05 - Les graphes

1 Définitions générales

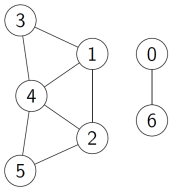
1.1 Graphes non orientés

Définition 1 - *Graphe non orienté, sommet, arête*
Un graphe non orienté $G = (S, A)$ est un couple formé de deux ensembles :

- un ensemble $S = \{s_0, \dots, s_{n-1}\}$ dont les éléments sont appelés **sommets** ou **noeuds** du graphe.
- un ensemble A formé de parties à deux éléments de S ; les éléments de A sont appelés les **arêtes** ou **arcs** du graphe.

***Remarque* 1 :** Une arête peut relier un sommet à lui-même : de telles arêtes sont appelées **boucles**.
Dans tout ce cours, on travaillera avec des graphes sans boucles, on parle de graphes simples.

Exemple 1 : Décrire les ensembles S et A pour le graphe suivant :



Définition 2 - *Degré*
Soit $G = (S, A)$ un graphe non orienté, et s un sommet de ce graphe. On appelle **degré de s** le nombre d'arêtes qui le relient aux autres sommets.

Exemple 2 : Donner le degré de chaque sommet du graphe précédent.

Définition 3 - *Chemin, cycle*
Soit $G = (S, A)$ un graphe non orienté, et s et t deux sommets de ce graphe.

- Un **chemin** de longueur k qui relie s à t est une séquence de sommets $< c_0, \dots, c_k >$ telle que $c_0 = s$, $c_k = t$ et pour tout i de $[\![1; k]\!]$, $\{c_{i-1}, c_i\}$ est une arête. Ce chemin comporte donc k arêtes.
- Un chemin est dit **élémentaire** si les sommets qu'il contient sont tous distincts, à part éventuellement le premier et le dernier qui peuvent être les mêmes.
- Un **cycle** est un chemin dont les sommets de départ et d'arrivée sont identiques.

Exemple 3 : On reprend le graphe de l'exemple 1. Donner deux chemins de différentes longueurs, qui relient le sommet 2 au sommet 3. Et pour relier les sommets 0 et 1 ?

Définition 4 - *Connexité*
Un graphe est **connexe** si pour tout couple de sommets, il existe un chemin qui les relie.

Exemple 4 : Le graphe de l'exemple 1 n'est pas connexe, car ....

1.2 A quoi ça sert ?

A plein de choses ! Citons en vrac :

- les réseaux routiers ou ferroviaires (application type Waze ou Google Maps)
- le Web (sommets=pages internet, arête=lien entre deux pages)
- les réseaux sociaux (sommets=les individus, arête=connexion entre deux individus)

Une fois que l'on a modélisé la situation, on peut alors, grâce à la théorie des graphes, répondre à de nombreux problèmes : rechercher des plus courts chemins, établir une connexion entre des individus, etc...

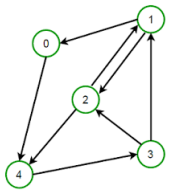
1.3 Graphes orientés

Définition 5 - *Graphe orienté, sommet, arête*
Un graphe orienté $G = (S, A)$ est un couple formé de deux ensembles :

- un ensemble $S = \{s_0, \dots, s_{n-1}\}$ dont les éléments sont appelés **sommets** ou **noeuds** du graphe.
- un ensemble A formé de **couples** d'éléments de S ; les éléments de A sont appelés les **arêtes** ou **arcs** du graphe.

Remarque* 2 *: Les graphes orientés apparaissent naturellement sur nos exemples précédents : dans un réseau social Alice peut suivre Bob sans que Bob suive Alice ; dans les pages web une page peut contenir un lien vers une autre page sans que ce soit réciproque, etc.

Exemple 5 : Décrire les ensembles S et A pour le graphe suivant :



Définition 6 - *Degré sortant et degré entrant*
Soit $G = (S, A)$ un graphe orienté, et s un sommet de ce graphe.

- On appelle **degré sortant de s**, noté le nombre d'arêtes dont l'extrémité initiale est s. On le note $d_+(s)$.
- On appelle **degré entrant de s**, noté le nombre d'arêtes dont l'extrémité finale est s. On le note $d_-(s)$.

Exemple 6 : Pour le graphe précédent, $d_+(0) = \dots$, $d_-(0) = \dots$, $d_+(4) = \dots$, $d_-(4) = \dots$

1.4 Pondération d'un graphe

Dans le cas par exemple d'une carte routière, les arêtes sont les routes reliant deux nœuds routiers. Il peut être intéressant de faire figurer la longueur de cette route. Nous utiliserons pour ça des étiquettes apposées sur les arêtes.

Définition 7 - *Graphe pondéré*
Un graphe pondéré est un triplet $G = (S, A, w)$, où (S, A) est un graphe, et où $w : A \to \mathbb{R}$ est une fonction qui à chaque arête associe une étiquette (ou pondération).

Exemple 7 (*CCINP 2025 PSI*) : .

Un randonneur souhaite effectuer un long Trek, c'est-à-dire une série de randonnées sur plusieurs jours. Il organise son parcours à partir d'un site qui propose différentes randonnées d'une journée. Chaque randonnée est définie par un niveau de difficulté allant de 1 (facile) à 5 (extrême). L'ensemble des données permet au randonneur d'établir un graphe où :

- les sommets représentent les points de départ / arrivée des randonnées,
- les arêtes représentent les randonnées possibles avec comme poids le niveau de la randonnée.

On suppose qu'il y a une unique randonnée qui relie 2 sommets du graphe.
On suppose que les randonnées peuvent être effectuées du point de départ vers le point d'arrivée ou du point d'arrivée vers le point de départ sans que la difficulté ne soit modifiée.
Le graphe représentant les différentes randonnées sera donc non orienté.

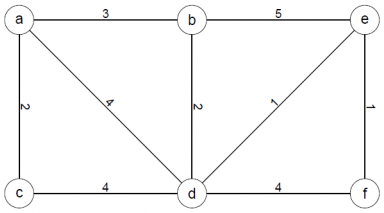


Figure 3 - Graphe G

2 Modélisation des graphes

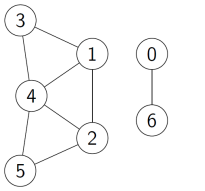
Nous allons décrire différentes manières d'implémenter les graphes, de manière à pouvoir les étudier efficacement.

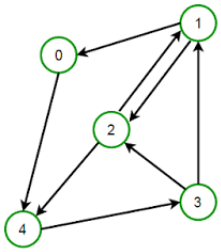
2.1 Matrice d'adjacence

On suppose que les sommets du graphe sont numérotés de 0 à $n-1$.

Définition 8
La matrice d'adjacence associée à un graphe non pondéré est une matrice à n lignes et n colonnes, remplie de 0 et de 1. Le coefficient d'indice (i, j) vaut 1 si et seulement si il existe une arête du sommet i au sommet j.

Exemple 8 : Donner les matrices d'adjacence pour les deux graphes suivants :





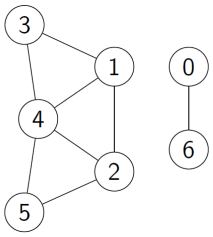
***Remarque* 3 :**
- Si le graphe est non orienté, la matrice est symétrique.
- Pour représenter la matrice d'adjacence d'un graphe sous Python, on utilisera un tableau `numpy` déclaré ainsi : `M=np.array([[0,1],[1,0]])`.
  (On suppose que le module `numpy` a été chargé par l'instruction `import numpy as np`)
- Cette définition peut facilement se prolonger au cas d'un graphe pondéré, dans ce cas le coefficient (i, j) sera égal au poids de l'arête du sommet i au sommet j.

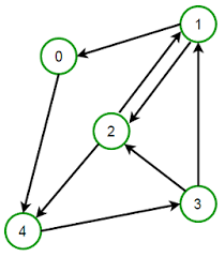
2.2 Liste d'adjacence

On suppose que les sommets du graphe sont numérotés de 0 à $n-1$.

Définition 9 - *Liste d'adjacence d'un sommet, d'un graphe*
• La liste d'adjacence d'un sommet est la liste de ses voisins (pour un graphe non orienté) ou de ses successeurs (pour un graphe orienté).
• La liste d'adjacence d'un graphe est la liste des listes d'adjacences de ses sommets (la liste d'adjacence du sommet i se trouvant à l'indice i de la liste finale).

Exemple 9 : Construire la liste d'adjacence des deux graphes suivants :





***Remarque* 4 :**
- Pour représenter la liste d'adjacence d'un graphe sous Python, on utilisera simplement une liste de listes.
- Pour implémenter un graphe pondéré, on pourra par exemple stocker dans la liste d'adjacences de chaque sommet une liste de couple (successeur,poids).

2.3 Avantages et inconvénients d'une matrice d'adjacence VS une liste d'adjacence

- Pour des graphes peu denses (i.e. peu d'arêtes) , les listes d'adjacences seront plus courtes, et la mémoire utilisée plus faible que lors d'utilisation des matrices d'adjacences.
- La matrice d'adjacence permet d'avoir un accès rapide à l'existence d'une arête entre deux sommets (en effet l'accès aux éléments d'une matrice s'effectue en temps constant).
- Avec une liste, pour déterminer si un sommet i a des voisins (ou un successeur), ou pour trouver le degré d'un sommet, on peut utiliser l'opérateur `len` dont la complexité est en temps constant (alors qu'il faudra parcourir toute la ligne i avec la matrice).
- Certaines propriétés du graphe ne peuvent s'obtenir qu'avec la matrice M d'adjacence, comme par exemple *"le coefficient (i,j) de la matrice M^p est le nombre de chemins de longueur p qui relient le sommet i au sommet j"* (pas au programme)

2.4 Dictionnaires

Si les sommets du graphe ne sont pas numérotés de 0 à n (par exemple ce sont des noms de villes), on peut alors utiliser un dictionnaire pour modéliser le graphe : chaque clé sera un sommet du graphe, et la valeur associée sera la liste d'adjacence du sommet, ou bien à nouveau un dictionnaire permettant d'avoir les successeurs et le poids de l'arête correspondante (cas d'un graphe pondéré).

Exemple 10 (*CCINP 2025 PSI*) : .

Le graphe G de la **figure 3** est représenté par le dictionnaire `G` défini de la manière suivante :

```
G = dict()
G['a'] = {'b':3, 'c':2, 'd':4}
G['b'] = {'a':3, 'd':2, 'e':5}
G['c'] = {'a':2, 'd':4}
G['d'] = {'a':4, 'b':2, 'c':4, 'e':1, 'f':4}
G['e'] = {'b':5, 'd':1, 'f':1}
G['f'] = {'d':4, 'e':1}
```

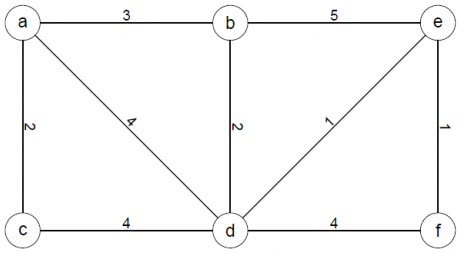


Figure 3 - Graphe G

2.5 Premiers exercices d'implémentation de graphes en Python

Exercice 1 : 1. Ecrire une fonction `MatriceToListe`, ayant pour paramètre d'entrée la matrice d'adjacence d'un graphe (sous forme d'un tableau `numpy`), et qui retourne la liste d'adjacence de ce graphe.
2. Ecrire une fonction `ListeToMatrice`, ayant pour paramètre d'entrée la liste d'adjacence d'un graphe, et qui retourne la matrice d'adjacence de ce graphe.

Exercice 2 : Pour chacun des problèmes ci-dessous, vous écrirez deux fonctions : une qui prend en entrée la matrice d'adjacence d'un graphe, et l'autre qui prend en entrée la liste d'adjacence d'un graphe. Il est interdit ici d'utiliser les fonctions créées dans l'exercice précédent.

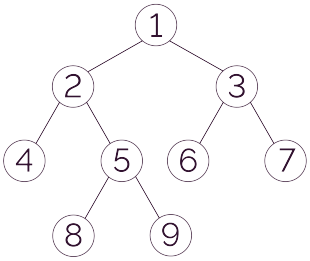
1. Déterminer le degré d'un sommet donné pour un graphe non orienté.
2. Déterminer si deux sommets donnés sont voisins, i.e. reliés par une arête. (dans le cas d'un graphe orienté, étant donnés deux sommets s et t, il s'agit de savoir s'il existe une arête partant de s et arrivant à t)
3. Pour chacune de ces quatre fonctions, déterminer leur complexité en fonction du nombre n de sommets du graphe.

3 Parcours de graphes

Parcourir efficacement les sommets d'un graphe en utilisant les arêtes qui les relient est essentiel pour par exemple :

- savoir s'il existe un chemin entre deux sommets donnés,
- déterminer le plus court chemin entre deux sommets donnés,
- savoir si un graphe non orienté est connexe,
- etc.

Nous allons tout d'abord expliquer les deux principes de parcours avec un graphe particulier : un arbre. C'est un graphe sans cycle.
Voici ci-contre un exemple d'arbre :



Il y a essentiellement deux manières de parcourir cet arbre en partant du sommet 1 :

- **en profondeur** : on parcourt toute une branche avant de passer à la suivante. Ici, en choisissant les branches de gauche en premier, on obtient la succession de sommets visités : .....
- **en largeur** : on parcourt l'arbre génération de voisins par générations de voisins. Ici on obtient la succession de sommets visités : .....

Nous allons voir comment ces deux types de parcours se généralisent à un graphe quelconque. Nous garderons le vocabulaire de parcours en profondeur/largeur.

Remarque* 5 *: Dans les arbres il n'y a pas de cycles, on ne peut donc pas rencontrer deux fois le même sommet dans les parcours expliqués ci-dessus. Dans les graphes, ce n'est pas le cas. Il faudra donc veiller à ce que les sommets ne soient pas visités plusieurs fois (sinon notre parcours pourrait ne jamais s'arrêter).

3.1 Parcours en profondeur

3.1.1 Principe de l'algorithme

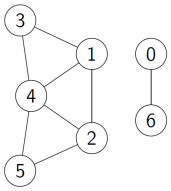
L'algorithme de parcours en profondeur d'un graphe consiste, partant d'un sommet de départ, à visiter les sommets du graphe en favorisant la génération de sommets suivants par rapport à la génération actuelle.

Il repose sur les étapes suivantes :

- créer une liste des sommets visités qui initialement est vide.
- créer une **pile** des sommets à traiter contenant initialement le sommet de départ.
- tant qu'il reste des sommets dans la pile :
  - extraire le sommet du haut de la pile.
  - si ce sommet n'est pas déjà dans la liste des sommets visités :
    - l'ajouter à la liste des sommets visités, et parcourir les sommets voisins.
    - pour chaque voisin : s'il ne fait pas partie des sommets déjà visités, l'ajouter à la pile des sommets à traiter.
- retourner la liste des sommets visités.

3.1.2 Illustration

Parcours en profondeur du graphe ci-dessous en partant du sommet 5 :



3.1.3 Implémentation en Python

> **Exercice 3 :** Écrire une fonction de paramètre une liste `L` d'adjacence d'un graphe, et un sommet `d` de départ, et qui retourne la liste des sommets visités à partir de ce sommet de départ, en utilisant le principe du parcours en profondeur.

Amélioration de ce programme : à chaque fois que l'on a besoin de savoir si le sommet que l'on rencontre a déjà été visité, le parcours de la liste des sommets déjà visités a une complexité linéaire. On peut améliorer cela en marquant les sommets que l'on a déjà visité ainsi : on crée une liste `marquage` de longueur le nombre de sommets du graphe, qui au départ ne contient que les booléens `False` ; à chaque fois que l'on visite un sommet `s`, alors on passe `marquage[s]` à `True`. Pour savoir si un sommet `c` a déjà été visité, il suffit d'accéder à `marquage[c]`, ce qui se fait en complexité constante.

> **Exercice 4 :** Améliorer la fonction précédente en suivant ce principe.

3.1.4 Application à la détection de connexité

> **Exercice 5 :** Ecrire une fonction permettant de savoir si un graphe non orienté est connexe ou non. On utilisera la fonction de parcours en profondeur, en s'interrogeant sur le résultat de cette fonction lorsque le graphe est connexe ou lorsqu'il ne l'est pas.

3.2 Parcours en largeur

3.2.1 Principe de l'algorithme

L'algorithme de parcours en largeur d'un graphe consiste, partant d'un sommet de départ, à visiter les sommets du graphe en traitant d'abord tous les voisins d'un sommet donné avant de passer à la génération suivante.

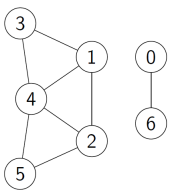
Il repose sur les étapes suivantes :

- créer une liste des sommets visités qui initialement est vide.
- créer une **file** des sommets à traiter contenant initialement le sommet de départ.
- tant qu'il reste des sommets dans la file :
  - extraire le sommet en tête de file.
  - si ce sommet n'est pas déjà dans la liste des sommets visités :
    - l'ajouter à la liste des sommets visités, et parcourir les sommets voisins.
    - pour chaque voisin : s'il ne fait pas partie des sommets déjà visités, l'ajouter à la file des sommets à traiter.
- retourner la liste des sommets visités.

Le fait d'utiliser une file (alors qu'on utilisait une pile dans le parcours en profondeur) a pour conséquence que les nouveaux voisins sont traités après les anciens. On privilégie donc les sommets d'une même génération par rapport à ceux des générations suivantes.

3.2.2 Illustration

Parcours en largeur du graphe ci-dessous en partant du sommet 5 :



3.2.3 Implémentation en Python

Exercice 6 : Écrire une fonction de paramètre une liste `L` d'adjacence d'un graphe, et un sommet `d` de départ, et qui retourne la liste des sommets visités à partir de ce sommet de départ, en utilisant le principe du parcours en largeur.

- *Pour implémenter la structure de file, vous utiliserez des objets de type* **`deque`**, *(penser à les importer avec* **`from collections import deque`**), *et les méthodes associées* **`append`** *et* **`popleft`**.
- *Pour savoir si un sommet a déjà été visité, vous utiliserez un tableau de booléens.*

4 Plus courts chemins dans un graphe

4.1 Arêtes de poids constants

Etant donné deux sommets x et y d'un graphe, on cherche le chemin le plus court (i.e. celui qui contient le moins d'arêtes) dont le point de départ est x et le point d'arrivée est y.

Pour cela il suffit d'effectuer un parcours en largeur du graphe en démarrant depuis le point x, et on va en fait calculer la distance la plus courte entre x et tous les sommets atteignables depuis x.

On démarre de x, puis on visite les voisins de x, puis on visite les voisins des voisins de x, ..., jusqu'à avoir visité tous les sommets atteignables en partant de x.

On va donc tenir à jour, dans l'algorithme du parcours en largeur à partir de x, une liste **`dist`** qui contiendra les distances entre x et chacun des sommets du graphe. Initialement cette liste sera initialisée avec des $+\infty$ (ce qui signifie que le sommet n'a pas été relié à x), sauf **`dist[x]`** qui vaut 0.

A retenir

En Python $+\infty$ se code **`float('inf')`**, ou bien **`np.inf`** si **`np`** désigne le module **`numpy`**.

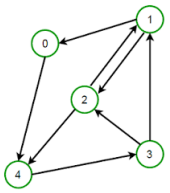
Cette liste de distance sera tenue à jour à chaque tour de la boucle **`tant que`** en suivant le code suivant , **à compléter** :

```
1  def distances(L, d):
2      ''' L est la liste d'adjacence d'un graphe, d est le sommet de départ'''
3      n = len(L)
4
5      # On initialise toutes les distances à l'infini
6      dist = [..............  for i in range(n)]
7
8      # Le sommet de départ est à une distance 0 de lui-même
9      dist[d] = 0
10
11     file = deque([d])
12
13     while len( file ) != 0:
14         s = ..............  #on prend le sommet en tête de file
15
16         # On parcourt les voisins de s
17         for x in L[s]:
18             # Si le sommet n'a pas encore été découvert
19             if dist[x] == ..............:
20                 dist[x] = dist[s] + ......
21                 file.append(x)
22
23     return dist
```

Cet algorithme fonctionne aussi sur un graphe orienté.

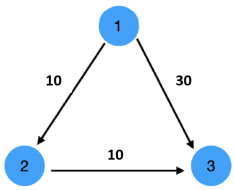
Exercice 7 : .

Tester votre programme sur le graphe suivant, en calculant les distances en partant du sommet 0.



4.2 Arêtes pondérées de poids positifs : l'algorithme de Dijkstra

Lorsque nous avons à faire à un graphe pondéré, un simple parcours en largeur ne suffit pas à trouver le plus court chemin entre deux sommets.
Sur l'exemple ci-contre, la longueur du plus court chemin entre les sommets 1 et 3 est 20, mais le parcours en largeur nous donnerait 30.



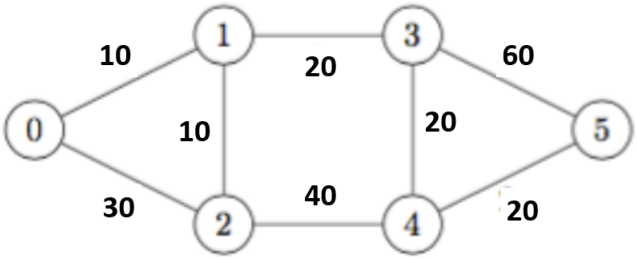
On peut avoir plusieurs idées pour trouver la distance minimum entre un somme t de départ d et un sommet d'arrivée a :

- Déterminer tous les chemins possibles de d à a et prendre celui de longueur minimum : le problème est que la complexité est dans le pire des cas **exponentielle**, voire en $O(n!)$ (si n est le nombre de sommets du graphe).
- Principe de **l'algorithme glouton** : partir de d et construire un chemin arrivant à a en prenant à chaque fois l'arête de plus petit poids. Le problème est que rien ne nous garantit que le chemin obtenu sera effectivement celui de poids minimal.

Pour faire efficace et correct, nous avons besoin de l'algorithme de Dijkstra (*Edsger Wybe Dijkstra est un mathématicien et informaticien néerlandais né en 1930 et mort en 2002*).

Tout d'abord, nous allons supposer que le graphe pondéré est implémenté de la façon suivante : nous allons utiliser une liste d'adjacence adaptée, où pour chaque sommet, la liste de ses voisins sera remplacée par un liste de couples (v, w), où v sera le numéro du voisin, et w le poids de l'arête correspondante.

Exercice 8 : Ecrire les instructions Python permettant de déclarer avec ce principe le graphe suivant :



4.2.1 Calcul des distances minimales depuis un sommet donné

Principe de l'algorithme de Dijkstra :

Paramètres d'entrée : un graphe pondéré implémenté sous la forme d'une liste d'adjacence, et un sommet de départ d.

En sortie : la fonction retourne la liste des distances minimales entre le sommet d et tous les sommets du graphe.

Description de l'algorithme : Soit n le nombre de sommets du graphe.

1. On crée une liste `dist` des distances,de taille n, dont toutes les valeurs sont initialisées à $+\infty$, sauf `dist[d]` qui vaut 0.
2. On va visiter les différents sommets du graphe, et il faut savoir lesquels ont déjà été visités ou non. On utilise une liste `visites` de taille n, de booléens initialement tous à `False` .

3. Tant que tous les sommets n'ont pas encore été visités (i.e. tant que `False in visites`) :
   3.1. On cherche, parmi les sommets pas encore visités, celui qui est à la plus petite distance du sommet de départ. On l'appelle u pour la suite.
   3.2. Si cette distance est infinie, on sort de la boucle "Tant que" (aucun sommet non encore visité n'est relié à u) en retournant la liste `dist` .
   3.3. Sinon :
      3.3.1. On passe le statut de u à "visité"
      3.3.2. On parcourt les voisins de u, et pour tous les voisins v de u qui ne sont pas encore visités, on actualise si besoin leur distance :
      Si `dist[u]+poids(u,v)<dist[v]` , alors `dist[v]=dist[u]+poids(u,v)`
4. On retourne la liste `dist` .

Exemple 11 : Dérouler "à la main" l'algorithme de Dijkstra, avec le graphe de l'exercice 8, en partant du sommet 0. On présentera les étapes sous forme d'un tableau des distances que l'on actualise au fur et à mesure.

Exercice 9 (*Implémentation de l'algorithme*) : .

1. *(implémentation de l'étape 3.1.)* Écrire une fonction `sommet_min(dist,visites)` qui prend en argument une liste `dist` d'entiers positifs ou égaux à $+\infty$ et une liste `visites` de booléens , et qui renvoie l'indice du minimum de `dist` parmi les indices i tels que `visites[i]` est `False`. (on suppose que la liste `visites` contient au moins un élement `False`)
2. Ecrire une fonction `Dijkstra` répondant au problème posé, i.e. prenant en entrée une liste d'adjacence comme décrit dans l'exercice 8, et un sommet de départ d, et retournant la liste des distances minimales du sommet d à tous les autres sommets. Dans cette fonction, vous utiliserez la fonction `sommet_min(dist,visites)` construite à la question précédente.
   Tester cette fonction avec l'exemple du graphe de l'exercice 9, en partant du sommet 0.

4.2.2 Calcul des distances minimales depuis un sommet donné et obtention du chemin correspondant

On souhaite à présent, en plus de déterminer la longueur du plus court chemin entre un sommet de départ d et les autres sommets, de connaitre le chemin qui permet d'obtenir cette longueur, c'est-à-dire de connaître la suite de sommets à parcourir depuis d pour avoir le chemin de longueur minimum.

Pour cela, il suffit de modifier légèrement l'algorithme de Dijkstra, pour qu'à chaque fois qu'une distance du tableau est actualisée, on note aussi le prédécesseur qui permet d'améliorer cette distance. A la fin de l'algorithme, il suffira de remonter la liste des prédécesseurs pour connaître le chemin de d jusqu'à chaque sommet.

Concrètement, si n est le nombre de sommets du graphe, on va créer une liste `pred` de longueur n, dont tous les éléments seront initialisés à `None`. A la fin du programme `pred[i]` contiendra le prédécesseur immédiat du sommet i pour le chemin minimal trouvé jusqu'à i.

Exemple 12 : Dérouler "à la main" l'algorithme de Dijkstra avec prédécesseurs, avec le graphe de l'exercice 8, en partant du sommet 0. On présentera les étapes sous forme d'un tableau des distances et des prédécesseurs, que l'on actualise au fur et à mesure.

Exercice 10 : 1. Ecrire une fonction `DijkstraPredecesseurs` répondant au problème posé, i.e. prenant en entrée une liste d'adjacence comme décrit dans l'exercice 8, et un sommet de départ d, et retournant la liste des distances minimales du sommet d à tous les autres sommets, et la liste des prédécesseurs de chaque sommet.

2. Implémenter en Python une fonction `PlusCourtChemin(G,d,a)`, de paramètres `G` liste d'adjacence du graphe, `d` et `a` deux sommets du graphe.

   Cette fonction devra déterminer (s'il existe) le plus court chemin pour aller du sommet `d` au sommet `a`.
   Si oui elle renverra ce chemin sous forme d'une liste de sommets (le premier élément de la liste étant `d` et le dernier étant `a`), sinon la fonction renverra `False`.

5 Exercice : Calcul du diamètre d'un graphe

Exercice 11 : *Le diamètre d'un graphe est la plus grande distance possible qui puisse exister entre deux de ses sommets (la distance entre deux sommets étant définie par la longueur d'un plus court chemin entre ces deux sommets).*

1. Implémenter une fonction qui calcule le diamètre d'un graphe non pondéré.(donné par sa liste d'adjacence)
2. Même question avec un graphe pondéré.

6 Deux exemples de graphes pour s'entrainer

.

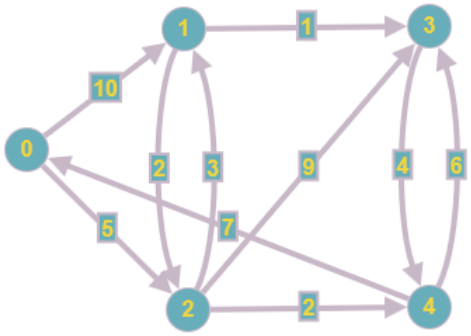


FIGURE 1 – Graphe A

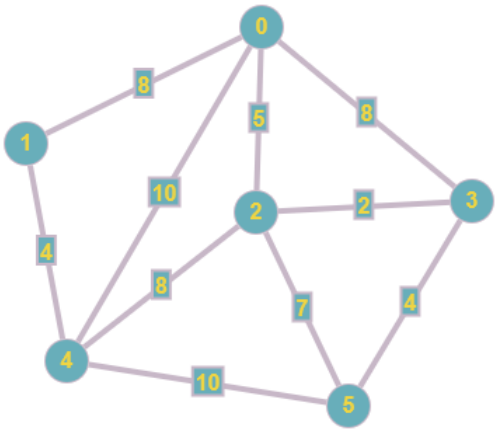


FIGURE 2 – Graphe B

7 Pour aller plus loin

7.1 Le parcours en profondeur récursif

Dans le paragraphe 3.1, nous avons implémenté le parcours en profondeur d'un graphe (DFS) de manière itérative en utilisant une structure de pile explicite (pile.pop(), pile.append()) et un tableau de marquage.

L'objectif maintenant est d'écrire la version récursive de ce même algorithme. En programmation récursive, c'est la pile d'appels de la machine qui remplace notre pile explicite.

Exercice 12 : Écrire une fonction récursive `prof_rec(L, s, marquage=None)` qui prend en paramètres le graphe L (liste d'adjacence) et un sommet de départ s, et qui renvoie la liste des sommets visités dans l'ordre du parcours en profondeur.

Indications :

- *Tout comme dans votre version itérative, il faut un tableau* `marquage` *de taille* `len(L)` *initialisé à* `False` *pour éviter de boucler indéfiniment.*
  Astuce : Utilisez un paramètre optionnel `marquage=None`*. Si* `marquage` *vaut* `None` *(lors du tout premier appel par l'utilisateur), initialisez votre liste de booléens. Transmettez ensuite cette même liste à chaque appel récursif.*
- *Visiter un sommet : Dès qu'on entre dans la fonction pour un sommet s, on commence par le marquer (*`marquage[s] = True`*) et on l'ajoute aux sommets visités.*
- *Explorer les voisins (Remplacement de la boucle et de la pile) :*
  Dans votre version itérative, vous empiliez les voisins non marqués.
  Dans la version récursive, au lieu de les empiler dans une liste pile, on appelle directement la fonction récursive pour chaque voisin non marqué, et on accumule les résultats obtenus.

7.2 Une amélioration de l'algorithme de Dijkstra : L'algorithme $\mathrm{A}^\star$

L'algorithme de Dijkstra a pour inconvénient de visiter tous les points du graphe. En particulier, il peut s'éloigner énormément de la trajectoire qui semble la plus courte pour aller du sommet d au sommet a.
On peut donc améliorer la rapidité de cet algorithme en utilisant une **heuristique** *(heuristique, nom féminin : art de trouver, de découvrir)*, qui est **une estimation de la distance qui reste à parcourir jusqu'à l'arrivée.**

Par exemple si on cherche à minimiser un temps de parcours en voiture, en supposant que l'on ne dépasse jamais les limitations de vitesse, une heuristique possible est la distance à vol d'oiseau entre les deux points divisée par la vitesse maximale sur autoroute. Dans la suite on notera $h(u)$ l'estimation de la distance à parcourir entre un sommet u et le sommet d'arrivée a.

Comment se déroule l'algorithme :
On reprend le principe de l'algorithme de Dijkstra, mais à chaque étape, au lieu de choisir le sommet non encore exploré à distance minimale du départ, **on choisit le sommet u tel que `dist[u]+h(u)` soit minimal**.

Autrement dit **on minimise "distance déjà faite + estimation de la distance qu'il reste"**

Une heuristique est dite admissible si elle ne surestime jamais la distance restante, c'est-à-dire que pour tout sommet u du graphe, on a $0 \leq h(u) \leq distance(u, a)$.
On peut montrer que si l'heuristique est admissible, alors l'algortihme retourne bien la plus petite distance entre d et a.

Pourquoi est-il si efficace ? Grâce à l'estimation $h(u)$, l'algorithme a une sorte de "boussole" : il oriente ses recherches en priorité vers la bonne direction (contrairement à Dijkstra qui explore "bêtement" en cercle).

Remarque : Si h est la fonction identiquement nulle, on retrouve l'algorithme de Dijkstra.