TD IPT^2 n° 4: Révisions

Graphes

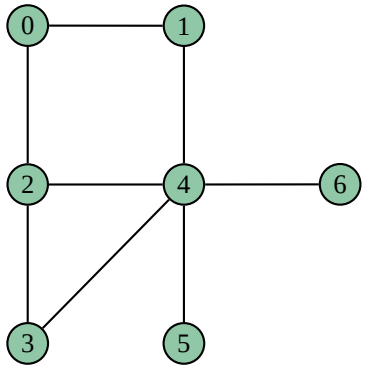
Exercice n°1: Implémentation d'un graphe par dictionnaire - analyse du graphe

On considère un graphe non orienté donné par une liste de ses sommets V et une liste d'arêtes A.

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

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 Proposer une fonction Python `contruc_Dicto(V,A)` qui renvoie le dictionnaire dont les clés représentent les sommets, et les valeurs associées les listes de voisins:

$$\{sommet_1 : [V_1, V_2, \ldots], sommet_2 : [V_1, V_5, \ldots], \ldots\}$$

❷ Ecrire une fonction Python `analyse_sommets(V,A)` qui reçoit là-encore les listes de sommets et d'arêtes V et A, et qui renvoie un dictionnaire dont les clés sont les sommets et les valeurs le degré associé à chacun.

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

Un arbre binaire est une catégorie particulière de graphe organisé en arborescence dans laquelle un nœud dit *père* (qui contient une valeur) possède exactement deux nœuds "fils" (contenant également chacun une valeur). On peut l'implémenter par une simple liste, et le parcours de l'arbre peut facilement se faire à l'aide de relations de récurrence élémentaires entre les indices des éléments de la liste. Ci-dessous, un exemple de représentation en arbre binaire d'une liste de valeurs entières quelconque:

indice	0	1	2	3	4	5	6	7
T[indice]	5	2	6	0	1	9	1	5

Figure 1: Liste des valeurs

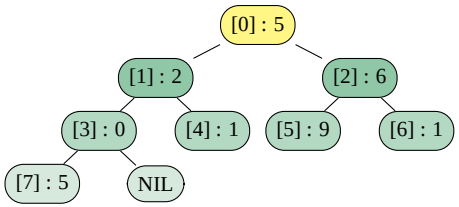


Figure 2: Implémentation de la liste en arbre binaire

NB: on désignera par `NIL` (abréviation du latin *nihil*, "rien".) tout sous-arbre vide.

Par exemple, dans l'arbre binaire ci-dessus, les nœuds [3] et [4] sont les *fils* du nœud [1].

1. Écrire une fonction python `fils_gauche(k:int)→ int` recevant en argument l'indice k d'un nœud de l'arbre et renvoyant l'indice du fils gauche.
2. Écrire de même une fonction python `fils_droit(k:int)→ int` qui renvoie l'indice du nœud *fils* droit du nœud k de l'arbre .
3. Enfin, rédiger une fonction python `pere(k:int)→ int` qui renvoie l'indice du nœud *père* du nœud d'indice k.
   On va chercher dans cet exercice à structurer un arbre binaire en une de ses classes particulières appelée *tas*, qui permet notamment l'implémentation d'une file de priorité (très utile, par exemple dans l'algorithme de Dijkstra).

Définition: *On appelle* **tas** *"min" un arbre* **presque complet** *représenté par une liste d'entiers T telle que pour tous les indices $i > 0$, la valeur $T[i]$ est supérieure ou égale à celle de $T[verspere(i)]$.*

Une fois transformée en tas "min", l'arbre initial devient donc:

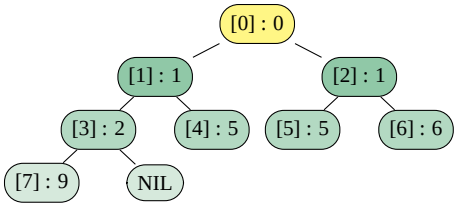


Figure 3: Arborescence organisée en tas

4. Rédiger une fonction Python `mini(T,i)` avec $0 \leq i < len(T)$ qui renvoie *iMini* (strictement inférieur à $len(T)$), indice de la plus petite valeur parmi $T[i]$, $T[fils\_gauche(i)]$, $T[fils\_droite(i)]$. En cas de valeurs égales, le plus petit indice est renvoyé.
5. Rédiger une fonction Python `permute(T:list,i:int,j:int)` → `None` qui permutent les valeurs $T[i]$ et $T[j]$ de la liste T.
6. On considère la fonction Python récursive suivante:

Listing 1:

```
1 def entasser_min(T,i):
2   iMini=mini(T,i)
3   if iMini!=i:
4     permute(T,i,iMini)
5     entasser_min(T,iMini)
```

qui reçoit la liste T et un indice $0 \leq i < len(T)$. Expliquer précisément ce qu'elle réalise.

7. Evaluer la complexité de l'algorithme `entasser_min(T,i)` dans le meilleur des cas.
8. On cherche désormais à évaluer sa complexité dans le pire des cas. Pour cela, on numérotera les niveaux de l'arborescence de $i = 0$, le sommet, à $i = h$, le dernier niveau (h est appelé *hauteur de l'arbre*); Dans un premier temps, on calculera la complexité lorsque l'on parcourt celui-ci en totalité et en passant systématiquement **"à droite"** lors de la descente d'un niveau dans l'arborescence. On supposera pour cela que l'arbre est **complet**, c'est à dire que chacun de ses niveaux comporte le nombre maximum possible d'éléments.

   Faire de même lorsque l'on parcourt l'arborescence en passant systématiquement **à gauche** à chaque descente d'un niveau.

   Conclure finalement sur la complexité.
9. Proposer une fonction Python `faire_file_prio_min(T:list)` → `list` exploitant `entasser_min(T,i)` et qui transforme la liste T initiale en un tas qui sera renvoyé. Proposer une utilisation d'un tel algorithme.
10. **Application:** l'une des étapes de l'algorithme de Dijkstra consiste à classer les éléments de la liste des nœuds **non encore visités** par ordre croissant des "distances" au nœud de départ. On rappelle ici la version vue en cours:

Listing 2:

```
1  def Dijkstra(matadj,v):
2    N,infini=max(matadj.shape),matadj[0,0]
3    #### Construction de A,C,tp, td ####
4    A,C=[],list(range(0,N))
5    td=np.ones(N,dtype=int)*infini
6    td[v]=0
7    tp=np.ones(N,dtype=int)*v
8    while C!=[]:
9      C.sort(key=lambda i:td[i]) #tri de C selon les poids croissants de td
10     a=C[0]
11     for c in C:
12       if td[a]+matadj[a,c]<td[c]:
13           td[c]=td[a]+matadj[a,c]
14             tp[c]=a
15     A.append(a)
16     C.remove(a)
17   return v,tp,td
```

Les lignes 9 et 10 de ce code réalisent le tri de la liste C (sommets non encore visités) suivant les poids croissants de *td* (ligne 10), puis l'extraction du nœud le plus proche du nœud de départ (ligne 11).

Proposer une modification de ces lignes ainsi que de la fonction `mini(T,i)` (à minima pour cette dernière) afin de réaliser cette même opération mais en faisant cette fois appel à une **file de priorité** avec la fonction `faire_file_prio_min`.

NB: on remarquera la nécessité ici d'intégrer dans la liste T à organiser en tas **non seulement les poids** $td[i]$ **comme valeurs permettant les comparaisons dans la fonction** `mini`, mais également **le nœud** i qui se trouve au poids $td[i]$ du nœud de départ (car il faudra ensuite l'ajouter à A et le retirer à C); pour cela, les éléments de T seront des tuples: $T = [....., (i, td[i]), ...]$

Remarque: lors de l'étape de tri de la liste C des sommets non visités, la seule contrainte imposée par l'algorithme de Dijkstra est de placer en première position le numéro du sommet le plus proche du sommet de départ. La file de priorité "min" assure naturellement ceci, mais l'organisation en arbre n'est évidemment pas une nécessité. Aussi, on peut proposer le code itératif suivant qui se contente de placer le sommet de poids minimal (parmi les sommets restants!) en première position de C:

Listing 3:

```
def poids_min_selon_poids_td(C,td):
    poids_min=td[C[0]]
    for k in range(1,len(C)):
        if td[C[k]]<poids_min:
            poids_min=td[C[k]]
            C[0],C[k]=C[k],C[0]
    return C
```

on notera cependant une complexité en $O(n)$ nettement moins favorable.

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

On rappelle ici les deux fonctions python de parcours d'une composante connexe d'un graphe et son parcours en profondeur à partir de la matrice d'adjacence du graphe `matadj`:

Listing 4: Parcours d'une composante connexe

```
def parcoursCO(matadj,i,coche=[]):
    nouvcoche=coche
    nouvcoche.append(i)
    n=matadj.shape[0]
    for j in range(0,n):
        if matadj[i,j]!=0 and j not in nouvcoche:
            print(u'appel récursif avec i='+str(j)+' venant du noeud'+str(i))
            nouvcoche=parcoursCO(matadj,j,nouvcoche)
    print(u'on est au sommet '+str(i)+u' sans sommet(s) adjacent(s) non coché(s)')
    return nouvcoche
```

Listing 5: Parcours en profondeur

```
def parcoursPROF(matadj):
    coche=[]
    n=matadj.shape[0]
    for i in range(0,n):
        if i not in coche:
          print ('appel principal avec i='+str(i))
          coche=parcoursCO(matadj,i,coche)
    return None
```

❶ Ecrire une fonction python **<u>récursive</u>** `comp(matadj,i,coche=[])`, recevant la matrice d'adjacence du graphe, le sommet i, qui parcourt le graphe pour construire **en place** la composante connexe de i dans la liste `compco`, modifie en place la liste `coche` et renvoie finalement `compco`.

❷ Construire finalement une fonction `compco(matadj)` qui renvoie la liste des composantes connexes du graphe, donc une liste de listes.