V

Révisions 5 : Rappels sur les graphes, implémentation et parcours - Algorithme de Dijkstra

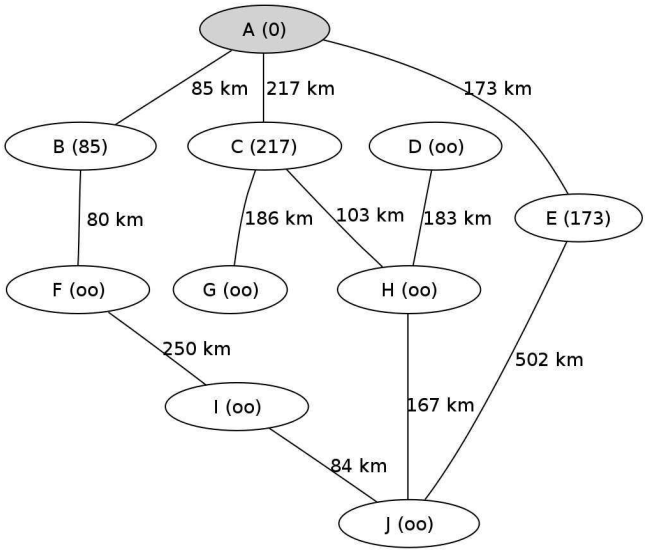


FIGURE V.1 – Edsger Dijkstra, prix Turing 1972

PLAN DU CHAPITRE

1 Notion de graphes

1.1 Définition et représentation par matrice d'adjacence

a - Graphes non orientés

Définition 1-1: GRAPHE NON ORIENTÉ

On appelle **graphe non orienté** un couple (V, E) dans lequel V désigne un ensemble de **noeuds** ou **sommets** (**V***ertices* en anglais) et E l'ensemble des arêtes (**E***dges* en anglais) qui relient ces noeuds.

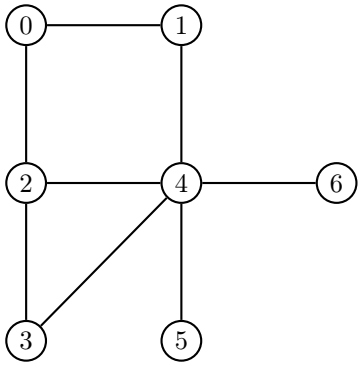


FIGURE V.2 – Exemple de graphe non orienté

On appelle :
- **chaine** : une suite d'éléments de **E**, donc une séries d'arêtes, reliant deux éléments de **V** donc deux sommets i_1 et i_n. Par exemple sur le graphe représenté ci-dessus, la chaîne $\{i_0, i_1\}, \{i_1, i_4\}, \{i_4, i_6\}$ relie les sommets 0 et 6.
- **sommets adjacents :** si ces derniers sont reliés par une arête
- **degré d'un sommet :** le nombre d'arêtes dont ce dernier est extrêmité
- **boucle :** un sommet relié à lui-même

MATRICES D'ADJACENCE :

La description d'un graphe peut se faire en en donnant simplement une représentation graphique telle que celle donnée ci-dessus ; cependant, l'implémentation de sa structure en machine nécessite d'introduire un nouvel objet formel permettant d'en manipuler les propriétés : **la matrice d'adjacence**.

Définition 1-2: MATRICE D'ADJACENCE

On appelle matrice d'adjacence du graphe non orienté G=(V, E) la matrice symétrique $E_G \in \mathcal{M}_n$ d'éléments e_{ij} définie par :

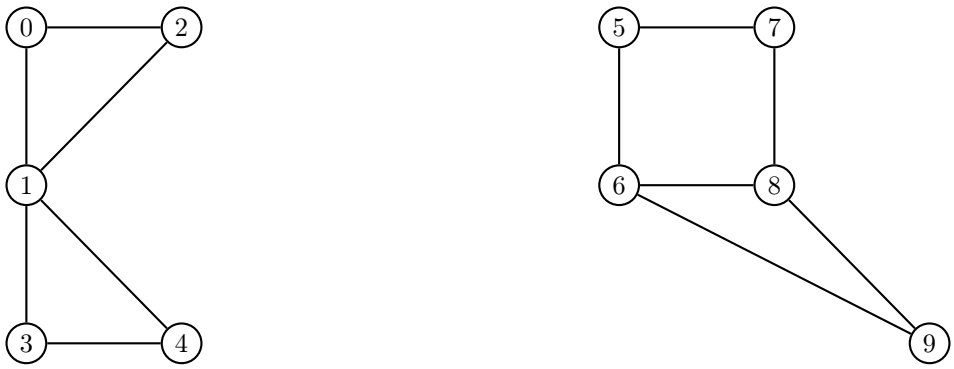
$$\begin{cases} e_{ij} = e_{ji} = 1 & \text{si } \{i, j\} \in E \\ e_{ij} = 0 & \text{sinon} \end{cases}$$

Exercice de cours: **(1.1) - n° 1.** *Bâtir la matrice d'adjacence E_G du graphe G donné en exemple ci-dessus*

Définition 1-3: GRAPHE CONNEXE

Un graphe (V, E) est dit connexe si pour tout couple de sommet (V_i, V_j) il existe une chaine de V_i à V_j.

Deux **sous graphes connexes** ou **composantes connexes** (sous-graphes) d'un **graphe non connexe** puisque les deux composantes ne sont reliées par aucune chaine :



b - Graphes orientés

Définition 1-4: GRAPHE ORIENTÉ

On appelle **graphe orienté** un couple (V, A) avec les éléments de A que l'on appelle **des arcs** (et non plus des arêtes) et qui sont des couples (i, j) et plus non plus des paires $\{i, j\}$ de sommets (leur ordre importe donc !).

On appelle :
- **chemin** une ensemble d'arcs qui amène d'un sommet i_1 à un sommet i_n. Ainsi, les chemins remplacent les chaines dans les graphes orientés.
- **degré entrant** $d^0_+(v)$ **:** le nombre d'arc(s) entrant(s) sur le noeud v.
- **degré sortant** $d^0_-(v)$ **:** le nombre d'arc(s) sortant(s) sur le noeud v.

MATRICES D'ADJACENCE :

Définition 1-5: MATRICE D'ADJACENCE D'UN GRAPHE ORIENTÉ

La matrice d'adjacence $E_G \in \mathcal{M}_n$ d'éléments e_{ij} du graphe orienté G=(V, E) est définie par (elle est nécessairement non symétrique cette fois !) :

$$\left[\begin{array}{ll} e_{ij} = 1 & \text{si } (i,j) \in E \\ e_{ij} = 0 & \text{sinon} \end{array}\right.$$

Exercice de cours: **(1.1) - n° 2.** *Déterminer la matrice d'adjacence du graphe orienté représenté ci-dessous :*

c - Graphes orientés pondérés

Définition 1-6: GRAPHE ORIENTÉ PONDÉRÉ

On appelle **graphe orienté pondéré** (ou graphe **valué**), un graphe (V, A) dans lequel on associe à chaque arc un poids w_{ij} (w pour *weight*). Un élément de A est maintenant noté (i, j, w_{ij}).

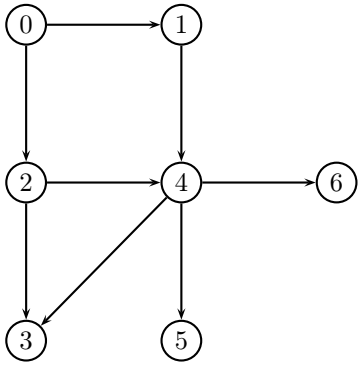


FIGURE V.3 – Exemple de graphe orienté

MATRICES D'ADJACENCE :

Définition 1-7: MATRICE D'ADJACENCE D'UN GRAPHE ORIENTÉ PONDÉRÉ

La matrice d'adjacence $E_G \in \mathcal{M}_n$ d'éléments e_{ij} du graphe orienté G=(V, A) est définie par :

$$\begin{cases} e_{ij} = w_{ij} & \text{si } (i, j, w_{ij}) \in A \\ e_{ij} = \star & \text{sinon} \end{cases}$$

avec $\star$ qui peut être tout sauf un poids du graphe (on peut prendre la somme des poids+n, l'objet NAN etc...)

Exercice de cours: **(1.1) - n° 3.** *Déterminer la matrice d'adjacence du graphe valué représenté ci-dessous :*

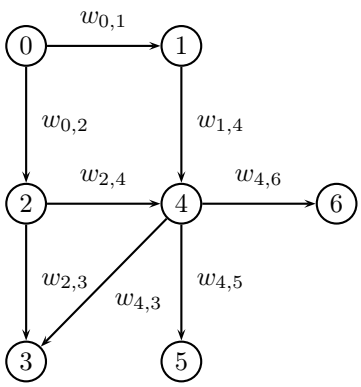


FIGURE V.4 – Exemple de graphe orienté pondéré

d - Code de construction d'une matrice d'adjacence

Exercice de cours: **(1.1) - n° 4.** *On donne le graphe précédent sous la forme d'une liste V de sommets :*

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

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

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

On donne également une liste A d'arcs correspondants au même graphe pondéré (là encore sous forme de listes de listes) :

$$A = [[0,1,"w01"],[0,2,"w02"],[1,4,"w14"],[2,4,"w24"],[2,3,"w23"],[4,3,"w43"],[4,5,"w45"],[4,6,"w46"]]$$

si le graphe est orienté pondéré.

1. *Ecrire une fonction Python* `construc_matNOR(V,A)` *permettant de construire la matrice d'adjacence du graphe non orienté (V, A).*
2. *Ecrire de même la fonction* `construc_matOR(V,A)` *qui pemet de construire la matrice d'adjacence du graphe orienté pondéré (V, A). On affectera par exemple aux termes correspondants à une absence d'arcs une valeur ne pouvant correspondre à aucun poids dans le graphe, par exemple la somme de tous les poids +1 :* $\sum_{i}^{len(A)} +1$

1.2 Algorithmes de parcours d'un graphe

Les problèmes exploitant les graphes en utilisant et en traitant leurs sommets et arêtes ou arcs nécessitent naturellement de pouvoir "naviguer" dans leur structure ; on parle de **parcours** des graphes :

- on part d'un sommet choisi arbitrairement (appartenant à un graphe connexe ou pas), et on le "coche"
- on choisit un sommet adjacent que l'on coche à son tour
- rendu sur un sommet où on ne peut plus poursuivre car tous ses sommets adjacents sont cochés, on retourne en arrière et on recommence

La procédure de parcours du graphe doit tenir compte de son caractère connexe ou pas :

a - Parcours d'une composante connexe (à partir d'un noeud)

On propose la procédure récursive suivante de parcours de la composante connexe comportant le sommet i :

Listing V.1 –

```
def parcoursCO(matadj,i,coche=[]):
    nouvcoche=coche #initialisation liste evolutive des coches
    nouvcoche.append(i) #que l'on incrémente du noeud i d'où l'on part
    n=matadj.shape[0] #on initialise le nombre maximum de noeuds du graphe
    for j in range(0,n): #on lance le parcours
        if matadj[i,j]!=0 and j not in nouvcoche: #on vérifie que le noeud suivant est adjacent et pas encore coché
            print('appel récursif avec i='+str(j)+' venant du noeud'+str(i)) #on affiche ce noeud adjacent
            nouvcoche=parcoursCO(matadj,j,nouvcoche) #prochaine récursion avec le nouveau noeud j et la liste des coch
    print(u'on est au sommet '+str(i)+u' sans sommet(s) adjacent(s) non coché(s)')
    return nouvcoche #on renvoie la liste finale des cochés
```

b - Parcours en profondeur ou *Depth First Search* (DFS)

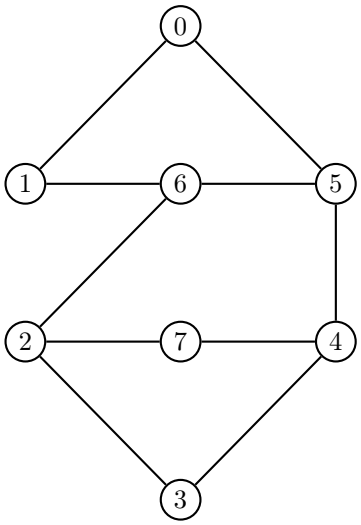
Pour réaliser le parcours en profondeur d'un graphe quelconque, en particulier un graphe non connexe, on peut bâtir un code exploitant la fonction `parcoursCO` (parcours de la composante connexe du noeud i) et permettant le "saut" d'une composante à l'autre afin de s'assurer de l'exploration totale du graphe :

Listing V.2 –

```
def parcoursPROF(matadj)
coche=[] #initialise la liste des coches
n=matadj.shape[0] #initialise la taille de la matrice d'adjacence
for i in range(0,n): #itere sur tous les sommets du graphe
    if i not in coche: # si i non coche
        print ('appel principal avec i='+str(i)) #annonce que l'on va explorer tous les sommets appartenant au graphe
        coche=parcoursCO(matadj,i,coche)
return None
```

c - Exemple de parcours d'un graphe connexe

On propose le graphe non orienté connexe suivant :



$$matadj = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}$$

On lance `parcoursCO(matadj,0)` qui renvoie :

```
appel récursif avec i=1 venant du noeud0
appel récursif avec i=6 venant du noeud1
appel récursif avec i=2 venant du noeud6
appel récursif avec i=3 venant du noeud2
appel récursif avec i=4 venant du noeud3
appel récursif avec i=5 venant du noeud4
on est au sommet 5 sans sommet(s) adjacent(s) non coché(s)s
appel récursif avec i=7 venant du noeud4
on est au sommet 7 sans sommet(s) adjacent(s) non coché(s)
on est au sommet 4 sans sommet(s) adjacent(s) non coché(s)
on est au sommet 3 sans sommet(s) adjacent(s) non coché(s)s
on est au sommet 2 sans sommet(s) adjacent(s) non coché(s)s
on est au sommet 6 sans sommet(s) adjacent(s) non coché(s)
on est au sommet 1 sans sommet(s) adjacent(s) non coché(s)
on est au sommet 0 sans sommet(s) adjacent(s) non coché(s)
[0, 1, 6, 2, 3, 4, 5, 7]
```

2 Le plus court chemin : algorithme de Dijkstra

2.1 Représentation des chemins par listes

Définition 2-1: POIDS D'UN CHEMIN

Dans un graphe orienté pondéré $G = (V, A)$, on considère un chemin allant du noeud $v = i_0$ au noeud $x = i_n$ (suite d'arcs : $\gamma = (i_0, i_1), (i_1, i_2), \ldots, (i_{n-1}, i_n)$). On appelle poids de ce chemin ou coût cumulé, la somme des poids des arcs qui le composent :

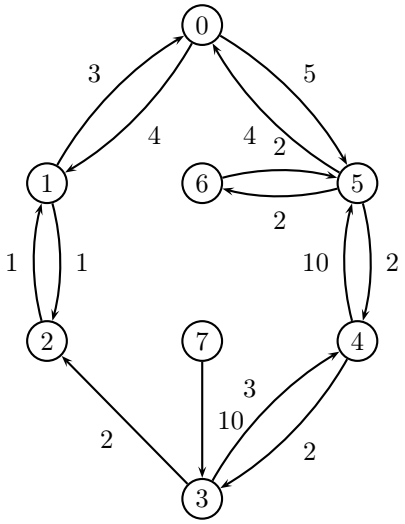
$$\text{coût}_\gamma = \sum_{k=0}^{n-1} w_{i_k, i_{k+1}}$$

Notre graphe G orienté et pondéré sera représenté par une matrice d'adjacence dans laquelle un terme m_{ij} est maximal si et seulement si il n'y a pas d'arc de i à j. Pour la suite, nous affecterons une valeur infini à ce maximum.

On peut remarquer que les chemins partant d'un noeud $v \in V$ qui n'ont pas de sommet en commun peuvent être représentés par une liste t_{pred} telle que :

$$\left[\begin{array}{ll} t_{pred}[i] = p & \text{si l'arc } (p, i) \text{ appartient à un de ces chemins} \\ t_{pred}[v] = v & \text{sinon} \end{array} \right.$$

Prenons le graphe suivant :



Par exemple la liste suivante :

$$t_p = [1, 2, 3, 3, 3, 4, 5, 7]$$

on a :

i	0	1	2	3	4	5	6	7
$t_p[i]$	1	2	3	3	3	4	5	7

que l'on lit ainsi :

- 0 a pour prédécesseur 1 qui a pour prédécesseur 2 qui a pour prédécesseur 3 $\Longrightarrow$ c'est un chemin partant de 3 : $3 \to 2 \to 1 \to 0$
- 4 a pour prédécesseur 3 ; 5 a pour prédécesseur 4 ; 6 a pour prédécesseur 5 $\Longrightarrow$ c'est encore un chemin partant de 3 : $3 \to 4 \to 5 \to 6$

2.2 L'algorithme

L'algorithme de Dijkstra s'appuient sur deux constats :

Propriété 2-1:

- si $i_0 \to i_1 \to \ldots i_k$ est un plus court chemin pour aller de $P = i_0$ à $Q = i_k$ alors le "sous-chemin" $i_0 \to i_1 \to \ldots i_j$ avec $1 \leq j \leq k$ est un plus court chemin de i_0 à i_j
- on peut représenter des plus courts chemins de P aux autres sommets de $G = (V, E)$ par une liste t_p dans laquelle $t_p[i]$ est l'indice du prédécesseur de i dans le plus court chemin (voir ci-dessus)

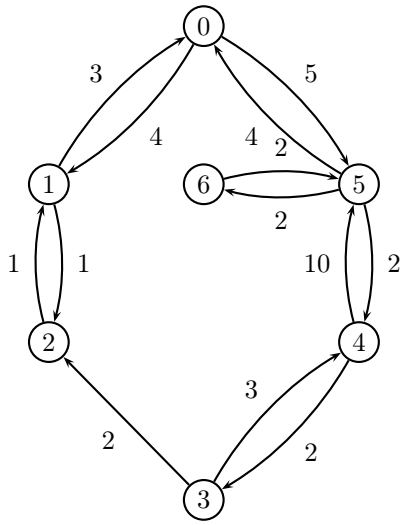
ALGORITHME :

- OBJECTIF : on donne la matrice d'adjacence *matadj* du graphe $G = (V, E)$ orienté et pondéré et un sommet **de départ** v et l'algorithme retourne pour chaque sommet i un plus court chemin de v à i.

- Les chemins seront représentés par deux tableaux t_p et t_d avec $t_p[i]$ représentant le prédécesseur de i dans le plus court chemin de v à i et $t_d[i]$ **est le poids total de ce chemin** $v \longrightarrow i$. **On notera** $arc(x)$ **les sommets extrémités d'un arc d'origine** x **(sommets adjacents à** x**).**
- On se donne pour tout le processus deux ensembles A et C tels que $A \cup C = V$ et $A \cap C = \emptyset$ avec a le sommet de C que l'on s'apprête à transférer vers A, donc $A \equiv$ **sommets visités et** $C \equiv$ **sommets non visités**
- On initialise $A = \emptyset$ et $C = V$ (tous les sommets)
- On inscrit **initialement** dans t_d les seuls chemins directs de v à x (x est prédécesseur) avec le poids "∞" (supérieur au poids maxi dans le graphe), sauf pour v lui-même qui prend le poids 0 (distance nulle)

- FONCTIONNEMENT:

  `Tant que` C `non vide:`
  - `on trie` C `dans le sens croissant des poids` $t_d(c)$
  - `on choisit pour` a `le premier sommet de C trié soit` $a = C[0]$ **puisque c'est le sommet le plus proche de v** `(principe de l'algorithme de Dijkstra)`
  - `Pour tout sommet` $c \in (C \cap arc(a))$ `(sommets adjacents à` a `non encore transférés dans` A`) si la distance` $v \to a \to c$ `est inférieure au poids déjà inscrit dans` $td(c)$`, alors on remplace cette dernière par` $td[a] + w_{a,c}$ `et on inscrit` a `comme prédécesseur de` c `dans` t_p.
  - `On extrait` a `de` C `que l'on ajoute à` A.



Partons du noeud $v = 3$ dans le graphe ci-contre :
$A = []$ $a =?$ $C = [0, 1, 2, 3, 4, 5, 6]$ $t_p = [3, 3, 3, 3, 3, 3, 3]$ $t_d = [\infty, \infty, \infty, 0, \infty, \infty, \infty]$
$A = []$ $a =?$ $C = [3, 0, 1, 2, 4, 5, 6]$ $t_p = [3, 3, 3, 3, 3, 3, 3]$ $t_d = [\infty, \infty, \infty, 0, \infty, \infty, \infty]$
$A = []$ $a = 3$:

- $3 \to 2$: $matadj_{32} = w_{32} = 2$ donc $td[2] = t_d[3] + w_{32} = 0 + 2 = 2$ et $t_p[2] = 3$
- $3 \to 4$: $matadj_{34} = w_{34} = 3$ donc $td[4] = t_d[3] + w_{34} = 0 + 3 = 3$ et $t_p[4] = 3$

$A = [3]$ $a =?$ $C = [0, 1, 2, 4, 5, 6]$ $t_p = [3, 3, 3, 3, 3, 3, 3]$ $t_d = [\infty, \infty, 2, 0, 3, \infty, \infty]$
$A = [3]$ $a =?$ $C = [2, 4, 0, 1, 5, 6]$ $t_p = [3, 3, 3, 3, 3, 3, 3]$ $t_d = [\infty, \infty, 2, 0, 3, \infty, \infty]$
$A = [3]$ $a = 2$:

- $2 \to 1$: $matadj_{21} = w_{21} = 1$ donc $td[1] = t_d[2] + w_{21} = 2 + 1 = 3$ et $t_p[1] = 2$

$A = [3, 2]$ $a =?$ $C = [4, 0, 1, 5, 6]$ $t_p = [3, 2, 3, 3, 3, 3, 3]$ $t_d = [\infty, 3, 2, 0, 3, \infty, \infty]$
$A = [3, 2]$ $a =?$ $C = [4, 1, 0, 5, 6]$ $t_p = [3, 2, 3, 3, 3, 3, 3]$ $t_d = [\infty, 3, 2, 0, 3, \infty, \infty]$
$A = [3, 2]$ $a = 4$:

- $4 \to 5$: $matadj_{45} = w_{45} = 10$ donc $td[5] = t_d[4] + w_{45} = 3 + 10 = 13$ et $t_p[5] = 4$

$A = [3, 2, 4]$ $a =?$ $C = [1, 0, 5, 6]$ $t_p = [3, 2, 3, 3, 3, 4, 3]$ $t_d = [\infty, 3, 2, 0, 3, 13, \infty]$
$A = [3, 2, 4]$ $a =?$ $C = [1, 5, 0, 6]$ $t_p = [3, 2, 3, 3, 3, 4, 3]$ $t_d = [\infty, 3, 2, 0, 3, 13, \infty]$
$A = [3, 2, 4]$ $a = 1$:

- $1 \to 0$: $matadj_{10} = w_{10} = 3$ donc $td[0] = t_d[1] + w_{10} = 3 + 3 = 6$ et $t_p[0] = 1$

$A = [3, 2, 4, 1]$ $a =?$ $C = [5, 0, 6]$ $t_p = [1, 2, 3, 3, 3, 4, 3]$ $t_d = [6, 3, 2, 0, 3, 13, \infty]$
$A = [3, 2, 4, 1]$ $a =?$ $C = [0, 5, 6]$ $t_p = [1, 2, 3, 3, 3, 4, 3]$ $t_d = [6, 3, 2, 0, 3, 13, \infty]$
$A = [3, 2, 4, 1]$ $a = 0$:

- $0 \to 5$: $matadj_{05} = w_{05} = 5$ et $t_d[0] + w_{05} = 6 + 5 = 11 < td[5] = 13$ **donc** $td[5] = 11$ **et** $t_p[5] = 0$ (**Attention :** modification du chemin pour aller à 5 !)

$A = [3, 2, 4, 1, 0]$ $a =?$ $C = [5, 6]$ $t_p = [1, 2, 3, 3, 3, 0, 3]$ $t_d = [6, 3, 2, 0, 3, 11, \infty]$
$A = [3, 2, 4, 1, 0]$ $a =?$ $C = [5, 6]$ $t_p = [1, 2, 3, 3, 3, 0, 3]$ $t_d = [6, 3, 2, 0, 3, 11, \infty]$
$A = [3, 2, 4, 1, 0]$ $a = 5$:

- $5 \to 6$: $matadj_{56} = w_{56} = 2$ et $t_d[6] = t_d[5] + w_{56} = 11 + 2 = 13$ et $t_p[6] = 5$

$A = [3, 2, 4, 1, 0, 5]$ $a =?$ $C = [6]$ $t_p = [1, 2, 3, 3, 3, 0, 5]$ $t_d = [6, 3, 2, 0, 3, 11, 13]$
$A = [3, 2, 4, 1, 0, 5]$ $a = 6$:

Aucune modification supplémentaire n'est ensuite effectuée sur les listes, sinon l'ajout du dernier sommet 6 puisqu'il ne reste plus que lui ; ainsi, en fin de traitement, on obtient les résultats suivants :

$$A = [3, 2, 4, 1, 0, 5, 6] \quad C = [] \quad t_p = [1, 2, 3, 3, 3, 0, 5] \quad t_d = [6, 3, 2, 0, 3, 11, 13]$$

$i =$	0	1	2	3	4	5	6
$tp[i] =$	1	2	3	3	3	0	5
$td[i] =$	6	3	2	0	3	11	13

Par exemple, le plus court chemin pour aller de 3 à 6 se lit ainsi :
$\Rightarrow$ 6 a pour prédécesseur 5, qui a pour prédécesseur 0, qui a pour prédécesseur 1, qui a pour prédécesseur 2, qui a pour prédécesseur 3. **Un plus court chemin pour aller sur le sommet 6 en partant de 3 est donc :**

$$\boxed{3 \to 2 \to 1 \to 0 \to 5 \to 6}$$

2.3 Code python

Listing V.3 –

```
def Dijkstra(matadj,v):
    N,infini=max(matadj.shape),matadj[0,0]
    #### Construction de A,C,tp, td ####
    A,C=[],list(range(0,N))
    td=np.ones(N,dtype=int)*infini
    td[v]=0
    tp=np.ones(N,dtype=int)*v

    while C!=[]:
        C.sort(key=lambda i:td[i]) #trie C en plaçant en tête son élément dont le poids est le plus faible dans td
        a=C[0]
        for c in C:
            if td[a]+matadj[a,c]<td[c]:
                td[c]=td[a]+matadj[a,c]
                tp[c]=a
        A.append(a)
        C.remove(a)
    return v,tp,td
```

En lançant `print Dijkstra(matadj,3)`, avec `matadj` matrice d'adjacence du graphe ci-dessus, on obtient la sortie suivante, conforme aux résultats partiels obtenus plus haut :

```
(3, array([1, 2, 3, 3, 3, 0, 5]), array([ 6, 3, 2, 0, 3, 11, 13]))
```