Algorithmes sur les graphes

1 Définitions et fonctions élémentaires

1.1 Définitions et vocabulaire

Un *graphe* G est un couple (X, E) constitué d'un ensemble X non vide et d'un ensemble E de paires d'éléments de X. Les éléments de X sont appelés *nœuds* (ou sommets) et les éléments de E, de la forme $e = \{x, y\}$ avec $x \in X$ et $y \in X$, sont appelés *arêtes*. Dans toute la suite, on note $n = |X|$ le nombre de nœuds et $m = |E|$ 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 : $e = \{x, y\}$ est synonyme de $e = \{y, x\}$. On dit dans ce cas que le graphe n'est pas orienté. Mais il existe aussi des *graphes orientés* dans lesquels E n'est pas formé de paires mais de *couples* (x, y) d'éléments de X. On dit alors que (x, y) est un *arc* et on le représente avec une flèche pour bien rappeler qu'il est distinct de l'arc (y, x).

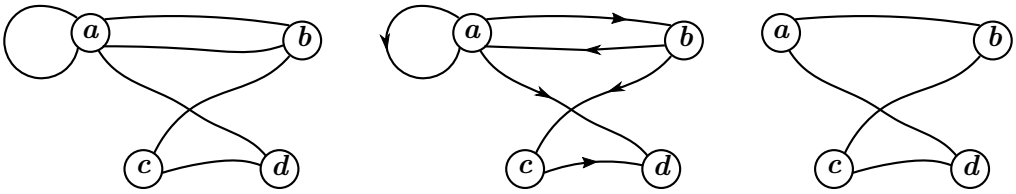


FIGURE 1 – Un graphe non orienté (à gauche), un graphe orienté (au milieu) et un graphe non orienté simple (à droite). Écrire à la main l'ensemble X et l'ensemble E pour chacun d'eux.

Le vocabulaire sur les graphes orientés diffère légèrement de celui applicables aux graphes non orientés mais nous ne chercherons pas la rigueur sémantique. Dans la suite de ce paragraphe, les dénominations particulières aux graphes orientés sont simplement indiquées entre parenthèses après leur analogue pour les graphes non orientés.

- *boucle* : arrête de type $\{a, a\}$
- *arrête double ou multiple* : répétition de l'arête $\{a, b\}$ dans E.
- *graphe simple* : graphe sans boucle ni arête multiple

Nous travaillerons surtout avec des graphes simples non orientés.

- *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 (ou degré intérieur) et on le note $d^-(x)$;
  - le nombre d'arêtes qui sortent de x, on le nomme degré sortant (ou degré extérieur) et on le note $d^+(x)$.
- *chaîne* (ou *chemin*) : suite de la forme $x_0, e_0, x_1, e_1, ..., x_{k-1}, e_{k-1}, x_k$ avec $x_i \in X$ et $e_i \in E$. Le nombre d'arêtes k est appelé *longueur de la chaîne*. Dans le cas d'un graphe simple, on peut omettre les e_i dans cette énumération puisqu'il existe au plus une arête d'un sommet à un autre.
- 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.

1.2 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 X et E, sous forme de deux listes par exemple, n'est jamais utilisée pour des raisons d'efficacité. On lui préfère d'autres représentations parmi lesquelles nous retenons deux approches.

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} est égal au nombre d'arêtes allant du nœud x_i au nœud x_j. Pour un graphe simple, la matrice ne contient que des zéros et des uns. Elle est symétrique pour un graphe non orienté. 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. On construit donc une liste `L` de n listes, la liste `L[i]` contenant les indices des nœuds voisins du nœud x_i. La complexité spatiale d'une telle représentation est en $O(m)$.

Écrire les matrices d'adjacence et les listes d'adjacence associées aux graphes dessinés plus haut

1.3 Codage de fonctions élémentaires

— 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 comportant éventuellement des arêtes multiples.

— Écrire une fonction `matrice_vers_listes(L)` qui fait le travail inverse. Pour simplifier, on suppose ici que le graphe est simple. Quelle est la complexité de ces deux fonctions ?

— Écrire une fonction `degre(i, M)` calculant le degré du sommet i dans un graphe défini par sa matrice d'adjacence.

— Écrire une fonction `est_chaine(suite, M)` qui prend en argument une suite d'entiers représentants 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.

— Selon le temps disponible, ou si vous êtes en avance, vous pouvez écrire une fonction `construit_graphe_simple_alea(n, degre_max)` qui construit aléatoirement et renvoie les listes d'adjacence d'un graphe aléatoire simple, non orienté, comportant n nœuds de degré compris 0 et `degre_max`. Vous pourrez utiliser la fonction `randint(a, b)` du module `random` qui renvoie un entier aléatoire compris entre a et b inclus. Attention : le graphe ne doit comporter ni boucle ni arrête multiple. Cette fonction vous permettra de produire des graphes pour essayer les fonctions précédentes ou celles de la partie suivante.

Vous pouvez faire tourner ces fonctions sur le graphe de la figure 2 ou sur des graphes plus volumineux générés aléatoirement et dont le professeur vous fournira les listes d'adjacence.

2 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, bien qu'il existe divers encadrement de cet entier, appelé *nombre chromatique* du graphe et noté γ. Il est par exemple 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

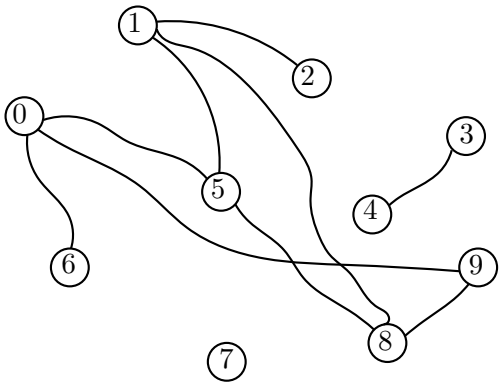


FIGURE 2 – Un graphe sur lequel exécuter les fonctions

mutuellement, attribution de fréquences à des antennes relais en évitant les interférences, résolution de Sudoku, 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 stocker ces produits 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.

	0	1	2	3	4	5	6	7
0		✖	✖	✖			✖	✖
1	✖				✖	✖	✖	
2	✖			✖		✖	✖	✖
3	✖		✖		✖			✖
4		✖		✖		✖	✖	
5		✖	✖		✖			
6	✖	✖	✖		✖			
7	✖		✖	✖				

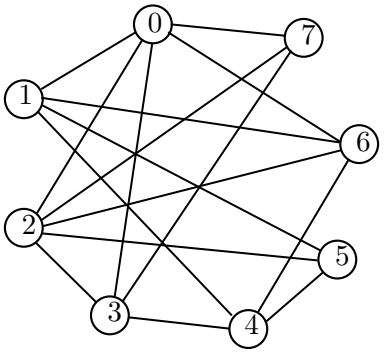


FIGURE 3 – Tableau d'incompatibilité de produits chimiques et graphe d'incompatibilité associé

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.

Algorithmes de coloration

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 compatible qui n'entre pas en conflit avec ses voisins déjà coloriés. Il s'agit d'un *algorithme glouton* : une fois qu'on a attribué une couleur à l'un des sommets, on ne revient plus sur ce choix même si on s'aperçoit plus tard que

ce n'est pas optimal.

1. Appliquer à la main l'algorithme sur le graphe de la figure 4.
2. Coder d'abord la fonction auxiliaire `est_deja_prise(c, liste_de_sommets, Couleur)` qui renvoie un booléen indiquant si un des sommets de la liste `liste_de_sommets` possède la couleur `c`. La couleur de chacun des sommets du graphe est indiquée dans la liste `Couleur` : `Couleur[i]` est la couleur du sommet i.
3. Coder l'algorithme glouton naïf 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 les couleurs à attribuer dans le même ordre
4. Appliquer cet algorithme à quelques exemples.

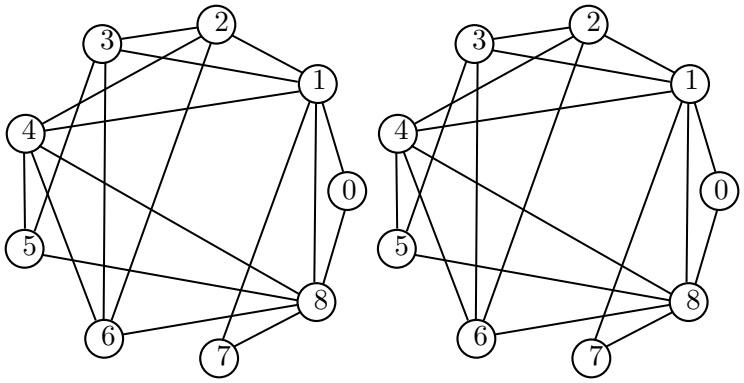


FIGURE 4 – Application manuelle des deux algorithmes de coloration

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 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. Écrire une fonction `degre(i, L)` qui renvoie le degré du sommet `i` dans un graphe de listes d'adjacence `L`.
2. Écrire une fonction `second_element(liste)` qui renvoie le second élément d'une liste ou d'un tuple. Vous verrez son utilité plus loin.
3. 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_degres` 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_degres.sort(key = second_element, reverse = True)`.
4. Appliquer la fonction `welsh_powell` aux mêmes exemples que l'algorithme naïf et comparer les résultats.

3 Parcours en profondeur et composantes connexes

À partir d'un nœud donné, on peut parcourir un graphe de proche en proche en suivant ses arêtes. Cette exploration peut avoir pour but de déterminer si un graphe est *connexe* ou de déterminer la *composante connexe* à laquelle le nœud choisi appartient. Ces termes sont définis ci-dessous.

— Une graphe est dit *connexe* s'il existe, pour tout sommet x et tout sommet y, une chaîne reliant x et y.

— Les *composantes connexes* d'un graphe G sont les sous-graphes engendrés connexes maximaux de G. Plutôt que de définir précisément le terme « sous-graphe engendré » et l'adjectif « maximal », illustrons cette définition par un dessin (figure 5).

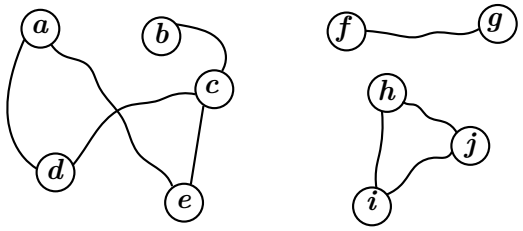


FIGURE 5 – Exemple de graphe non connexe présentant trois composantes connexes. Entourez-chacune d'elles au crayon comme à l'école maternelle.

La difficulté d'un tel parcours exploratoire réside dans la nécessité de ne visiter qu'une seule fois chaque nœud tout en étant certain de n'en oublier aucun. Pour éviter les doubles visites, il suffit de tenir à jour un tableau de booléens indiquant, pour chaque nœud, s'il a déjà été visité ou non. Quant à l'exploration elle-même, on peut l'effectuer selon l'algorithme du *parcours en profondeur d'abord* dont le principe est le suivant : partant d'un nœud, on visite un par un tous ses voisins. Chacune de ces visites inclut la visite des voisins du voisin visité. Cette idée appelle naturellement une écriture récursive dont le cas d'arrêt correspond à l'absence de voisin.

À titre d'exemple de parcours en profondeur d'un graphe, on peut citer le langage de description de molécules SMILES qui permet de représenter une molécule par une chaîne de caractères obtenue en parcourant en profondeur d'abord sa formule développée, considérée comme un graphe, et en appliquant certains règles conventionnelles.

1. Coder la fonction `parcours_profondeur_rec_0(L, i, marques)` qui réalise le parcours en profondeur. Les paramètres sont la liste de listes d'adjacence `L`, le nœud de départ i et un tableau de booléens `marques` qui sera initialisé avant l'appel à la fonction et indiquant si chaque nœud a été visité. Cette fonction ne renvoie rien mais modifie le tableau `marques` qui, à l'issue de l'exécution, contiendra l'information voulue.

   Pour suivre visuellement l'exécution, vous pouvez placer à l'endroit approprié une instruction affichant le message « début de l'exploration à partir du sommet ... » ou bien « fin de l'exploration pour le sommet ... ». Ces deux options correspondent à deux ordres d'affichage des sommets parcourus que l'on appelle respectivement *ordre préfixe* et *ordre postfixe*.

   Procéder à quelques essais sur les graphes fournis en exemples. Réfléchir au choix du vocabulaire : « parcours en *profondeur d'abord* ».

2. Quelle est la complexité de ce parcours en profondeur dans le pire des cas ? On compte comme unité de coût les affectations et les tests.

3. Coder la fonction `est_connnexe(L)` qui renvoie un booléen indiquant si le graphe de listes d'adjacence `L` est connexe. La tester.

4. Coder la fonction `parcours_profondeur_rec_1(L, i, marques, L_atteignables)` qui, en plus d'attribuer les marques, construit la liste `L_atteignables` des nœuds que l'on peut atteindre depuis le nœud `i` (lui-même compris). Cette fonction, fortement inspirée de `parcours_profondeur_rec_0`, ne renvoie rien non plus mais modifie la liste de booléens `marques` et celle `L_atteignables` qui énumère les sommets atteignables depuis i. Ces deux listes seront initialisées avant

l'appel de la fonction et contiendront la réponse après son exécution. Je vous impose ici d'opter pour un énumération *postfixe* des sommets, c'est à dire de placer chacune d'eux dans la liste `L_atteignables` après la fin de l'exploration de ses descendants.

5. La structure choisie ci-dessus n'est pas très pratique : on doit créer avant l'appel de la fonction des listes `marques` et `L_atteignables` que la fonction modifiera. Pour plus de commodité, nous allons encapsuler la fonction `parcours_profondeur_rec_1` et l'initialisation de ces deux listes dans une autre fonction `parcours_profondeur(L, i)`. Coder cette fonction pour qu'elle *renvoie* la composante connexe à laquelle `i` appartient. L'appliquer à quelques exemples.
6. Écrire la fonction `composantes_connexes(L)` qui renvoie les composantes connexes d'un graphe de listes d'adjacences `L` sous forme d'une liste de listes, chacune d'elle rassemblant les nœuds d'une même composante connexe. Il suffit pour cela de parcourir successivement tous les nœuds : chaque fois qu'on en découvre un qui n'a pas été déjà visité, on « ouvre » une nouvelle composante connexe que l'on construit en réalisant le parcours en profondeur depuis le nœud en question.

Remarque
Nous savons que l'exécution d'une fonction récursive s'effectue au moyen d'une pile de récursivité. De fait, il est possible de « dérécursifier » l'algorithme de parcours en profondeur en manipulant explicitement une pile des sommets à visiter. Chacun des voisins à visiter est empilé, puis dépilé lors de sa visite qui comprend l'empilement de ses propres voisins. Ce n'est pas très difficile à coder, mais je doute que nous en prenions le temps, d'autant que l'écriture récursive s'avère plus confortable

4 Graphes valués et algorithme de Dijkstra

Considérons les deux graphes de la figure 6, l'un orienté et l'autre non. À côté de chaque arête nous avons placé un nombre qui représente sa *valeur* ; le graphe est dit *valué* et nous noterons ℓ_{ij} la valeur de l'arc 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 de tels exemples pour lesquels toutes les valeurs sont positives ou nulles. Dans ce contexte, nous appellerons ℓ_{ij} « longueur de l'arête $\{x_i, x_j\}$ » [1]. Comme plus haut, nous pouvons adopter deux types de représentations informatiques.

— **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.

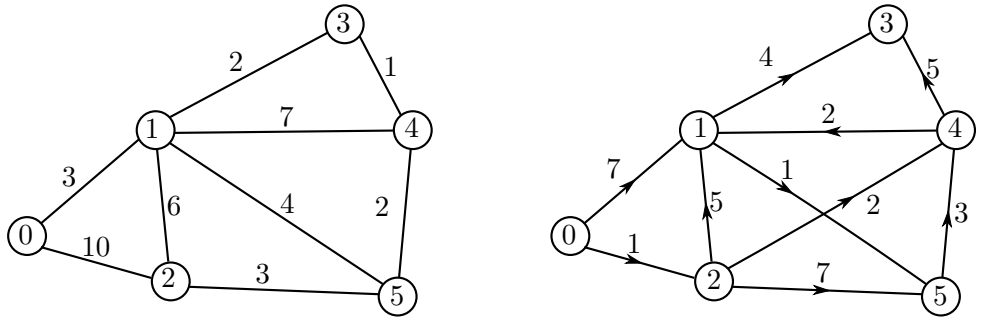


FIGURE 6 – Exemples de graphes valués. Écrire leur matrice d'adjacence et leurs listes d'adjacence.

Considérons maintenant deux sommets x_i et x_j reliés par une chaîne. On peut attribuer à cette chaîne une longueur égale à la somme des longueurs des arêtes qui la constituent. Il existe en général plusieurs chemins menant de x_i à x_j et, dans de nombreuses applications, on souhaite trouver celui qui est le plus

1. Ce mot prend ici un sens différent de celui qu'il avait dans la première partie

court. C'est ce problème que résolvent par exemple les logiciels de navigation routière ainsi que le protocole OSPF de transport d'information sur internet. Cette résolution se fait selon l'algorithme de Dijkstra, célèbre informaticien et mathématicien néerlandais. En voici le principe.

Nous supposons le graphe connexe et cherchons le plus court chemin d'un nœud de départ x_0 vers chacun des nœuds du graphe. On attribue à chaque sommet x une *étiquette* $\lambda(x)$ qui va évoluer au fil des itérations de l'algorithme pour progressivement atteindre la longueur minimale cherchée entre x_0 et x. Simultanément, l'ensemble $\mathcal{S}$ des nœuds x pour lesquels $\lambda(x)$ a atteint sa valeur minimale (et pour lesquels le chemin optimal est donc connu) grossit peu à peu.

Au départ, le seul nœud vers lequel on connaît trivialement le chemin minimal est x_0 : comme il est à distance nulle de lui-même, on pose $\lambda(x_0) = 0$ et cette étiquette n'aura pas à être modifiée. Pour les autres sommets on pose $\lambda(x) = \infty$, signifiant par là que le chemin optimal qui y mène est au départ inconnu. Après cette initialisation, l'algorithme réalise des itérations qui se poursuivent tant que $\mathcal{S}$ n'est pas égal à X ou, de manière équivalente, tant que $\overline{\mathcal{S}}$ n'est pas vide. Chaque itération met en jeu un « sommet actif » t_0, au départ égal à x_0, et se déroule en deux temps.

1. **révision des étiquettes**
   Pour tout sommet x de $\overline{\mathcal{S}}$ voisin de t_0, calculer $\lambda(t_0) + \ell_{t_0x}$. Si cette somme est plus petite que $\lambda(x)$, modifier $\lambda(x)$ en lui attribuant cette nouvelle valeur.
2. **choix d'un sommet d'étiquette minimale**
   Choisir dans $\overline{\mathcal{S}}$ un sommet t_0 d'étiquette minimale et le placer dans l'ensemble $\mathcal{S}$. Il joue le rôle de sommet actif pour la prochaine itération.

La situation est représentée sur la figure 7. Le professeur fournira peut-être, selon le temps disponible, une preuve de terminaison et de correction de cet algorithme. La correction met en jeu l'invariant de boucle suivant.

« Pour tous les sommets x de $\mathcal{S}$, $\lambda(x)$ est la longueur du chemin minimal de x_0 à x. »

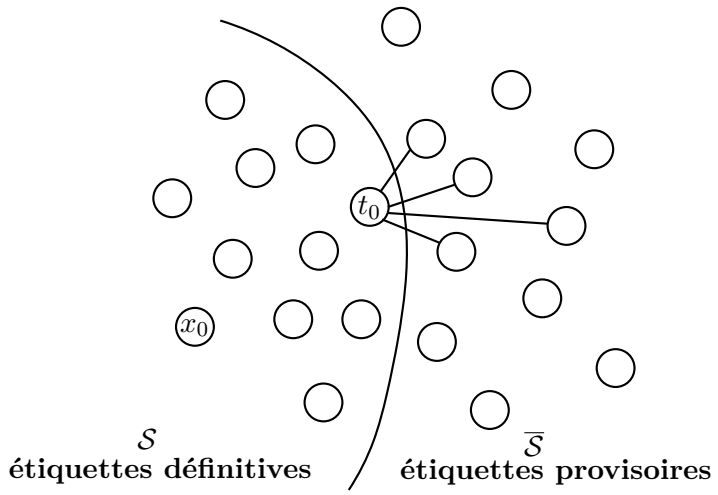


FIGURE 7 – Illustration de l'algorithme de Dijsktra

En pratique, on souhaite trouver non seulement la longueur minimale de x_0 à x mais aussi le chemin optimal qu'il faut suivre du nœud de départ au nœud d'arrivée. Dans ce but, on attribue un *père* à chaque sommet x : c'est le sommet qui précède x dans ce chemin optimal de x_0 à x, c'est à dire le sommet actif t_0 par lequel on a calculé l'étiquette définitive de x sous la forme $\lambda(t_0) + \ell_{t_0x}$. *Lors de l'étape de révision des étiquettes, on révise donc également les pères.*

Pour nous familiariser avec l'algorithme, appliquons-le à la main sur les deux exemples de la figure 6 en partant du sommet $A = 0$ en remplissant les tableaux placés plus bas. Les sommets sont désignés par une lettre plutôt que par un numéro afin de ne pas les confondre avec les longueurs des chemins. À chaque itération de l'algorithme, lors de la révision des étiquettes, on écrit dans la colonne portant le sommet x en en-tête, les valeurs de $\lambda(x)$. On note aussi dans cette colonne à partir de quel sommet t_0 le sommet x a été atteint, c'est à dire son père provisoire.

Lorsqu'on sélectionne t_0, on souligne l'étiquette correspondante sur cette ligne. Elle devient alors définitive, t_0 entre dans $\mathcal{S}$ et, comme on ne modifiera plus $\lambda(t_0)$, on trace un trait vertical dans la partie restante

de sa colonne. De même le père provisoire de t_0 reçoit le statut de père définitif et on l'indique dans la colonne la plus à droite. Le point de départ A n'a pas de père et nous le qualifions d'orphelin.

itér.	A	B	C	D	E	F	t_0	$\lambda(t_0)$	éléments de S	père de t_0 définitif
0	$\underline{0}$	∞	∞	∞	∞	∞	A	0	A	A orphelin
1	|	$\underline{3}$ A	10 A	∞	∞	∞	B	3	AB	p(B) = A
2	|	|								
3	|	|								
4	|	|								
5	|	|								
6	|	|								

itér.	A	B	C	D	E	F	t_0	$\lambda(t_0)$	éléments de S	père de t_0 définitif
0	$\underline{0}$	∞	∞	∞	∞	∞	A	0	A	A orphelin
1	|									
2	|									
3	|									
4	|									
5	|									
6	|									

Codage de l'algorithme de Dijsktra

Il faut garder à l'esprit que les sommets sont codés par des entiers et que les graphes sont donnés par des listes d'adjacence. Les voici en exemple pour les deux graphes de la figure 6.

```
L_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)]]
L_3 = [[(1,7),(2,1)],[(3,4),(5,1)],[(1,5),(4,2),(5,7)],[],[(1,2),(3,5)],[(4,3)]]
```

On utilise aussi :

— une liste `etiquette` de longueur n dont l'élément d'indice i est $\lambda(x_i)$;

— une liste `pere` de longueur n dont l'élément `pere[i]` est le père du nœud x_i ;

— une liste `S_barre` de longueur variable contenant les éléments de $\overline{\mathcal{S}}$.

1. Coder la fonction `revision(t_0, L, etiquette, S_barre, pere)` qui révise les étiquettes à partir d'un sommet actif t_0 et met à jour la liste des pères. On pourra utiliser l'expression `a in Liste` qui renvoie un booléen indiquant si l'élément `a` figure dans la liste `Liste`. Cette fonction ne renvoie rien mais modifie les listes `etiquette` et `pere`.
2. Écrire la fonction `choix_sommet_etiquette_minimale(etiquette, S_barre)` qui renvoie le sommet de $\overline{\mathcal{S}}$ dont l'étiquette est minimale.
3. Écrie la fonction `Dijkstra(L, x_0)` qui met en place l'algorithme. Elle renvoie la liste des étiquettes définitives et celle des pères définitifs. Par souci de simplicité, on utilise la méthode `remove` applicable aux listes : `Liste.remove(x)` supprime l'élément `x` de la liste `L` contenant `x`.
4. Écrire la fonction `construit_chemin(pere, depart, arrivee)` qui utilise le tableau des pères relatifs à un nœud de départ pour reconstruire le chemin optimal vers un nœud d'arrivée. Ce chemin est renvoyé comme la liste ordonnée des nœuds à parcourir. Un écriture avec une boucle `while` est possible, mais le choix d'un codage récursif est ici tout à fait naturel.
5. Appliquer l'algorithme aux graphes de la figure 6 ou à d'autres plus compliqués que le professeur vous fournira.

5 Arbre couvrant minimal et algorithme de Kruskal

Reprenons le graphe de la partie gauche de la figure 6 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 de manière à pouvoir relier deux villes quelconques en passant éventuellement par des villes intermédiaires. En langage mathématique, nous cherchons un sous-graphe connexe $G' = (X, A)$ de $G = (X, E)$, possédant les mêmes nœuds et un sous-ensemble A de ses arêtes, tel que $\sum_{e \in A} l_e$ soit minimale. On démontre que ce sous-graphe est forcément un arbre, c'est à dire qu'il ne comporte pas de cycle [2]. On dit donc que l'on recherche *l'arbre couvrant minimal* de G. Nous admettons qu'il comporte $m' = n - 1$ arêtes.

La recherche de G' ne peut pas s'effectuer par examen exhaustif de tous les sous-graphes de G car ils sont en général trop nombreux. Selon le théorème de Cayley, un graphe complet [3] d'ordre n en possède n^{n-2} ce qui donne environ $2,6.10^{23}$ pour $n = 20$.

L'algorithme de Kruskal consiste à construire pas à pas G'. On part d'un graphe ne comportant aucune arête : $G' = (X, A)$ avec $A = \varnothing$. De manière itérative, on sélectionne dans E l'arête la plus courte possible, on la raye de E, puis on la place dans A *si cela est possible*. Ce processus se poursuit jusqu'à ce que A possède les $n-1$ arêtes requises pour couvrir tous les nœuds. En ajoutant petit à petit de nouvelles arêtes, il faut veiller à ne pas former de cycle, ce qu'exprime l'expression « si cela est possible ». Appliquons à la main l'algorithme sur l'exemple du graphe associé à quelques villes de France. Lorsqu'on sélectionne une arête et qu'on la place dans A, on la surligne. Si on la rejette pour cause de cyclicité, on la biffe de hachures.

Codage

1. Pour coder l'algorithme de Kruskal, on a besoin de la liste des arêtes du graphe. Écrire la fonction `liste_adjacence_vers_liste_aretes(L)` qui prend en paramètres les listes d'adjacence d'un graphe non orienté 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.
2. Écrire la fonction `trouve_arete_mini(F)` qui prend en argument une liste d'arêtes et en renvoie une de longueur minimale, sous forme de tuple comme décrit dans la question précédente.

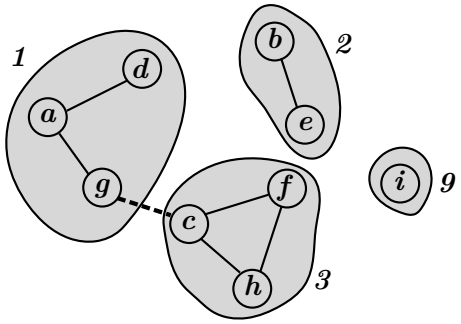


FIGURE 8 – gestion de la connexité dans l'algorithme de Kruskal

3. 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', 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. Dès que deux nœuds sont mis en relation par une arête, les composantes connexes auxquelles ils appartiennent fusionnent. On donne donc à tous ces nœuds le même marqueur, arbitrairement choisi égal à celui de la première extrémité x_i de l'arête. Sur la figure 8 par exemple, on a quatre composantes connexes portant les numéros 1, 2, 3 et 9. Si on place l'arête en pointillés entre (c, g), les trois sommets

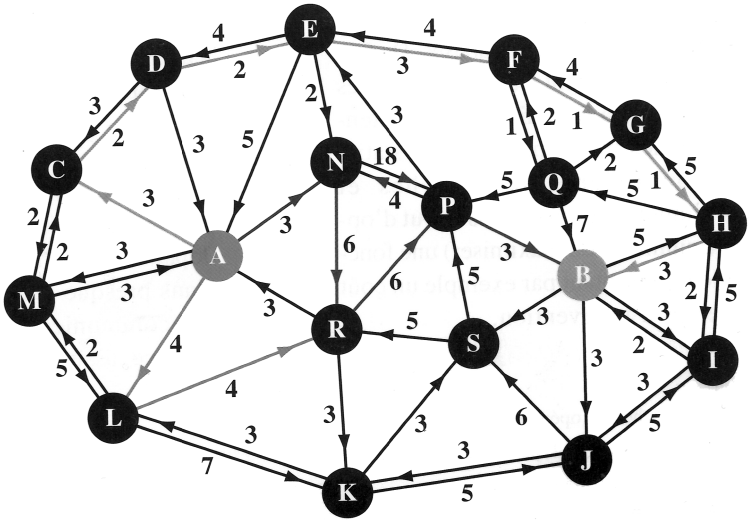
2. On l'aurait deviné puisqu'un cycle correspond à l'existence de deux chemins distincts entre deux nœuds. On peut faire des économies en en supprimant un

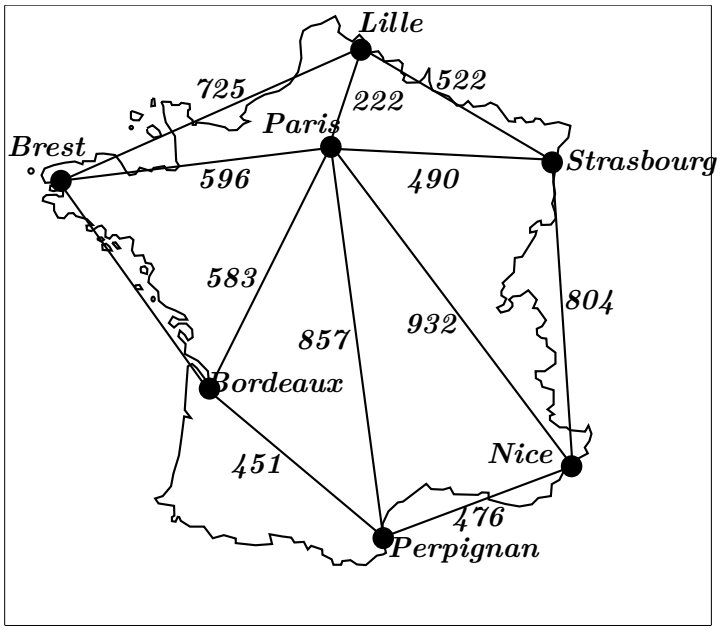
3. un graphe est dit complet si chaque nœud est voisin de tout autre nœuds.

a, d et g se voient attribuer le marqueur 3. Les marqueurs des sommets sont stockés dans une liste de longueur n : `marqueur[i]` est le marqueur du sommet x_i.

Écrire la fonction `introduit_un_cycle_ou_fusion(arete, marqueur)` qui 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 la liste `marqueur` passée par référence.

4. Écrire enfin la fonction `Kruskal(L_adjacence)` qui prend comme seul argument les listes d'adjacence d'un graphe connexe valué et qui renvoie la liste des arêtes constituant l'arbre couvrant minimal. Par souci de simplicité, on utilisera la méthode `remove` sur les listes.

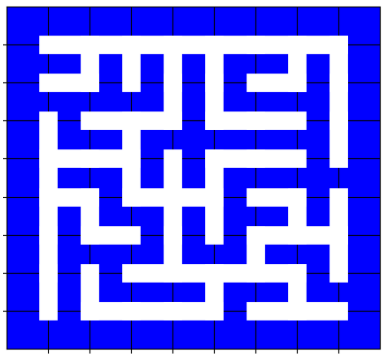


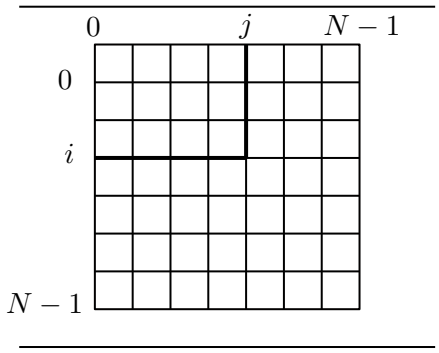


Application : construction d'un labyrinthe parfait

Un labyrinthe peut être considéré comme un graphe dans lequel chaque nœud est connecté ou non à chacun de ses plus proches voisins, selon qu'il en séparé par un passage ou par un mur. 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. Un labyrinthe parfait est donc un graphe du type arbre connexe et on peut l'obtenir commodément en utilisant le concept d'arbre couvrant minimal.

Pour cela, on considère d'abord un réseau dans lequel chaque nœud est relié à ses plus proches voisins. Pour simplifier, nous nous limitons à un réseau à maille carrée comportant N lignes et N colonnes. Puis on attribue à chaque arête un poids aléatoire et enfin on extrait de ce graphe un arbre courant minimal : le tour est joué !





1. Dans le réseau carré, un nœud est repéré par ses indices (i, j) alors que dans les graphes manipulés jusqu'ici, un nœud est associé à un entier. Pour utiliser les fonctions précédentes, nous devons assurer le passage d'un type de repérage à l'autre. On numérote les nœuds du réseau carré de 0 à $N^2 - 1$ selon l'ordre de lecture (on parcourt la première ligne de gauche à droite, puis la seconde, etc). Écrire la fonction `numero(i, j, N)` qui renvoie le numéro du nœud d'indices (i, j).
2. Écrire la fonction `indices(k, N)` qui fait le travail inverse, c'est à dire qui renvoie les indices (i, j) du nœud numéro k.
3. Écrire la fonction `ajoute_arete_alea(L_adj, p, q)` qui, prenant en argument la liste de listes d'adjacence d'un graphe, ajoute un arc allant du nœud d'indice p vers celui d'indice q.
4. Écrire la fonction `reseau_carre_alea(N)` qui renvoie la liste de listes d'adjacence d'un graphe pondéré aléatoire dont les nœuds sont les points d'un réseau carré de taille $N \times N$. On utilisera la fonction `random()` du module `random` pour obtenir des flottants compris entre 0 et 1.
5. Écrire enfin la fonction `produit_labyrinthe_alea(N)` qui produit le labyrinthe. Il doit être renvoyé sous la forme d'une liste d'arêtes, une arête étant ici un tuple de deux couples d'indice (i, j). Par exemple, $((3, 4), (3, 5))$ signifie qu'on peut circuler entre les points $(3, 4)$ et $(3, 5)$.
6. Je vous fournirai une fonction pour représenter graphiquement le labyrinthe.

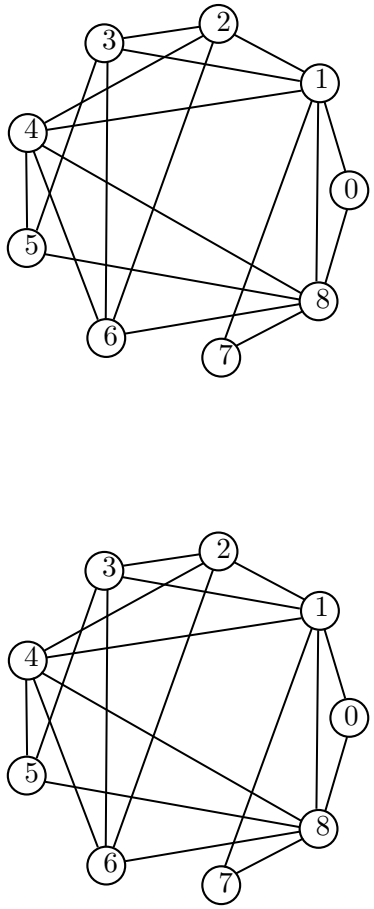


FIGURE 9 – Application manuelle des deux algorithmes de coloration

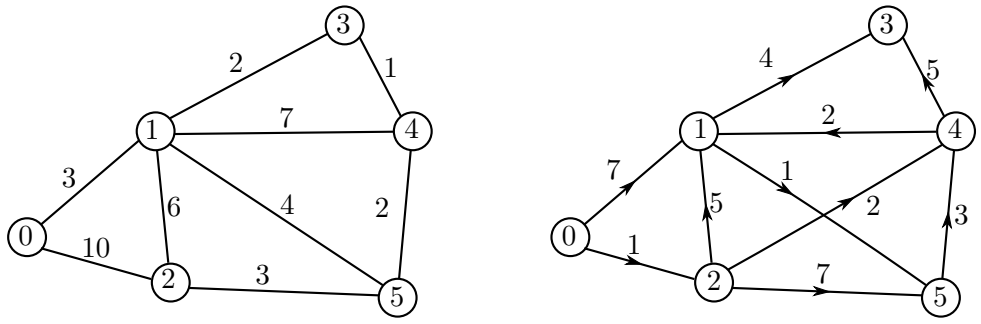
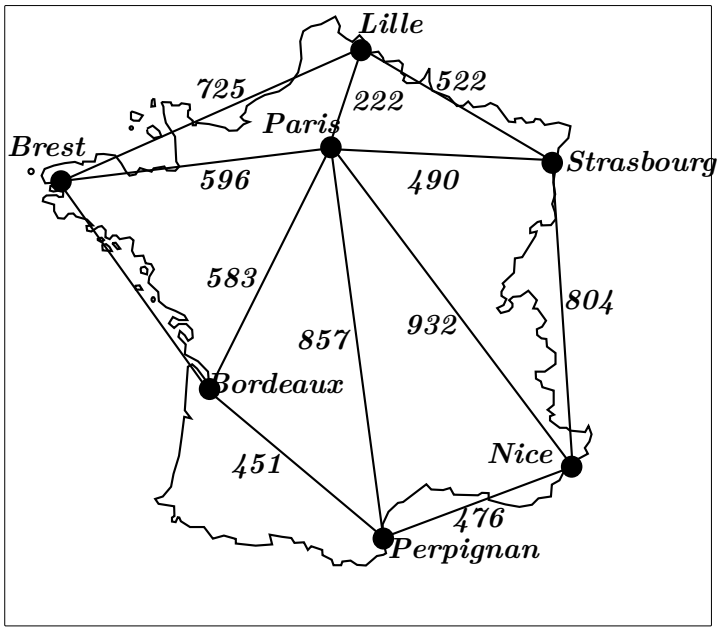


FIGURE 10 – Exemples de graphes valués. Écrire leur matrice d'adjacence et leurs listes d'adjacence.

itér.	A	B	C	D	E	F	t_0	$\lambda(t_0)$	éléments de S	père de t_0 définitif
0	0	∞	∞	∞	∞	∞	A	0	A	A orphelin
1	|	3 A	10 A	∞	∞	∞	B	3	AB	p(B) = A
2	|	|								
3	|	|								
4	|	|								
5	|	|								
6	|	|								

itér.	A	B	C	D	E	F	t_0	$\lambda(t_0)$	éléments de S	père de t_0 définitif
0	0	∞	∞	∞	∞	∞	A	0	A	A orphelin
1	|									
2	|									
3	|									
4	|									
5	|									
6	|									


Lille
725
522
222
Paris
Brest
Strasbourg
596
490
583
804
857
932
Bordeaux
451
Nice
476
Perpignan