IF4b – Graphes : Parcours

Les graphes servent dans de nombreux domaines à modéliser des situations. Des algorithmes performants ont été développés pour répondre à différentes questions.

Objectifs :
- Acquérir des notions sur les principaux algorithmes utilisés avec les graphes
- Appliquer une méthode de recherche du plus court chemin entre deux sommets.

I – Parcours d'un graphe

D'une manière générale, le parcours d'un graphe consiste, en partant d'un sommet s donné, à explorer les arêtes du graphe pour découvrir le reste du graphe. Au cours de cette recherche, on accumule un certain nombre d'informations qui serviront à des applications ultérieures. A noter que de nombreux algorithmes sont directement des adaptations d'un algorithme de parcours de graphe.

Le parcours d'un graphe explore tous les arcs issus d'un sous-ensemble de sommets initialement réduit au singleton s. Par conséquent, à moins de relancer un parcours, on ne visite que les sommets v accessibles à partir de s, c'est-à-dire pour lesquels il existe un chemin de s à v dans le graphe. Au cours de la recherche, chaque sommet accessible passe par trois états :

- Blanc s'il n'a pas encore été « découvert » par l'algorithme ;
- Gris s'il a été découvert, mais que son traitement n'est pas encore terminé ;
- Noir s'il n'y aura plus d'informations à calculer à partir de ce sommet.

On utilise principalement deux parcours de graphes : le parcours en largeur, et le parcours en profondeur.

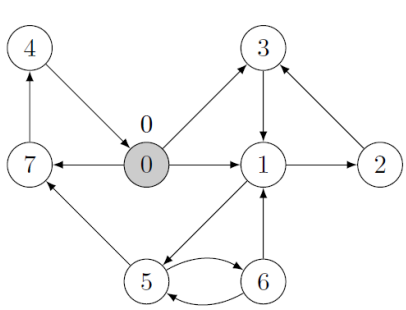
II - Parcours en largeur

II-1) Principe

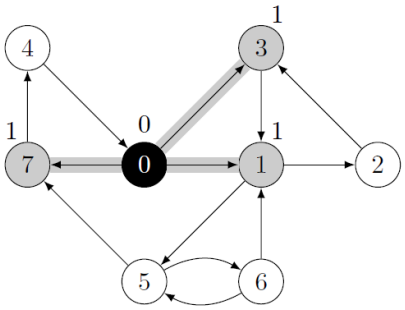
A partir d'un sommet, on explore tous ses voisins immédiats. Puis à partir d'un voisin, on explore tous ses voisins immédiats sauf ceux déjà explorés. Et ainsi de suite.

L'algorithme utilise une structure de donné appelée file fifo `f/`. Le principe d'une file fifo est celui d'une liste d'attente à un guichet : on peut ajouter et supprimer les éléments à la file, mais chaque suppression enlève uniquement l'élément le plus ancien parmi les éléments présents dans la file. Initialement, on ajoute le sommet s origine du parcours à la file. A chaque étape de l'algorithme, on défile un sommet u, puis on ajoute à la file tous les éléments blancs de sa liste d'adjacence. A chaque ajout, le sommet a blanc est colorié en gris. Une fois que toute la liste d'adjacence est ajoutée, le sommet défilé u est colorié en noir.

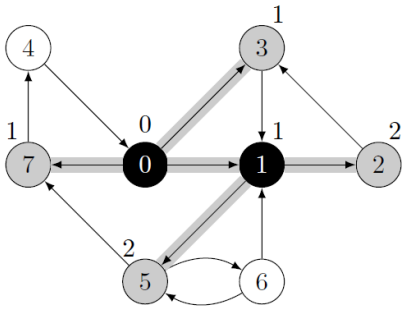
En parallèle à chaque ajout, l'algorithme attribue à « a » la valeur de u dans un tableau p (tableau des pères), ainsi qu'une valeur $d(a)$ égale à $d(u)+1$ dans un tableau d (tableau des distances à s). Voici ce que l'on obtient sur l'exemple en partant du sommet 0 : les valeurs de d sont représentées à côté des nœuds au fur et à mesure qu'ils sont calculés, les arcs $(a, p(a))$ sont surlignés en gris au fur et à mesure. Le contenu de la file est représenté sous le graphe.



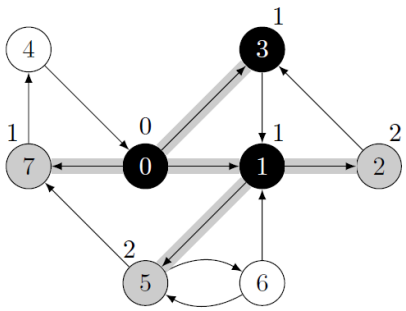
f	0



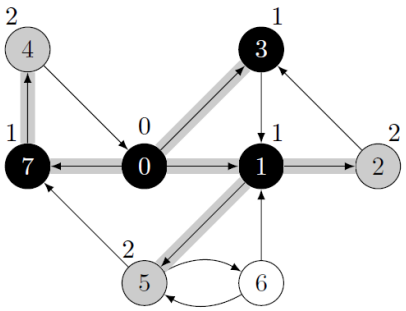
f	1	3	7



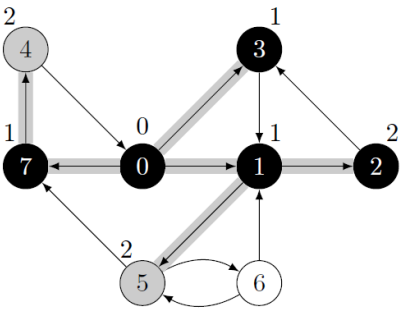
f	3	7	2	5



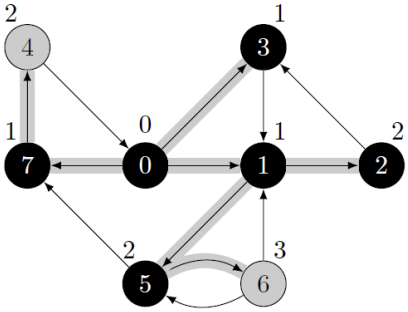
f	7	2	5



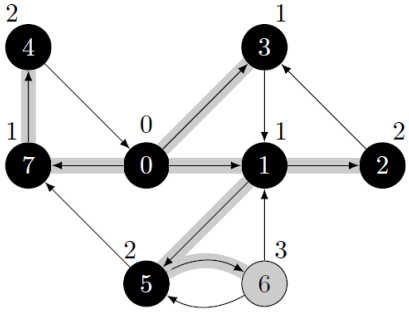
f	2	5	4



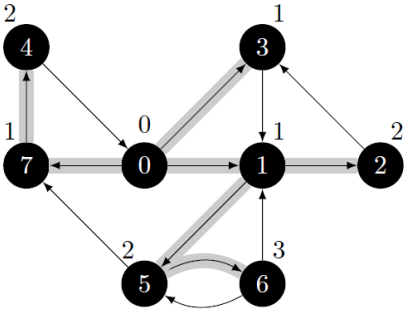
f	5	4



f	4	6



f	6



f	–

On notera que le résultat de l'algorithme dépend de l'ordre dans lequel les sommets apparaissent dans chaque liste d'adjacence. Les valeurs du tableau p (tableau des pères) peuvent changer, mais les valeurs de d (distance au sommet s de départ : 0) sont indépendantes de l'ordre.

L'implémentation de l'algorithme nécessite l'utilisation de la structure de file fifo. Python possède un module appelé « collection » qui comporte un sous-module dequeue (pour « double ended queue », qui correspond à listes doublement chainées en français) qui est une implémentation efficace de cette structure de donnée. Outre les fonctions append et pop qui fonctionnent de manière similaire aux listes, elle comporte les fonctions popleft et appendleft qui réalisent l'ajout et la suppression à gauche de la liste. La fonction len est également disponible et s'utilise comme pour une liste. Toutes les opérations sont de coût constant. L'écriture de la fonction parcours_en_largeur peut s'écrire de la manière suivante. Le code renvoie les deux tableaux d et p.

II-2) Le programme

Entrée [35]:
```
from collections import deque

def parcours_en_largeur(G,s):
    n=len(G)
    #print("Liste des sommets",[i for i in range(n)])
    file = deque()
    c = ["b"]*n #On initialise les sommets à blanc
    p = [-1]*n #La liste des pères, le sommet initial n'a pas de père qu'on représente par '-1'
    d = [-1]*n #La liste des distances par rapport au sommet initial
    c[s]="g" #On grise le sommet de départ.
    d[s]=0 #La distance à lui-même est nulle.
    file.append(s) #On ajoute notre sommet initial à notre file
    #z=file.copy() #On copie la file pour à la fin retrouver l'ordre de passage des sommets.
    while len(file)>0:
        u=file.popleft() #On enlève de la file le sommet étudié
        ### et on récupère sa valeur par la gauche.
        for x in G[u]: #On va chercher les sommets fils du sommet u sélectionné.
            if c[x]=="b":
                p[x]=u #On rentre le sommet étudié dans la liste des pères
                c[x]="g" #On passe à gris le fils en contact du sommet étudié
                d[x]=d[u]+1 #Tout les sommets sont à une distance "1" du sommet u rencontré.
                file.append(x) #On ajoute à la file le fils étudié.
        c[u]="n" #une fois tous les proches voisins testés on transforme le sommet en noir.
    #    if u not in z: z.append(u)
    #print("Ordre des sommets rencontrés",z)
    return (d,p)

G1 = [[1,3,7],[2,5],[3],[1],[0],[6,7],[1,5],[4]]
parcours_en_largeur(G1,0)
```

Out[35]:
```
([0, 1, 2, 1, 2, 2, 3, 1], [-1, 0, 1, 0, 7, 1, 5, 0])
```

II-3) Plus court chemin

Le tableau d renvoyé est le tableau des plus courtes distances. Pour tout entier x, la case d[x] contient la longueur minimale d'un chemin de s vers x. Le tableau p est le tableau des pères. Il indique, pour tout entier x de quel sommet il faut arriver pour réaliser un plus court chemin de s vers x. Le sommet s n'a pas de père (la valeur p[s] vaut (-1)), tout comme n'importe quel sommet non accessible depuis s. Pour reconstituer un plus court chemin entre s et x, s'il existe, il suffit donc de remonter de père en père, ce qui peut se faire de la manière suivante.

Entrée [2]:
```
def pcc(G,s1,s2):
    (d,p)=parcours_en_largeur(G,s1)
    if p[s2]==(-1):
        return []
    L=[s2]
    while L[-1]!=s1: #Cette commande permet de tester la dernière occurence de la liste.
        L.append(p[L[-1]]) #On ajoute le père de l'occurence testée dans L.
    L.reverse() #On renverse, c'est plus joli :)
    return(L)

pcc(G1,0,6)
```

Out[2]:
```
[0, 1, 5, 6]
```

III – Parcours en profondeur

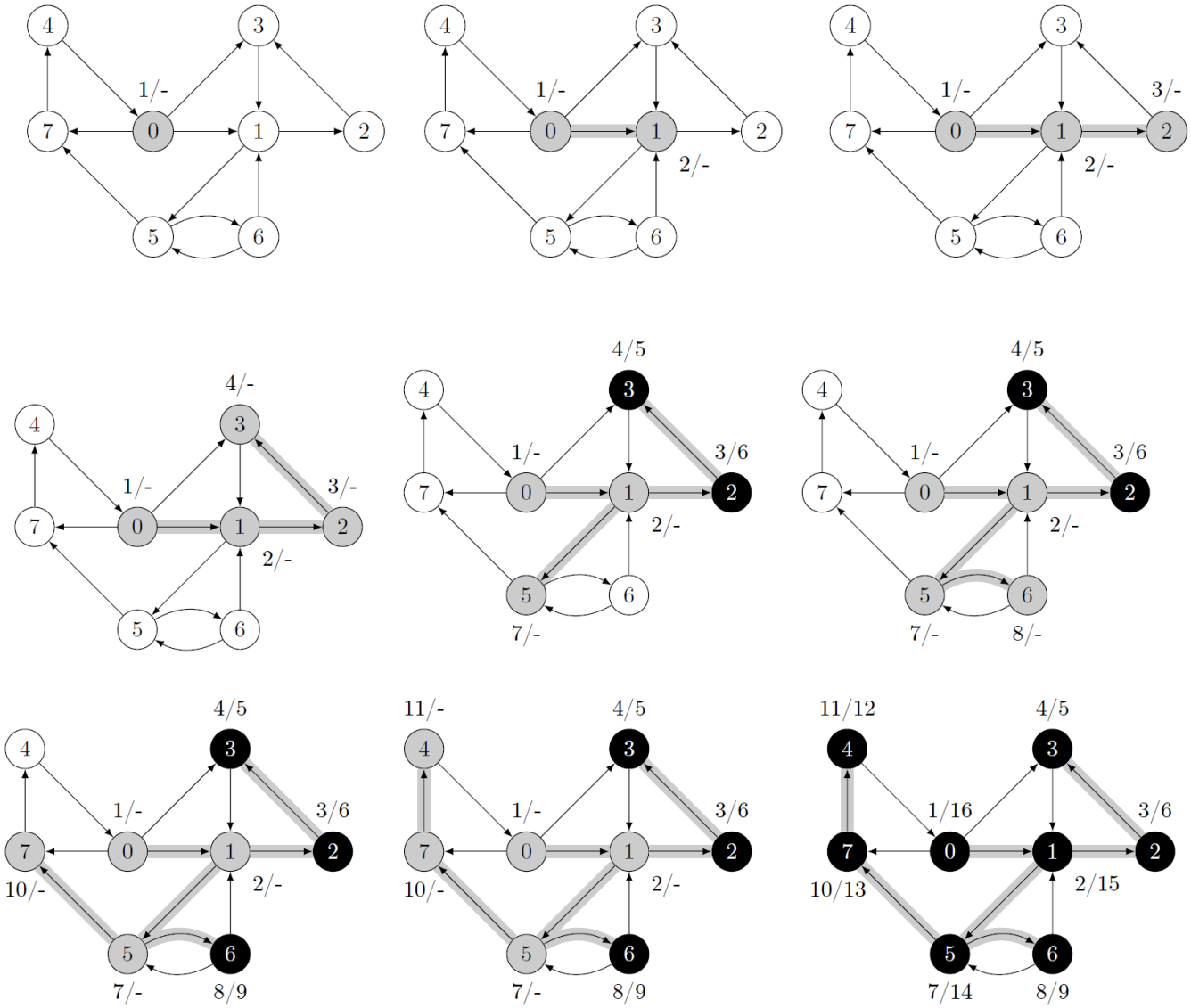
III-1) Principe

A partir d'un sommet, on passe à un de ses voisins, puis à un voisin de ce voisin et ainsi de suite. S'il n'y a pas de voisin, on revient au sommet précédent et on passe à un autre de ses voisins.

Le parcours en profondeur d'un graphe diffère essentiellement du parcours en largeur par l'ordre dans lequel sont traités les éléments accessibles à partir d'un sommet. Considérons un élément s, dont la liste d'adjacence contient les éléments a_1,..., a_k. Le parcours en profondeur va chercher à descendre systématiquement le plus loin dans le graphe tant que c'est possible. En d'autres termes, le sommet a_2 ne sera traité que lorsque tous les sommets accessibles depuis a_1 ont été traités (à moins qu'il n'apparaisse une nouvelle fois dans les descendants de a_1). Lorsque l'algorithme tombe sur un sommet dont tous les éléments de la liste d'adjacence ont été découverts, il remonte au sommet précédent et recommence la descente à partir du deuxième sommet blanc disponible, et s'arrête lorsque tous les sommets accessibles ont été traités.
A nouveau, les sommets accessibles passent par les statuts blancs (non découvert), gris (découvert) et noir (traité) tout au long de l'exécution. Un compteur date est incrémenté à chaque étape de l'algorithme. On emmagasine également les informations suivantes pour chaque sommet a accessible :

- d(a) : date à laquelle le sommet est découvert
- p(a) : père de a pour le parcours issu de s, c'est-à-dire l'unique sommet x tel que a est apparu pour la première fois dans l'arête (x,a).
- f(a) : date de fin de traitement de a, c'est-à-dire celle à laquelle tous les descendants de a ont été découverts.

L'exécution sur l'exemple précédent en partant du sommet 0 donne le déroulement suivant (les dates de début et de fin de traitement sont indiquées à proximité des nœuds). On remarquera notamment que les arcs surlignés illustrant ne sont plus du tout les mêmes que ceux du parcours en largeur.



L'écriture du code se fait naturellement à l'aide d'une sous fonction récursive qui permet de « traiter » un sommet blanc : cela consiste à le colorier en gris, lui attribuer sa date de début de traitement, appeler récursivement la fonction à tous les éléments de sa liste d'adjacence et enfin, une fois ce traitement achevé, à le colorier en noir puis lui attribuer sa date de fin de traitement.

III-2) Programme

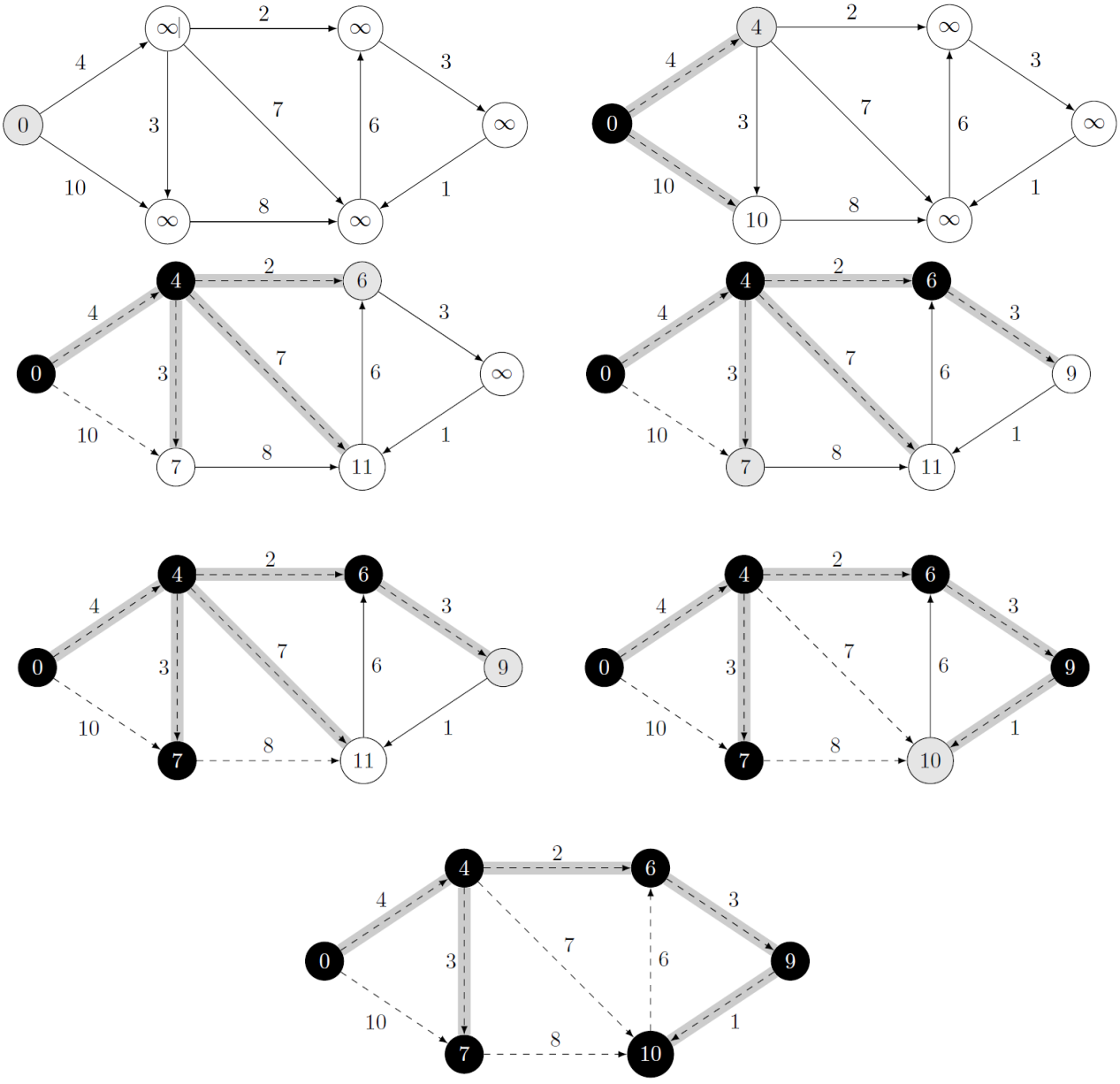
Entrée [22]:
```
def parcours_en_profondeur(G,s):
    n=len(G);
    date=1;
    c = ["b"]*n
    p = [-1]*n #Liste des pères
    d = [-1]*n #Liste des dates
    f = [-1]*n #Liste des fins de traitement
    def visiter_sommet(u):
        nonlocal date #On ne veut pas que date soit une variable locale
        c[u]="g" #on grise notre sommet
        d[u]=date
        date=date+1 #on incrémente pour le prochain sommet
        for x in G[u]:
            if c[x]=="b": #On teste si blanc, si oui on l'ajoute à notre test.
                p[x]=u
                visiter_sommet(x)
        f[u]=date
        date=date+1
        c[u]="n" #On passe en noir notre sommet car on a testé toute sa liste d'adjacence
    visiter_sommet(s)
    return(d,p,f)

G1 = [[1,3,7],[2,5],[3],[1],[0],[6,7],[1,5],[4]]
parcours_en_profondeur(G1,0)
```

Out[22]:
```
([1, 2, 3, 4, 11, 7, 8, 10],
 [-1, 0, 1, 2, 7, 1, 5, 5],
 [16, 15, 6, 5, 12, 14, 9, 13])
```

IV – Algorithme de Djikstra

L'algorithme de Dijkstra fonctionne de la manière suivante : il maintient un ensemble de sommets E dont les distances minimales à s ne sont pas encore calculées. Initialement, cet ensemble est égal à l'ensemble de tous les sommets. A chaque étape, l'algorithme extrait de E le sommet u dont la distance courante à s est minimale, et relâche tous les arcs issus de u. Il s'arrête lorsque E est vide.

Voici un exemple d'exécution de l'algorithme. Le sommet de départ est celui de gauche et les valeurs courantes des distances minimales sont directement représentées à l'intérieur des nœuds, qui n'ont pas de nom pour l'occasion. Les sommets sont coloriés en noirs au fur et à mesure qu'ils sont extraits de E, le sommet réalisant le minimum des distances parmi les éléments de E est signalé en gris. Les arcs relâchés sont hachurés. Les relations père/fils sont, comme pour les parcours, les arcs surlignés en gris.



- Bibliographie : Cours d'informatique - M.Puyhaubert