Correction du TD ipt^2 n° 4: Graphes
Implémentation - parcours

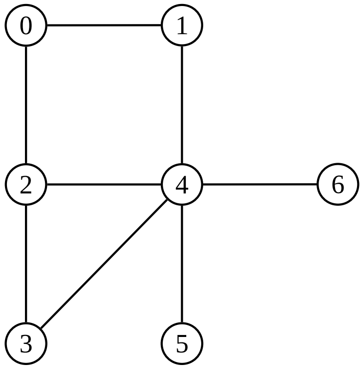
Exercice n°1: Implémentation d'un graphe par dictionnaire - analyse du graphe

On rappelle ici le graphe ainsi que les listes de ses sommets et arêtes:

$$V = [0, 1, 2, 3, 4, 5, 6]$$

et une liste d'arêtes (sous forme de liste de listes):

$$A = [[0, 1], [0, 2], [1, 4], [2, 4], [2, 3], [4, 3], [4, 5], [4, 6]]$$



❶ La fonction va simplement itérer sur la liste des sommets, puis pour chaque sommet i, examiner à l'aide des listes d'arêtes les autres sommets reliés à i pour les intégrer comme valeurs du dictionnaire:

Listing 1: construc_Dicto(V,A)

```python
def construc_Dicto(V,A):
    n=len(V)
    dict={}
    for cle in V:
        dict[cle]=[]
    for el in A:
            dict[el[0]].append(el[1])
            dict[el[1]].append(el[0])
    return dict
```

On peut également éviter la première boucle `for` en implémentant les listes vides du dictionnaire à la volée lorsque le sommet i est rencontré pour la première fois:

Listing 2: construc_Dicto(V,A)

```python
def construc_Dicto_bis(V,A):
    dict={}
    for el in A:
        if el[0] not in dict:
            dict[el[0]]=[]
        if el[1] not in dict:
            dict[el[1]]=[]
        dict[el[0]].append(el[1])
        dict[el[1]].append(el[0])
    return dict
```

On obtient la sortie suivante avec le graphe proposé:

```
>>> print(construc_Dicto(V,A))
{0: [1, 2], 1: [0, 4], 2: [0, 4, 3],
 3: [2, 4], 4: [1, 2, 3, 5, 6], 5: [4], 6: [4]}
>>> print(construc_Dicto_bis(V,A))
{0: [1, 2], 1: [0, 4], 2: [0, 4, 3],
4: [1, 2, 3, 5, 6], 3: [2, 4], 5: [4], 6: [4]}
```

❷ Une solution simple est d'adapter le code rédigé en question 1; celui-ci doit simplement recenser et écrire comme valeur de la clé i (représentant le sommet i) le nombre de sommets adjacents rencontrés dans la liste A pour le sommet i:

Listing 3: analyse_sommets(V,A)

```python
def analyse_sommets(V,A):
    dict={}
    for el in A:
        if el[0] not in dict:
            dict[el[0]]=0
        if el[1] not in dict:
            dict[el[1]]=0
        dict[el[0]]+=1
```

```python
        dict[el[1]]+=1
return dict
```

>>> print(u"Détermination du degré de chaque sommet:",analyse_sommets(V,A))
`{0: 2, 1: 2, 2: 3, 4: 5, 3: 2, 5: 1, 6: 1}`

On peut faire encore plus simple en exploitant l'un des deux codes de la question 1, par exemple `analyse_sommets_bis(V,A)` avec:

Listing 4:

```python
def analyse_sommets_bis(V,A):
    dictio=construc_Dictio_bis(V,A) #construit le dictionnaire du graphe
    for clé in dictio:
        dictio[clé]=len(dictio[clé]) # associe à chaque sommet (clé) le nombre d'arête(s) dont il est extrémité (degré).
    return dictio
```

Exercice n°2: Implémentation d'une file de priorité à l'aide d'un arbre binaire organisé en tas - utilisation dans l'algorithme de Dijkstra

Les trois relations de récurrence pour naviguer dans les noeuds adjacents du graphe sont élémentaires avec: $k \to (k-1)//2$ pour le trajet fils->père, puis $k \to 2k+1$ pour le trajet père->fils gauche, et enfin $k \to 2k+2$ pour le trajet père->fils droit; les fonctions correspondantes s'écrivent donc:

Listing 5:

1.
```python
def fils_gauche(k):
        return 2*k+1
```

Listing 6:

2.
```python
def fils_droit(k):
        return 2*k+2
```

Listing 7:

3.
```python
def pere(k):
        return (k-1)//2
```

4. On propose pour la fonction `mini`:

Listing 8:

```python
def mini(T,i):
        if i<0 or i>=len(T):
                return "erreur"
        else:
                mini,iMini=T[i],i
                if fils_gauche(i)<len(T) and T[fils_gauche(i)]<mini:
                        mini,iMini=T[fils_gauche(i)],fils_gauche(i)
                if fils_droit(i)<len(T) and T[fils_droit(i)]<mini:
                        mini,iMini=T[fils_droit(i)],fils_droit(i)
        return iMini
```

5. Cette fonction est évidemment élémentaire:

Listing 9:

```python
def permute(L,i,j):
        L[i],L[j]=L[j],L[i]
```

6. On rappelle la fonction:

Listing 10:

```python
def entasser_min(T,i):
        iMini=mini(T,i)
        if iMini!=i:
                permute(T,i,iMini)
                entasser_min(T,iMini)

```

qui reçoit la liste T et un indice $0 \leq i < len(T)$.

La fonction `entasser_min(T,i)` est récursive; elle commence par repérer l'indice *imini* du minimum entre le nœud père d'indice i et ses fils d'indice $fils(gauche(i))$ et $fils(droit(i))$. Si le minimum n'est pas le "père" alors on permute les éléments $T[imini]$ et $T[i]$, et la fonction est relancée avec comme paramètre i, l'indice *imini*. La fonction procède donc à un "entassement partiel" qui se propage vers le bas dans l'éventualité où la position i du père n'est jamais la position du minimum des trois valeurs père, fils gauche, et fils droit. Le cas de base est implicite: la fonction termine si le père est déjà le minimum des trois valeurs.

7. Dans le meilleur des cas, le premier appel à la fonction `mini(T,i)` renvoie i, c'est à dire que l'indice du minimim entre $T[i]$,$T[versdroite(i)]$, et $T[versgauche(i)]$ est i; l'entrée dans la boucle conditionnelle ne se faisant pas, la complexité est:

$$C_{meilleur} = \mathcal{O}(1)$$

8. Le calcul de la complexité dans le pire des cas est un peu technique:

   Pour cette étude, c'est à dire que l'entassement partiel se propage jusqu'à la base de l'arbre, cherchons par exemple l'évaluation de la complexité pour un parcours complet de l'arborescence en passant toujours "à droite".

   Appelons u_k l'indice de l'élément le plus à droite du niveau k; on a donc la récurrence suivante:

$$u_k = 2\left(u_{k-1}+1\right) = 2\left(2\left(u_{k-2}+1\right)+1\right)$$
$$= 2^2 u_{k-2} + 2^2 + 2^1 = 2^k \underbrace{u_0}_{=0} + 2^k + 2^{k-1} + \ldots + 2^1$$
$$= 2^k + 2^{k-1} + \ldots 2^1 = 2\frac{1-2^k}{1-2} = 2^{k+1} - 2$$

   Pour le parcours complet on a donc:

$$n = u_h = 2^{h+1} - 2 \;\Rightarrow\; h = \ln_2(n+2) - 1$$

   La complexité en gros correspond donc à l'ordre de grandeur du nombre d'étapes du parcours, c'est à dire la hauteur de l'arbre.

$$C_{pire}(droite) = \mathcal{O}(\ln_2(n))$$

   Si l'on reprend ce calcul en supposant cette fois un passage systématique "à gauche" dans la descente de l'arborescence, il vient:

$$u_k = 2u_{k-1} + 1 = 2\left(2u_{k-2}+1\right) + 1 = 2^2 u_{k-2} + 2 + 1$$
$$= 2^k \underbrace{u_0}_{=0} + 2^{k-1} + 2^{k-2} + \ldots + 1 = \frac{1-2^k}{1-2} = 2^k - 1$$

   Soit pour le parcours complet:

$$n = u_h = 2^h - 1 \;\Rightarrow\; h = \ln_2(n+1)$$

   donc la complexité:

$$C_{pire}(gauche) = \mathcal{O}(\ln_2(n))$$

   Finalement, la complexité en gros de la fonction est donc: $C(n) = \mathcal{O}(\ln_2(n))$

9. Pour réaliser une file de priorité, il suffit d'exploiter la fonction `entasser_min(T,i)` dans une boucle partant de la dernière feuille en bas à droite de l'arbre et remontant celui-ci jusqu'à la racine $T[0]$; ainsi, à chaque étape, on a l'assurance que le sous-arbre dont le noeud courant de la boucle est la racine est organisé en tas "min"; une fois la boucle terminée, l'arbre entier est organisé en tas "min", c'est à dire que la plus petite valeur se trouve à la racine $T[0]$; cette structure peut ainsi servir de file de priorité "min"; on propose donc la fonction suivante:

   Listing 11:

```python
def faire_file_prio_min(T):
        for i in range(len(T)-1,-1,-1):
                entasser_min(T,i)
        return T
```

10. On propose en remplacement des lignes 9 et 10:

   Listing 12:

```python
9  liste_poids_sommets_restants=[(i,td[i]) for i in C]
10 a=faire_file_prio_min(liste_poids_sommets_restants)[0][0]
```

Exercice n°3: Détection des composantes connexes d'un graphe

Il s'agit simplement ici de modifier les fonctions `parcoursCO` et `parcoursPROF` afin de renvoyer la liste des composantes connexes.

❶ On peut proposer la fonction suivante qui ajoute le sommet i aux listes `coche` et `compco` puisqu'il est visité, et fait partie de la composante connexe en cours de parcours, puis examine récursivement tous les sommets de la composante connexe:

Listing 13: comp(matadj,i,coche=[],compco)

```python
def comp(matadj, i, coche=[], compco):
    n=matadj.shape[0]
    coche.append(i)
    compco.append(i)
    for j in range(n):
        if matadj[i,j]!=0 and j not in coche:
            compco=comp(matadj, j, coche, compco)
    return compco
```

Au fur et à mesure des appels récursifs, la liste `compco` se construit et est transmise à chaque appel après sa modification.

❷ Cette seconde fonction va explorer chaque composante connexe à l'aide d'une boucle itérative et de la fonction précédente:

Listing 14: composanteco(matadj)

```python
def compco(matadj):
    n=matadj.shape[0]
    coche=[]
    liste_compco=[]
    for i in range(n):
        if i not in coche:
            comp=comp(matadj, i, coche, [])
            liste_compco.append(comp)
    return liste_compco
```