

    if noeuds_accessibles == [] : # c'est un cul-de-sac
        return ..., ...
10
    else : # on explore depuis chacun des points voisins
        for voisin in noeuds_accessibles :
```

15

```
G_lab[...][...] = -1 # on déclare ce passage fermé
G_lab[...][...] = -1 # en sens inverse aussi
# Appel récursif
essai, suite_chemin = recherche_chemin_rec(..., ..., ....)
if essai : # on a trouvé !
    return ... , [depart] + ....
return ..., .... # si TOUS les voisins ont conduit à une impasse
```

