Graphes (Graphs)
Implémentations et parcours

1 Représentations / Implémentations

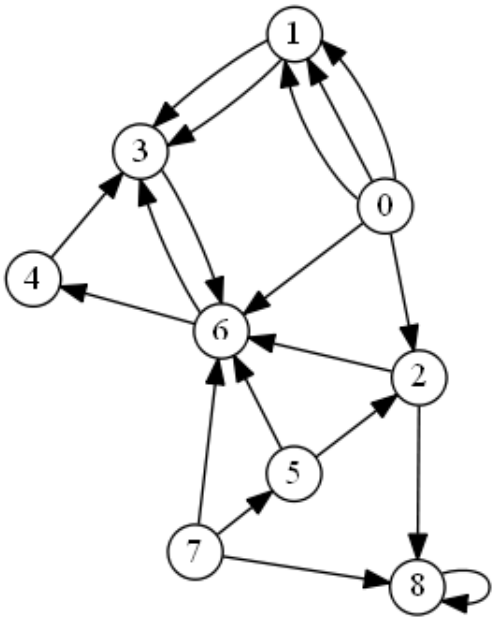


FIGURE 1 – Graphe orienté (Digraph) G'_1

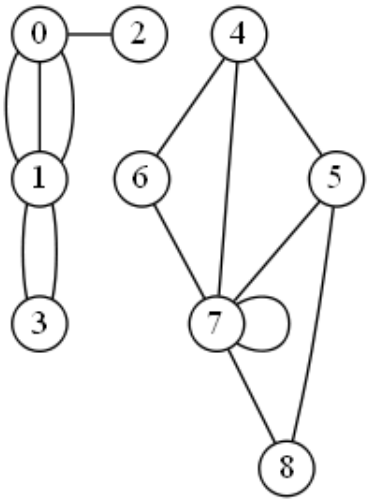


FIGURE 2 – Graphe non orienté (Graph) G'_2

Exercice 1.1 (`GraphMat` : Matrice d'adjacence)

On utilise pour cette première implémentation, la représentation par matrice d'adjacence.

1. Quelles sont les différences, pour l'implémentation, lorsque le graphe est orienté ou non orienté, valué ou non, possède des liaisons simples ou multiples ?
2. Donner les matrices d'adjacence représentant les graphes des figures 1 et 2.
3. Sachant que l'on veut pouvoir utiliser le même type pour représenter un graphe qu'il soit orienté ou non, simple ou 1-graphe, ou encore un multigraphe ou p-graphe, que doit contenir l'implémentation ?

Exercice 1.2 (`Graph` : Listes d'adajcence)

1. Quelle est l'autre manière de représenter / implémenter un graphe ?
2. Quelles sont les différences pour l'implémentation lorsque le graphe est orienté ou non orienté, possède des liaisons multiples ?
3. Donner les représentations des graphes des figures 1 et 2.
4. Sachant que l'on veut pouvoir utiliser le même type pour représenter n'importe quel type de graphe : orienté ou non, simple (ou 1-graphe) ou multigraphe (ou p-graphe) ; que doit contenir l'implémentation ?

Exercice 1.3 (dot)

Écrire les fonctions qui construisent la représentation au format `dot` (simplifié) dans les deux implémentations.

Exemples :

- Graphe G'_1 (figure 1)

```
>>> print(todot(G1))
digraph {
  0 -> 1
  0 -> 2
  0 -> 6
  1 -> 3
  2 -> 6
  2 -> 8
  3 -> 6
  4 -> 3
  5 -> 2
  5 -> 6
  6 -> 3
  6 -> 4
  7 -> 5
  7 -> 6
  7 -> 8
  8 -> 8
}
```

- Graphe G'_2 (figure 2)

```
>>> print(todot(G2))
graph {
  1 -- 0
  1 -- 0
  1 -- 0
  2 -- 0
  3 -- 1
  3 -- 1
  5 -- 4
  6 -- 4
  7 -- 4
  7 -- 5
  7 -- 6
  7 -- 7
  8 -- 5
  8 -- 7
}
```

Exercice 1.4 (Load)

Notre format de fichier `GRA` est un fichier texte contenant :
- une première ligne contenant 0 ou 1 : 0 pour non orienté, 1 pour orienté
- une seconde ligne contenant l'ordre du graphe
- une suite de lignes contenant les liaisons : deux numéros de sommets séparés par un espace

Voir les fichiers fournis : `digraph1.gra graph2.gra`.

Écrire les fonctions qui construisent un graphe à partir d'un fichier `".gra"` dans les deux implémentations.

Notes : Sauf indications particulières, les graphes utilisés dans les exercices suivants seront des graphes simples pour les non orientés, et des 1-graphes sans boucles pour les orientés. Les exemples utilisés ici seront les graphes G_1 (1-graphe sans boucles issu de G'_1) et G_2 (graphe partiel simple issu de G'_2).

---

2 Parcours

Exercice 2.1 (Parcours largeur (Breadth-first search))

1. Donner les forêts couvrantes obtenues lors des parcours largeur complets des graphes G_1 et G_2 à partir du sommet 0 puis à partir du sommet 7 (les sommets sont choisis en ordre croissant).
2. Donner le principe de l'algorithme de parcours largeur. Comparer avec le parcours d'un arbre général.
3. Comment stocker la forêt couvrante de manière linéaire ?
4. Écrire les fonctions de parcours largeur, avec construction de la forêt couvrante, pour les deux implémentations.

Exercice 2.2 (Parcours profondeur (Depth-first search))

1. Donner les forêts couvrantes obtenues lors des parcours profondeur des graphes G_1 et G_2 à partir du sommet 0 puis du sommet 7 (les sommets sont choisis en ordre croissant).
2. Donner le principe de l'algorithme récursif du parcours profondeur. Comparer avec le parcours d'un arbre général.
3. **Graphes non orientés**
   (a) Quels sont les différents types d'arcs rencontrés lors du parcours profondeur d'un graphe non orienté ?
   Classer les arcs du parcours profondeur de G_2 effectué en question 1 et ajouter les arcs manquants à la forêt.
   (b) Que faut-il ajouter au parcours profondeur pour repérer les différents types d'arcs ?
   (c) Écrire la fonction du parcours profondeur d'un graphe non orienté et représenté par une matrice d'adjacence. Indiquer au cours du parcours, le type des arcs rencontrés.
4. **Graphes orientés**
   (a) Quels sont les différents types d'arcs rencontrés lors d'un parcours profondeur ?
   Classer les arcs du parcours profondeur de G_1 effectué en question 1 et ajouter à la forêt couvrante les arcs manquants.
   Comment peut-on reconnaître le type d'un arc ?
   (b) On utilise la notion d'ordre préfixe de visite (première rencontre) et ordre suffixe de visite (dernière rencontre). En numérotant les sommets suivant ces deux ordres avec un unique compteur, écrire les conditions de classification des arcs.
   (c) Écrire la fonction du parcours profondeur d'un graphe orienté et représenté par listes d'adjacence. Indiquer au cours du parcours, le type des arcs rencontrés.
5. **Bonus**
   Le parcours profondeur peut être fait en itératif.
   Donner le principe d'un tel parcours et écrire la fonction correspondante pour un graphe orienté en représentation par listes d'adjacence.

3 Applications

Exercice 3.1 (Chemin – *Final S3# 2017*)

1. Comment trouver un chemin (ou une chaîne) entre deux sommets donnés dans un graphe ? Donner au moins deux méthodes différentes, les comparer.

2. Écrire une fonction qui cherche un chemin entre deux sommets. Si un chemin a été trouvé, il devra être retourné (une liste de sommets).

Exercice 3.2 (Graphes bipartis (Bipartite graph) – *Final S3 - 2017*)

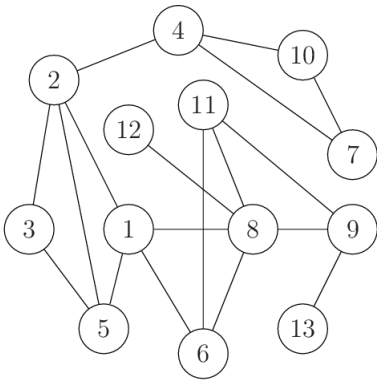


FIGURE 3 – Graphe G_3

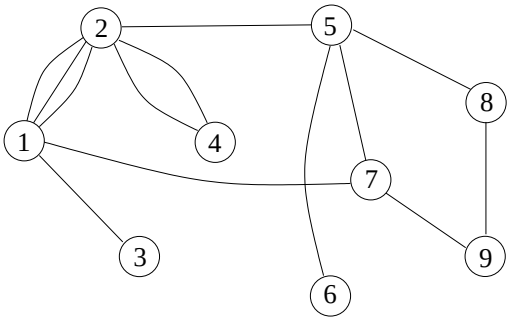


FIGURE 4 – Graphe G_4

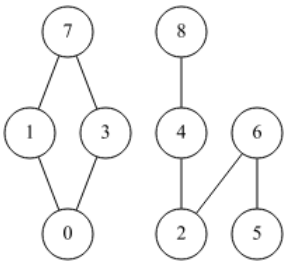


FIGURE 5 – Graphe G_5

Un graphe biparti est un graphe (un multigraphe) non orienté $G =< S, A >$, dans lequel S peut être partitionné en deux ensembles S_1 et S_2 tels que $(u, v) \in A$ implique soit que $u \in S_1$ et $v \in S_2$, soit que $u \in S_2$ et $v \in S_1$. Aucune arête ne doit relier deux sommets d'un même ensemble.

1. Les graphes des figures 3 à 5 sont-ils bipartis ? Pour chaque graphe biparti, donner les deux ensembles S_1 et S_2.

2. Écrire une fonction qui teste si un graphe est biparti.

Exercice 3.3 (Acyclic – *Final S3 - 2017*)

Définition :
Un graphe orienté *acyclique* est un graphe sans circuits.

Écrire la fonction `is_acyclic` qui vérifie si un graphe orienté représenté par listes d'adjacence est *acyclique*.

Exercice 3.4 (I want to be tree – *Final S3# - 2018*)

Définition :

Un **arbre** est un graphe **connexe sans cycle**.

Écrire la fonction `isTree` qui vérifie si un graphe non orienté est un arbre.

Exercice 3.5 (Distances et centre – *Final S3# - 2018*)

Définitions :

- La **distance** entre deux sommets d'un graphe est le nombre d'arêtes d'une **plus courte chaîne** entre ces deux sommets.
- On appelle **excentricité** d'un sommet x dans un graphe $G =< S, A >$ la quantité :

$$\text{exc}(x) = \max_{y \in S}\{distance(x, y)\}$$

- Le **rayon** d'un graphe est l'excentricité minimale de ses sommets, c'est-à-dire la plus petite distance à laquelle puisse se trouver un sommet de tous les autres.
- Le **centre** d'un graphe est formé de l'ensemble de ses sommets dont l'excentricité est le rayon du graphe (donc les sommets d'excentricité minimale).

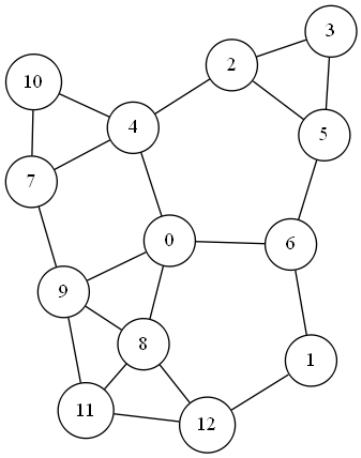
Écrire la fonction `center(G)` qui retourne le centre du graphe G (une liste).

Pour le graphe G_6 :

Les sommets 0, 4 et 6 ont pour excentricité 3.
Les sommets 3 et 10 ont pour excentricité 5.
Les sommets restants ont pour excentricité 4.

Le rayon de G_6 est donc 3 et son centre est constitué des sommets 0, 4 et 6.

```
>>> center(G6)
[0, 4, 6]
```



Graphe G_6

Exercice 3.6 (Compilation, cuisine...)

1. *Ordonnancement , un exemple simple :*
   Supposons l'ensemble d'instructions suivantes à effectuer par un seul processeur :

   ⓪ lire(a)
   ① b ← a + d
   ② c ← 2 * a
   ③ d ← e + 1
   ④ lire(e)
   ⑤ f ← h + c / e
   ⑥ g ← d * h
   ⑦ h ← e - 5
   ⑧ i ← h - f

   Quels sont les ordres possibles d'exécution ?
   Comment représenter ce problème sous forme de graphe ?

   Chaque solution correspond à un *tri topologique* du graphe.

2. Quelle doit être la propriété du graphe pour qu'une solution de tri topologique existe ?

3. Si on dessine le graphe en alignant les sommets dans l'ordre d'une solution de tri topologique, que peut-on constater ?

4. (a) Soit $suffix$ le tableau contenant les dates de dernière visite : l'ordre suffixe de tous les sommets de G lors d'un parcours en profondeur.
Démontrer que pour une paire quelconque de sommets distincts $u, v \in S$, s'il existe un arc dans G de u à v, et si G a la propriété de la question 2, alors $suffix[v] < suffix[u]$.

   (b) En déduire un algorithme qui trouve une solution de tri topologique pour un graphe (on supposera qu'une solution existe).

   (c) Que faut-il changer à cet algorithme pour vérifier l'existence d'une solution dans un graphe quelconque ?

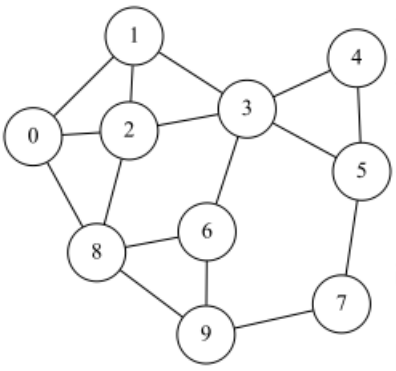
   (d) Écrire la fonction Python qui retourne une solution de tri topologique sous la forme d'une liste de sommets.

Et la cuisine dans tout ça ?

Exercice 3.7 (What is this ? – *Final S3# - 2018*)

Soit la fonction suivante :

```
1  def buildGraph(G, s, n):
2      map = [None] * G.order
3      dist = [-1] * G.order
4      NG = graph.Graph(1)
5      dist[s] = 0
6      map[s] = 0
7      q = queue.Queue()
8      q.enqueue(s)
9      while not q.isempty():
10         s = q.dequeue()
11         for adj in G.adjlists[s]:
12             if (dist[adj] == -1) and (dist[s] < n):
13                 dist[adj] = dist[s] + 1
14                 map[adj] = NG.order
15                 NG.addvertex()
16                 q.enqueue(adj)
17             if dist[adj] != -1:
18                 NG.addedge(map[s], map[adj])
19     return (NG, dist, map)
```



Graphe G_7

1. On appelle cette fonction avec `build_graph(`G_7`, 4, 2)` (avec G_7 le graphe ci-dessus, dont les listes de successeurs sont en ordre croissant de numéros).

   (a) Remplir le vecteur `dist`.

   (b) Remplir le vecteur `map`.

   (c) Dessiner le graphe résultat (NG).

2. `build_graph(`G`, `s`, `n`)` est appelée avec G un graphe quelconque (non vide), s un sommet de G, et n un entier naturel non nul.

   (a) Que représente le vecteur `dist` ?

   (b) À quoi sert le vecteur `map` ?

   (c) Que représente le graphe NG ?