TP INFO 11 | Graphes

Vous trouverez ici de nombreux exercices classiques sur la théorie des graphes.

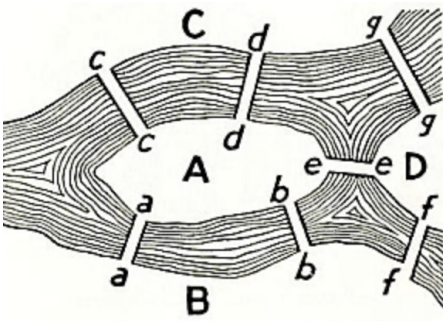


FIGURE 98. *Geographic Map: The Königsberg Bridges.*

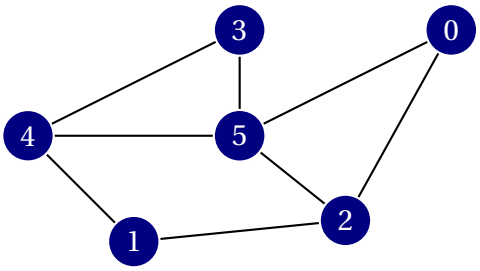
1. Exercices

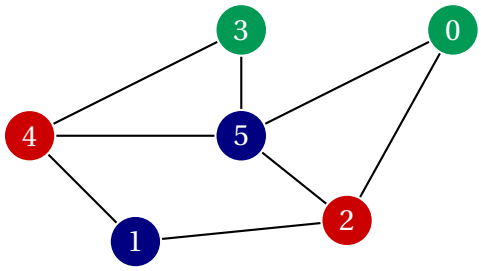
1 — *De la liste d'adjacence à la liste des arrêtes*

Écrire une fonction `listeDesAretes` prenant en argument la liste d'adjacence `L` d'un graphe non orienté et simple G = (V,E) avec E = $[\![0, n-1]\!]$ renvoyant la liste des arêtes de G. On codera une arrête par la liste de ses extrémités `[i,j]` avec la convention $i < j$. Quelle est la complexité de votre algorithme ?

2 — *Coloration d'un graphe non orienté* f

Dans cet exercice, on considère un graphe G := (S,A) simple et non orienté avec S := $[\![0, n-1]\!]$ où $n \in \mathbb{N}^*$. Colorer le graphe G signifie attribuer une couleur à chacun de ses sommets de façon à ce des sommets voisins soient de couleurs distinctes.





Une proposition de coloration sera codée au moyen d'une liste `colors` telle que `colors[i]` donne le numéro de la couleur attribuée au sommet i pour tout i dans $[\![0, n-1]\!]$.

1. On considère un graphe G, pour lequel on a une proposition de coloration codée par une liste `colors`. On souhaite déterminer si cette coloration est valide.
   - **a.** Écrire une fonction `coloration_valide` prenant en arguments la liste d'adjacence `t` de G et une liste de couleurs `colors`, qui renvoie `True` si la coloration est valide, `False` sinon.
   - **b.** Quelle est la complexité dans le pire des cas de la fonction `coloration_valide` ?
2. Nous allons à présent étudier un algorithme de coloration du graphe G. L'attribution des couleurs à chaque sommet est caractérisée par une liste `colors` où `colors[i]` est la couleur attribuée au sommet i pour tout i dans $[\![0, n-1]\!]$. Initialement, la liste `colors` ne contient que des -1 et ses valeurs sont modifiées progressivement au fur et à mesure que les couleurs sont attribuées.
   - **a.** Écrire une fonction `colore_sommet` prenant en arguments la liste `colors` des couleurs attribuées, le numéro `s` du sommet à colorer et la liste d'adjacence `t` caractérisant le graphe. Cette fonction ne renvoie rien mais modifie la liste `colors` en donnant à `colors[s]` la plus petite couleur possible, en fonction des couleurs des sommets voisins qui sont déjà colorés.

     Considérons le graphe donné en exemple ci-dessus. La coloration des sommets 0 et 5 aboutit à `colors=[0,-1,-1,-1,-1,1]`. Les sommets 0 et 5 ont donc été colorés avec les couleurs `colors[0]=0` et `colors[5]=1`. L'appel `colore_sommet(colors,4,t)` modifie la liste `colors` en `colors=[0,-1,-1,-1,0,1]`. Cela veut dire que le sommet 4, adjacent au sommet 5 mais pas au sommet 0, a reçu la même couleur que le sommet 0.

b. En déduire une fonction `colorer` avec pour argument une liste `t` caractérisant un graphe, qui crée et renvoie la liste `colors` des numéros des couleurs attribuées en colorant les sommets un par un par ordre croissant de leurs numéros.

Par exemple, l'application de la fonction `colorer` au graphe G de l'exemple ci-dessus renverra la liste de couleurs `[0,0,1,0,1,2]`.

3. Justifier au moyen d'un exemple simple que l'algorithme utilisé dans `colorer` n'est pas toujours optimal, i.e. ne renvoie pas toujours une coloration du graphe avec le moins de couleurs possible.

3 — *Puits d'un graphe orienté simple* ƒ

Soit $n \in \mathbb{N}^*$ et $G := \big([\![0, n-1]\!], A\big)$ un graphe simple et orienté.

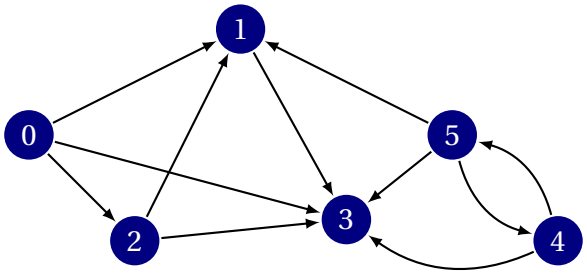
⇛ Soit $(i, j) \in [\![0, n-1]\!]$. On dit que :

- i est un prédécesseur de j si $(i, j) \in A$;
- i est un successeur de j si $(j, i) \in A$.

⇛ Un sommet i_0 de G est un puits s'il n'admet aucun successeur et si tous les autres sommets sont des prédécesseurs de i_0.

Les graphes seront représentés par leur matrice d'adjacence au format d'une liste de listes.

1. Le graphe G ci-dessous admet-il un puits ?



2. Combien y a-t-il de puits au maximum dans G ?

3. Écrire une fonction `compte_pred` qui prend en entrée un graphe G et un de ses sommets i, et qui renvoie le nombre de prédécesseurs de i. Quelle est la complexité de cette fonction ?

4. Écrire une fonction `trouve_puits` qui prend en entrée un graphe G et qui renvoie le puits de G s'il existe et `None` sinon. Quelle est la complexité de cet algorithme en fonction de n ?

5. Écrire une fonction `trouve_puits_opt` qui prend en argument un graphe G et renvoie le puits de G s'il existe et `None` sinon, avec une complexité linéaire en n. On pourra considérer l'algorithme suivant : $i = 0$, pour j variant de 1 à $n-1$, poser $i = j$ si la personne i est un prédécesseur de j.

6. Le choix de coder un graphe par sa matrice d'adjacence est-il judicieux pour la recherche d'un éventuel puits ?

4 — *Plus court chemin dans un graphe simple, non orienté et non pondéré* ƒƒ

Soit $G := (S, A)$ un graphe simple, non orienté avec $S := [\![0, n-1]\!]$ et $n \in \mathbb{N}^*$.

1. On suppose G connexe. Soit $i \in [\![0, n-1]\!]$. Comment parcourir le graphe pour obtenir $\big(d(i,j)\big)_{0 \leqslant j \leqslant n-1}$?

   La notation $d(i,j)$ désignant la distance de i à j avec la convention $d(i,i) = 0$.

2. On ne suppose plus que G est connexe. Modifier la fonction du 1. pour la convention $d(i,j) = -1$ si i et j ne sont pas connectés.

5 — *Calcul des composantes connexes* ***ff***

On considère un graphe G := (S, A) non orienté où S := $[\![0, n-1]\!]$. Nous allons étudier deux stratégies pour savoir si G est connexe ou non.

1. Dans cette question, le graphe G sera représenté par une liste d'adjacence.

   a. Écrire une fonction `com_connexe` prenant en argument un graphe G, un sommet s et renvoyant sous la forme d'une liste la composante connexe du sommet s dans G.

   b. Écrire une fonction `est_connexe_1` prenant en argument un graphe G renvoyant `True` s'il est connexe, et `False` sinon.

2. Dans cette question, le graphe G sera représenté par sa matrice d'adjacence M au format d'une liste de listes.

   a. Montrer que, $\forall m \in \mathbb{N}$, $\forall (i,j) \in [\![0, n-1]\!]^2$, $\big(\mathrm{M}^m\big)_{i,j}$ est égal au nombre de chemin(s) de longueur m menant de i à j.

   b. En déduire une caractérisation de la connexité de G au moyen de la matrice $\mathrm{I}_n + \mathrm{M} + \cdots + \mathrm{M}^{n-1}$.

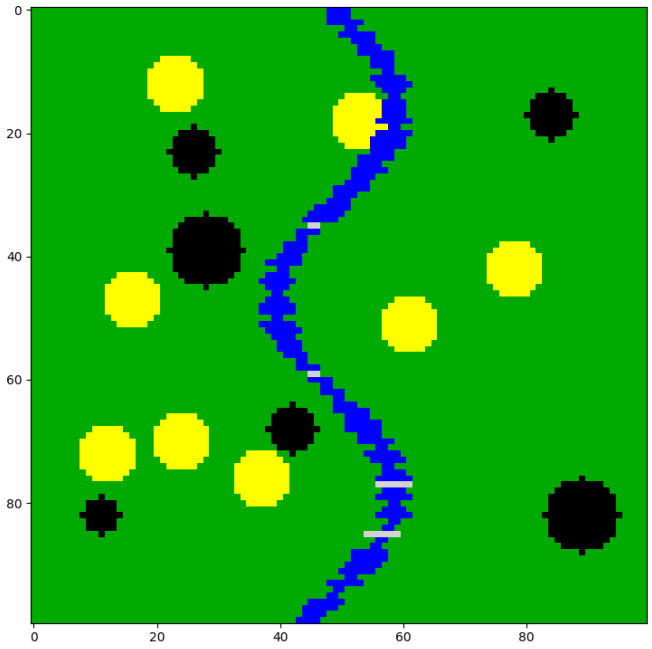
   c. En déduire une fonction `est_connexe_2` prenant en argument un graphe G renvoyant `True` s'il est connexe, et `False` sinon.

   INDICATION : On utilisera la bibliothèque `numpy`. On transforme une liste de listes A en une matrice de type `ndarray` par l'instruction `np.array(A)`. Les matrices 0 et I_n s'obtiennent par `np.zeros([n,n])` et `np.eye(n)`, le produit AB par `np.dot(A,B)` et la somme A + B par `A+B`.

6 — *Algorithme de Dijkstra* ***fff***

Nous allons appliquer l'algorithme de Dijkstra pour calculer un plus court chemin entre deux points d'un milieu hétérogène. Le terrain comporte un fleuve (en bleu), de l'herbe (en vert), des bancs de sable (en jaune), quatre ponts (en gris) qui enjambent le fleuve et des obstacles (en noir). Les obstacles sont infranchissables et le fleuve ne peut-être traversé qu'aux ponts. Ce milieu est codé par une matrice `t` de taille 100 × 100 et de type `ndarray` dont les coefficients valent

0 : herbe , 1 : sable , 2 : fleuve , 3 : pont , 4 : obstacle



On peut se déplacer d'un pixel à un autre dans les 8 directions (horizontalement, verticalement et en diagonale) tant que l'on reste dans la matrice et que le milieu le permet. Le coût d'un déplacement dépend de la nature de la case d'arrivée et de la direction, il s'écrit $d \times c$ où

$$d = \begin{cases} \sqrt{2} & \text{si le déplacement est diagonal} \\ 1 & \text{sinon} \end{cases} \quad \text{et} \quad c = \begin{cases} 2 & \text{si déplacement vers le sable} \\ 1 & \text{si déplacement vers herbe ou pont} \end{cases}$$

La recherche d'un chemin de coût minimal d'un point de départ A à un point d'arrivée B peut être résolue par l'algorithme de Dijkstra pour le graphe G suivant :

- Les sommets du graphe sont les cases de la matrices, repérées par le tuple (i, j) (indices de ligne et de colonne) ;
- Chaque case est reliée à ses voisines immédiates (8 au maximum) appartenant à la matrice et qui ne représentent ni de l'eau, ni un obstacle ;
- Les coûts décrits ci-dessus sont les valuations du graphe.

On téléchargera les fichiers `terrain.txt`, `dijkstra.py` et on écrira ses scripts dans ce dernier.

1. Écrire une fonction `voisines` prenant en argument le terrain `t`, les coordonnées i, j d'une case de cette matrice et qui renvoie la liste des voisines (ℓ, k) de (i, j) dans le graphe G.

2. Écrire une fonction `cout` prenant en argument le terrain `t`, deux sommets `s`, `v` du graphe G et qui renvoie le coût du déplacement de `s` vers `v`.

3. Écrire une fonction `cleMin` prenant en argument un dictionnaires à valeurs numériques et qui renvoie une clé correspondant à la plus petite valeur de `d`.

4. Écrire une fonction `dijkstra` prenant en argument le terrain `t`, les points de départ et d'arrivée A, B et qui renvoie la distance entre A et B ainsi qu'un plus court chemin de A à B dans G sous la forme d'une liste de cases.

5. Tester la fonction `dijkstra` pour plusieurs couples (A, B).

7 — *Optimisation avec l'algorithme A⋆* ***fff***

L'algorithme A* est une variante de celui de Dijkstra. Il ne donne pas nécessairement la solution optimale mais il a l'avantage d'arriver très vite à un résultat approché.

L'inconvénient de l'algorithme de Dijkstra pour un chemin de A à B est qu'il explore le graphe G en partant de A *dans toutes les directions*, dont la plupart sont pourtant peu pertinentes. Afin d'aider l'algorithme à aller dans *les bonnes directions*, on utilise une estimation de la distance au point d'arrivée.

Pour cela, on choisit une fonction h qui estime la distance *à vol d'oiseau* entre deux sommets. Lors du parcours du graphe de l'algorithme de Dijkstra en partant d'un sommet A, on choisit le sommet s qui minimise $d(\mathrm{A}, s) + h(s, \mathrm{B})$ (au lieu de $d(\mathrm{A}, s)$).

On revient à l'exemple du chemin optimal sur un terrain hétérogène de l'exercice précédent.

1. Quelle fonction h vous semble-t-elle adaptée dans ce cadre ? En déduire une fonction `H` prenant en argument deux sommets s et t et renvoyant la valeur de $h(s, t)$.

2. Écrire une fonction `a_star` prenant en argument le terrain `t` et les points de départ et d'arrivée A, B et qui renvoie la distance entre A et B ainsi qu'un chemin de A à B dans G (obtenu par l'algorithme A⋆) sous la forme d'une liste de cases.

3. Comparer les deux algorithmes : les solutions obtenues et le nombre de sommets visités par l'algorithme. On sera amener à modifier légèrement les fonctions `dijkstra` et `a_star`.

8 — *Algorithme de Floyd-Warshall* ***fff***

Soit $\mathrm{G} := (\mathrm{V}, \mathrm{E}, p)$ un graphe simple orienté et pondéré où $\mathrm{V} := [\![0, n-1]\!]$ avec $n \in \mathbb{N}^*$. Nous supposons que G peut contenir des arcs de poids négatifs mais aucun circuit de longueur strictement négative. Pour tout $(i, j) \in \mathrm{V}^2$ et $k \in [\![0, n]\!]$, $d_{i,j}^{(k)}$ est la longueur d'un plus court chemin de i à j n'utilisant comme sommets *intermédiaires* que des éléments $[\![0, k-1]\!]$.

1. Établir que, pour tout $(i, j) \in \mathrm{V}^2$, la distance entre i et j vaut $d^{(n)}(i, j)$.

2. Justifier que, pour tout $(i, j) \in \mathrm{V}^2$ et $k \in [\![0, n]\!]$, $d_{i,j}^{(k)} = \begin{cases} p_{i,j} & \text{si } k = 0 \\ \min\left(d_{i,j}^{(k-1)}, d_{i,k-1}^{(k-1)} + d_{k-1,j}^{(k-1)}\right) & \text{sinon} \end{cases}$

3. En déduire une fonction `distances` prenant en argument la matrice de pondération du graphe G et renvoyant la matrice $\left(d_{i,j}^{(n)}\right)_{(i,j)\in\mathrm{V}^2}$.

4. Quelle est la complexité de cet algorithme ?

5. Pour $(i,j) \in V^2$, on pose $\pi_{i,j}^{(0)} = i$ et, pour tout $k \in \mathbb{N}^*$:

$$\pi_{i,j}^{(k)} = \begin{cases} \pi_{i,j}^{(k-1)} & \text{si } d_{i,j}^{(k)} \leqslant d_{i,k-1}^{(k-1)} + d_{k-1,j}^{(k-1)} \\ \pi_{k,j}^{(k-1)} & \text{sinon} \end{cases}$$

a. Justifier que, pour tout $k \in [\![0,n]\!]$, pour tout $(i,j) \in V^2$, il existe un plus court chemin de i à j n'utilisant comme sommets intermédiaires que des éléments $[\![0,k-1]\!]$, dans lequel $\pi_{i,j}^{(k)}$ est un prédécesseur de j.

b. Écrire une fonction `plusCourtsChemins` prenant en argument la matrice de pondération du graphe G et renvoyant la matrice $A := \left(\pi_{i,j}^{(n)}\right)_{(i,j)\in V^2}$.

c. Comment calculer un plus court chemin entre i et j (sommets de G) à partir de la matrice A ?

9 — *Algorithme de Bellman-Ford* ***fff***

Soit $G := (V, E, p)$ un graphe simple orienté, connexe et pondéré où $V := [\![0, n-1]\!]$ avec $n \in \mathbb{N}^*$. On fixe $s \in V$ et on note m le cardinal de E. Nous supposons que G peut contenir des arcs de poids négatifs mais aucun circuit de longueur strictement négative.

1. Pour $k \in \mathbb{N}^*$ et $i \in V$, on note $d^{(k)}(i)$ la longueur d'un plus court chemin entre s et i contenant au maximum k arcs. Démontrer que

$$d^{(k)}(i) = \begin{cases} p_{s,i} \text{ si } k = 1 \\ \min\left(d^{(k-1)}(i), \min\limits_{j\in V}\left(d^{(k-1)}(j) + p_{j,i}\right)\right) \text{ sinon} \end{cases}$$

2. En déduire une fonction `distances` prenant en argument la matrice de pondération de G et un sommet s de G et renvoyant la liste des distances à s.

3. Proposer une méthode pour calculer un plus court chemin de s à $i \in V$ en cas d'existence.

2. Problème

10 — *Stable d'un graphe simple et non orienté* ***ff - fff***

Soit $n \in \mathbb{N}^*$ et $G := \left([\![0, n-1]\!], A\right)$ un graphe simple (c'est-à-dire sans boucle) et non orienté.

⇛ On notera classiquement m le cardinal de A , i.e. le nombre d'arêtes du graphe G .

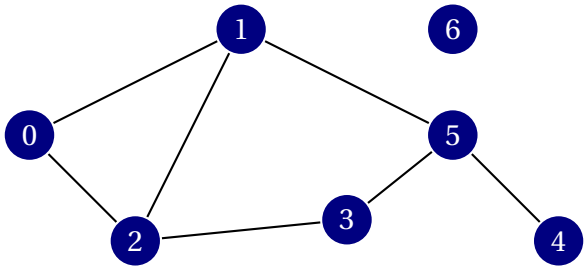
⇛ On appelle stable de G toute partie E de $[\![0, n-1]\!]$ telle que $\forall (i,j) \in E^2, \quad \{i,j\} \notin A$.

Moins formellement, un stable est donc un sous-ensemble de sommets deux à deux non voisins.

⇛ Un stable E de G est dit maximal si, pour tout stable E′ de G, $E \subset E' \implies E = E'$.

⇛ Un stable E de G est dit maximum s'il est de cardinal maximal parmi tous les stables de G.

⇛ On rappelle que le pluriel de maximal est maximaux, et que celui de maximum est maximums.



Par exemple, voici la liste de tous les stables du graphe dessiné ci-dessus : iI y a $\varnothing$, les singletons, les paires de sommets non voisins, $\{1,3,4\}$, $\{1,3,6\}$, $\{1,6,4\}$, $\{2,4,6\}$, $\{2,5,6\}$, $\{3,4,6\}$, $\{0,3,4\}$, $\{0,5,6\}$, $\{0,4,6\}$, $\{0,3,6\}$, $\{1,3,4,6\}$ et $\{0,3,4,6\}$. Les stables $\{0\}$ et $\{0,3,4,6\}$ sont respectivement non maximal et maximal.

Les graphes seront codés par listes d'adjacence dans tout le sujet sauf dans la partie IV où l'on opte plutôt pour un codage par dictionnaire d'adjacence. Une liste d'adjacence du graphe ci-dessus est par exemple :

```
L=[[1,2],[0,2,5],[0,1,3],[2,5],[5],[1,3,4],[]]
```

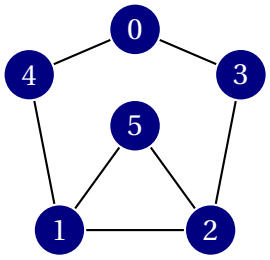
et un dictionnaire d'adjacence de ce même graphe est par exemple :

```
D={ 0:[1,2], 1:[0,2,5], 2:[0,1,3], 3:[2,5], 4:[5],5:[1,3,4], 6:[] }
```

Partie I – Préliminaires

Dans cette partie, on aborde quelques considérations théoriques sur les stables avant de créer trois fonctions qui seront utilisées dans la suite du sujet.

1. **a.** Déterminer tous les stables maximaux et tous les stables maximums du graphe suivant :



b. Un stable maximal est-il nécessairement maximum ?

c. Un stable maximum est-il nécessairement maximal ? On justifiera avec soin sa réponse.

2. On cherche à réaliser le plus possible de tâches parmi $\mathcal{T}_0,\ldots,\mathcal{T}_{n-1}$ sachant que certaines d'entre elles sont incompatibles. Justifier que ce problème se ramène au calcul d'un stable maximum d'un graphe G bien choisi que l'on décrira avec soin.

3. Écrire une fonction bin prenant en argument un entier naturel n, un élément k de $[\![0,2^n-1]\!]$ et renvoyant la liste de taille n de ses chiffres en base deux. Estimer sa complexité.

Par exemple, pour $n = 4$ et $k = 4$, la fonction doit renvoyer `[0,0,1,0]` car $4 = \underline{100}$. On notera que l'ordre souhaité dans la liste est inverse à celui de l'écriture traditionnelle en base deux et qu'il faut le cas échéant la compléter par des zéros afin qu'elle soit bien de longueur n.

4. Une partie E de $[\![0, n-1]\!]$ peut être codée par la liste binaire t de longueur n définie par

$$\text{t[i]} = \begin{cases} 1 & \text{si } i \in \text{E} \\ 0 & \text{sinon} \end{cases}$$

a. Écrire une fonction `codeEnsemble` prenant en argument n et une liste E d'élément de $[\![0, n-1]\!]$ et renvoyant son code binaire t. Estimer sa complexité.

Par exemple, l'appel `codeEnsemble(5,[1,4])` doit renvoyer `[0,1,0,0,1]`.

b. Écrire une fonction `decodeEnsemble` prenant en argument une liste binaire t de longueur n et renvoyant sous forme de liste la partie E de $[\![0, n-1]\!]$ qu'elle représente. Estimer sa complexité.

Par exemple, l'appel `decodeEnsemble([1,0,0,1,0])` doit renvoyer `[0,3]`.

Partie II – Algorithmes pour étudier la stabilité et la stabilité maximale

Dans cette partie, G est représenté par une liste d'adjacence L.

1. Écrire une fonction `estStable` prenant en argument la liste d'adjacence L de G et une liste de sommets E renvoyant `True` si l'ensemble des éléments de E est un stable de G, et `False` sinon.

Par exemple, pour le graphe donné en exemple dans le préambule, l'appel `estStable(L,[1,3,4])` doit renvoyer `True`. On impose une complexité en $O(n+m)$ que l'on justifiera avec soin.

2. Écrire une fonction `estStableMaximal` prenant en argument une liste d'adjacence L de G et une liste de sommets E et renvoyant `True` si l'ensemble des éléments de E est un stable maximal, et `False` sinon.

Pour le graphe donné en exemple dans le préambule, l'appel `estStableMaximal(L,[1,4])` doit renvoyer `False`. On impose une complexité en $O\big((n-k)(n+m)\big)$ où k est le cardinal de l'ensemble représenté par E.

3. Pour construire un stable maximal de G , on peut suivre l'algorithme suivant :

- ⇛ On marque à `False` tous les sommets et on initialise E au vide.
- ⇛ Tant qu'il existe un sommet x non marqué, on remplace E par $\text{E} \cup \{x\}$, on marque x puis on marque tous les voisins de x.
- ⇛ Renvoyer E.

a. Donner sans le justifier un invariant de boucle permettant de démontrer la correction de cet algorithme.

b. Écrire une fonction `constructionStableMaximal` prenant en argument un liste d'adjacence L de G et renvoyant sous la forme d'une liste de sommets un stable maximal de G.

Partie III – Énumération par force brute des stables maximaux et maximums

Dans cette partie, G est représenté par une liste d'adjacence L.

1. Pour déterminer tous les stables maximaux de G , l'algorithme par force brute consiste à énumérer toutes les parties E de $[\![0, n-1]\!]$ en tester à chaque fois s'il s'agit ou non d'un stable du graphe.

a. Écrire une fonction `stablesMaximaux` prenant en argument une liste d'adjacence L du graphe G et renvoyant la liste des stables maximaux de G.

Pour le graphe du préambule, l'appel `stablesMaximaux(L)` doit renvoyer la liste

```
[[1,3,4,6],[0,3,4,6]]
```

b. Estimer la complexité de votre fonction.

2. Dans cette question, on propose un algorithme de calcul des stables maximums de G par force brute.

a. Écrire une fonction `stableMaximum` prenant en argument une liste d'adjacence `L` du graphe G et renvoyant la liste des stables maximums de G.

b. Estimer la complexité de votre fonction.

Partie IV – Calcul d'un stable maximum par Branching

On propose dans cette section, un algorithme récursif de calcul d'un stable maximum de G. Le graphe sera dans cette partie codé par un dictionnaire d'adjacence D (cf. le préambule du sujet pour un exemple). Le principe de l'algorithme de Branching pour calculer un stable maximum est le suivant :

⇛ On choisit un sommet s.

⇛ On applique récursivement l'algorithme au graphe G_s obtenu à partir de G en supprimant le sommet s. On note E_s le stable maximum ainsi obtenu.

⇛ On applique récursivement l'algorithme au graphe G_{vs} obtenu à partir de G en supprimant s ainsi que les sommets voisins de s dans G. On note E_{vs} le stable maximum ainsi obtenu.

⇛ L'ensemble de plus grand cardinal entre E_s et $\{s\} \cup E_{vs}$ est un stable maximum de G.

1. Justifier brièvement cet algorithme récursif.

2. Déterminer les cas de base de cet algorithme (i.e. les cas d'initialisation).

3. En déduire une fonction `stableByBranching` prenant en argument un dictionnaire d'adjacence D du graphe G et renvoyant un stable maximum de celui-ci sous la forme d'une liste.

REMARQUE : Pour supprimer un sommet s du graphe, on pourra se contenter d'effectuer `del D[s]` mais on prendra garde dans la suite de l'algorithme à ce que le sommet s peut encore figurer dans certaines listes de voisins.

4. Expliquer comment modifier la fonction précédente afin d'obtenir tous les stables maximums de G.

3. Indications

1

Il suffit d'itérer sur tous les sommets puis sur les voisins de chacun d'entre-eux.

2

Pour le calcul de complexité, on se souviendra que $\sum_{i=0}^{n-1} d(i) = 2m$.

3

Au 5., on pourra démontrer qu'à la fin de l'exécution de l'algorithme donné, la variable i est nécessairement le puits du graphe si celui-ci en contient un.

4

Le parcours en largeur est très adapté car il s'effectue selon des distances croissances.

5

On remarquera que l'existence d'un chemin de i à j équivaut à l'existence d'un chemin de longueur inférieure ou égale à $n-1$ de i à j.

6

Attention, par rapport au cours, le graphe est parcouru sans structure se stockage prédéfinie mais au moyen de la fonction `voisines`.

7

On pourra choisir la distance euclidienne pour h.

8

Comme dans l'algorithme de Dijkstra, on peut trouver un plus court chemin au moyen d'une liste de prédécesseurs mise à jour tout au long de l'algorithme.

9

Comme dans l'algorithme de Dijkstra, on peut trouver un plus court chemin au moyen d'une liste de prédécesseurs mise à jour tout au long de l'algorithme.

10

il est important de passer beaucoup de temps sur les exemples donnés avant de commencer à écrire des fonctions ou des algorithmes.