TP n° 11 – Construction de graphes, parcours en largeur

1 Construction de graphes

Commençons par rappeler qu'un graphe non orienté G est un objet du type $G = (S, \mathcal{A})$ où :

— S est un ensemble fini appelé ensemble des **sommets** du graphe G ;
— $\mathcal{A} \subset \{\{s,t\} \mid (s,t) \in S \times S \text{ et } s \neq t\}$ est appelé ensemble des **arêtes** du graphe G.

Pour définir un graphe, on utilisera un dictionnaire. Un dictionnaire est un type de données dynamiques opérant sur les éléments d'un ensemble totalement ordonné, appelés les **clés**. À chaque clé on associe une **valeur**. Les clés sont uniques et doivent être des objets non mutables.

Le type dictionnaire (`dict`) est présent nativement dans Python. Un dictionnaire `d` est constitué de deux ensembles : un ensemble de clés `d.keys()` et un ensemble de valeurs `d.values()`. Il est caractérisé par les opérations :

— recherche/appartenance d'une clé ;
— l'association clé–valeur (insertion) ;
— la suppression d'une clé.

L'intérêt d'un dictionnaire est de pouvoir trouver (ou tester la présence) d'une clé très rapidement. Comme les dictionnaires utilisent des tables de hachage, le temps moyen de recherche d'un élément est en $O(1)$. Une table de hachage est une généralisation de la notion de tableau, seulement au lieu d'utiliser la clé directement comme indice du tableau, l'indice est calculé (haché) à partir de la clé en utilisant une **fonction de hachage**). Une table de hachage utilise donc un tableau de taille proportionnelle au nombre de clés effectivement stockées.

On donne ci-dessous la syntaxe des opérations élémentaires.

— Créer un dictionnaire vide :

```
d = {}
d = dict()
```

— Associer une valeur `v` à la clé `c` :

```
d[c]=v
```

Si une valeur était déjà associée à `c`, la nouvelle écrase l'ancienne.

— Renvoyer la valeur associée à la clé `c` :

```
d.get(c)
```

Cette commande renvoie `None` si `c` n'est pas une clé de `d`. Il est aussi possible d'utiliser

```
d[c]
```

mais une erreur est renvoyée si aucune valeur est associée à `c`.

— Tester l'appartenance d'une clé, voir si `c` est une clef de `d` :

```
c in d
c in d.keys()
```

— Supprimer l'entrée correspondant à `c` dans `d` et renvoyer la valeur associée à la clé supprimée :

```
d.pop(c, None)
```

où le second argument `None` évite de renvoyer un message d'erreur dans le cas où `c` n'est pas une clé de `d`. Dans le cas où `c` est de façon certaine une clé de `d`, il est possible d'utiliser

```
del d[c]
```

— enfin, on peut parcourir un dictionnaire :
avec ses clés :

```
for c in d:
    print(c)
```

par ses valeurs :

```
for v in d.values():
    print(v)
```

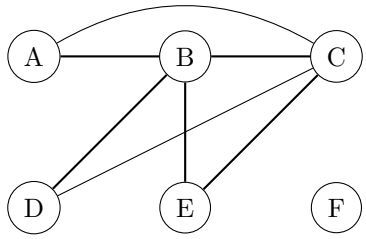
les deux simultanément :

```
for c,v in d.items():
    print(c,v)
```

Attention, les clés utilisées doivent être de type non-dynamique pour pouvoir être hachées, ce qui est le cas des entiers, des flottants, des tuples, des chaînes de caractères, ... mais pas des listes ou des tableaux.

Définir un graphe sous forme de dictionnaire

On considère le graphe non orienté $G = (S, \mathcal{A})$ représenté ci-dessous :



Dans ce qui suit, on cherche à le définir avec la suite de commandes :

```
>>> G=sommets([chr(65+k) for k in range(6)])
>>> G
{'A': {'adj': []}, 'B': {'adj': []}, 'C': {'adj': []}, 'D': {'adj': []},
'E': {'adj': []}, 'F': {'adj': []}}
>>> aretes(G,[("A","B"),("A","C"),("B","C"),("B","D"),("B","E"),("C","D"),("C","E")])
>>> G
{'A': {'adj': ['B', 'C']}, 'B': {'adj': ['A', 'C', 'D', 'E']},
'C': {'adj': ['A', 'B', 'D', 'E']}, 'D': {'adj': ['B', 'C']},
'E': {'adj': ['B', 'C']}, 'F': {'adj': []}}
```

tel qu'à chaque clé soit associé un dictionnaire avec `'adj'` comme clé et une liste des sommets adjacents, initialisée comme vide.

Exercice 11.1

1. Définir une fonction `sommet` qui prend comme arguments un dictionnaire `G` associé à un graphe et une chaîne de caractère `s` associée à un sommet et qui ajoute le sommet au dictionnaire du graphe.
2. Définir une fonction `sommets` qui prend comme argument une séquence de sommets et qui renvoie le dictionnaire initialisé associé.
3. Définir une fonction `arete` qui prend comme arguments un dictionnaire associé à un graphe `G` et deux sommets `s` et `t` et qui ajoute les sommets adjacents, si nécessaire.
   ℹ *On veillera en particulier à ne pas définir deux fois un sommet adjacent.*
4. Définir une fonction `aretes` qui prend comme arguments un dictionnaire associé à un graphe `G` et une séquence d'arêtes `A`, chacune donnée sous forme de couple de la forme `(s,t)` et qui complète, si besoin, le dictionnaire avec les sommets adjacents.
5. Définir une fonction `degre` qui prend comme arguments un dictionnaire `G` associé à un graphe et une chaîne de caractère `s` associée à un sommet et qui renvoie le degré du sommet $d(s)$, correspondant au nombre d'arêtes contenant ce sommet.
6. Expliquer pourquoi il n'est pas raisonnable d'utiliser la fonction `degre` pour calculer le degré de tous les sommets. En tenir compte pour définir une fonction `degres` qui prend comme argument un dictionnaire associé à un graphe `G` et ajoute le degré de chaque sommet dans son dictionnaire comme valeur associée à la clé `"d"`. Vérifier que vous ayez :

```
>>> degres(G); [(s,G[s]["d"]) for s in G]
[('A', 2), ('B', 4), ('C', 4), ('D', 2), ('E', 2), ('F', 0)]
```

2 Parcours en largeur

Dans ce qui suit, on souhaite implémenter le parcours en largeur d'un graphe $G = (S, \mathcal{A})$ dont le principe est :

1. on choisit un sommet de départ, noté r ;
2. on visite les sommets situés à une distance 1 de r, puis ceux à une distance 2, puis ceux à une distance 3, etc. ;
3. on s'arrête après avoir visité tous les sommets accessibles depuis r ;
4. on recommence avec un nouveau sommet de départ non visité (s'il en reste), jusqu'à avoir exploré tous les sous-graphes connexes de G.

Un parcours en largeur d'une composante connexe peut facilement être implémenté à l'aide d'une **file** qui stocke les sommets à traiter. Suivant le principe du FIFO (*First In, First Out*), une file permet :

— d'ajouter un élément à la fin de la file (enfiler) ;
— de retirer le premier élément de la file (défiler).

Pour manipuler des files, on utilisera le module `deque` :

```
from collections import deque
F = deque(S)     # initialiser une file avec une séquence S (liste, tuple)
F.append(e)      # enfiler un élément e
e = F.popleft() # défiler le premier élément, devenu e
```

Pour l'exploration d'une composante connexe d'un graphe G, depuis un sommet r, on utilisera un dictionnaire de parcours de la forme :

```
P = {
  <sommet>: {
    "adj": [<liste des sommets adjacents>],
    "vu": <étiquette, 1 si vu>,
    "d": <distance, en nombre de noeuds depuis la racine>,
    "p": <sommet parent lors du parcours>,
    },
  <autre sommet>: {
      ...
    }
}
```

qui permet de garder une mémoire du parcours sans modifier le dictionnaire d'un graphe `G` pour lequel les étiquettes `"vu"`, `"d"` ou `"p"` n'ont pas de sens car ce ne sont pas des caractéristiques intrinsèques. On intialisera le dictionnaire de parcours d'un graphe G à partir d'une copie profonde du dictionnaire du graphe `G` :

```
P = {s:G[s].copy() for s in G}
```

afin de ne pas lier les dictionnaires de chaque sommet (d'où le `G[s].copy()`). Ensuite, en début de parcours, on initialisera la file avec :

```
F = deque([r])
```

À chaque itération de l'algorithme, on défile le premier sommet de la file

```
v = F.popleft()
```

puis pour chacun de ses voisins s non encore exploré :

— on enfile s :

```
F.append(s)
```

— on marque s comme vu dans le dictionnaire de parcours :

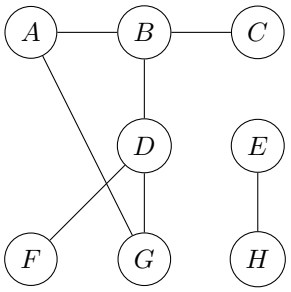
```
P[s]["vu"]=1
```

— éventuellement, on étiquette sur s sa distance à r (celle de son père à r incrémentée de 1) dans le dictionnaire de parcours :

```
P[s]["d"]=...
```

L'algorithme s'arrête lorsque la file est vide. On le relance alors sur un nouveau sommet non marqué (s'il en reste), jusqu'à avoir exploré toutes les composantes connexes du graphe.

On prendra comme premier exemple le graphe support du cours représenté ci-dessous



et que l'on peut définir avec :

```
G = sommets([chr(65+k) for k in range(8)])
aretes(G,[("A","B"),("A","G"),("B","C"),("B","D"),("D","F"),("D","G"),("E","H")])
```

Exercice 11.2

1. Définir une fonction `init_parcours` qui qui prend comme argument un dictionnaire `G` associé à un graphe et qui renvoie un dictionnaire de parcours avec, pour chaque sommet, une étiquette `"vu"` initialisée à 0.
2. Définir une fonction `exploration_composante` qui prend comme arguments un dictionnaire `G` associé à un graphe et un sommet de départ `r`, qui explore toute la composante connexe de r et renvoie une liste contenant les sommets explorés dans l'ordre depuis r.
   ℹ *On pourra afficher à chaque étape l'état de la file.*
3. Utiliser votre fonction pour explorer toute la composante connexe de A. Comparer vos résultats à ceux obtenus lors du parcours manuel en cours.
4. Définir une fonction `parcours_largeur` qui prend comme argument un dictionnaire `G` associé à un graphe et qui parcours tout le graphe et renvoie une liste contenant les sommets successivements explorés.
5. Utiliser votre fonction pour explorer tout le graphe et comparer vos résultats à ceux obtenus lors du parcours manuel en cours.

On souhaite maintenant compléter nos fonctions avec des notions de distances. Soit $G = (S, \mathcal{A})$ un graphe, $(s,t) \in S^2$ et $L_{s,t} \subset \mathbb{N}$ l'ensemble des longueurs des chemins de s à t. On appelle distance de s à t la valeur :

$$\begin{cases} \min(L_{s,t}) \text{ si } L_{s,t} \neq \emptyset \\ +\infty \text{ sinon} \end{cases}.$$

On pourra utiliser une notion d'infini avec le nombre `float('inf')`.

6. Modifier votre fonction `init_parcours` pour ajouter et initialiser des étiquettes de distance `"d"` et de sommet parent `"p"` au dictionnaire de parcours.
7. Définir une fonction `distance` qui prend comme arguments un dictionnaire `G` associé à un graphe et deux sommets `r` et `a` et qui renvoie la distance de r à a.
   Vérifier que vous avez :

   ```
   >>> distance(G,"A","F")
   3
   ```

8. Définir une fonction `chemin` qui prend comme arguments un dictionnaire `G` associé à un graphe et deux sommets `r` et `a` et qui renvoie le chemin sous forme de liste de sommets parcourus de r à a.
   Vérifier que vous avez :

   ```
   >>> chemin(G,"A","F"), len(chemin(G,"A","F"))-1==distance(G,"A","F")
   ['A', 'B', 'D', 'F'], True
   ```

* *
*