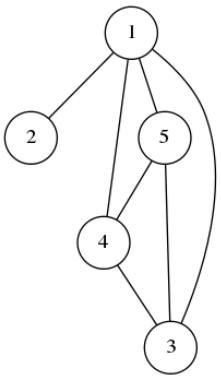
I Graphes simples

1. Définitions

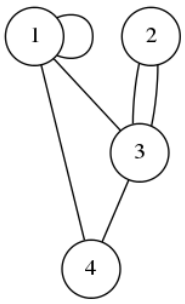
On appelle **graphe simple** un couple $G = (V, E)$ où V est un ensemble de points appelés « sommets » (*vertices* en anglais) et E est une partie de $V \times V$ telle que $\forall x \in V, (x, x) \notin E$; les éléments de E sont appelés « arêtes » (*edges* en anglais). On peut représenter un graphe par un dessin dans le plan en associant un point à chaque sommet et une portion de droite (ou de courbe) aux arêtes. La figure suivante correspond au graphe $G = (V, E)$, où $V = \{1, 2, 3, 4, 5\}$ et $E = \{(1,2), (2,1), (1,4), (4,1), (1,5), (5,1), (1,3), (3,1), (3,4), (4,3), (3,5), (5,3), (4,5), (5,4)\}$.



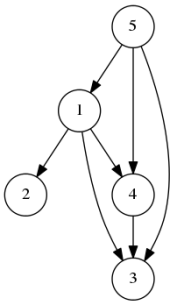
Il existe de nombreuses situations que l'on peut modéliser à l'aide d'un graphe : une carte routière, les pages internets (où les arêtes sont les liens hypertextes entre elles), les réseaux sociaux,...

Remarque(s) :

- La notion de graphe simple provient du fait que deux sommets ne sont reliés que par au plus une arête et qu'aucune arête ne joint un point avec lui même (pas de boucle autour d'un sommet).
- Un graphe est dit « non orienté »si $\forall (x, y) \in E, (y, x) \in E$ (donc on peut parcourir les arêtes dans n'importe quel sens) ; dans le cas contraire, il est dit orienté.



Un multigraphe



Un graphe orienté
$V = \{1, 2, 3, 4, 5\}$ et
$E = \{(1,2), (1,3), (1,4), (5,1), (4,3), (5,3), (5,4)\}$

2. Représentations

Il existe différentes façons de représenter un graphe simple :

— par sa **matrice d'adjacence** : c'est la matrice $A = (a_{i,j})_{1 \leqslant i,j \leqslant n}$, où $n = \text{Card}(V)$, définie par

$$a_{i,j} = \begin{cases} 1 & \text{si } (x_i, x_j) \in E \\ 0 & \text{sinon} \end{cases}$$

lorsque l'on a numéroté les sommets $E = \{x_1, \ldots, x_n\}$.
Pour un graphe simple, on a toujours $a_{i,i} = 0$ et pour un graphe non orienté, cette matrice est symétrique.
Pour l'exemple initial, la matrice d'adjacence est

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

— par sa **liste d'adjacence** : on donne la liste des arêtes.
Dans le premier exemple :

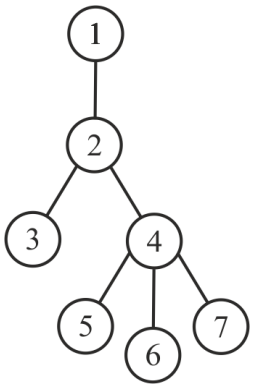
`L=[(1,2),(2,1),(1,4),(4,1),(1,5),(5,1),(1,3),(3,1),(3,4),(4,3),(3,5),(5,3),(4,5),(5,4)]`.

— par un dictionnaire : à chaque sommet, qui sera une clé, on associe la liste des sommets qui lui sont reliés.
Pour ce même exemple :

`dico={'1':['2','3','4','5'],'2':['1'],'3':['1','4','5'],'4':['1','3','5'],'5':['1','3','4']}`.

3. Arbres

On dit qu'un graphe simple non orienté G est **connexe** si pour toute paire de sommets $\{x, y\}$ de G, il existe une succession d'arêtes permettant de passer de x à y.
On dit qu'un graphe simple non orienté g est **sans cycle** s'il n'existe aucun sommet x de G que l'on peut joindre à lui même par une succession d'arêtes (sans revenir sur ses pas).
Le premier graphe dessiné est connexe (il est « en un seul morceau ») mais n'est pas sans cycle : on peut créer un cycle avec les sommets 1, 4 et 5 par exemple.
Un **arbre** est un graphe simple non orienté connexe et sans cycle.



Dans un arbre, on distingue deux types de sommets :
— les **feuilles**, qui sont les sommets reliés à une seule arête : sur l'arbre précédent, les feuilles sont les sommets 1, 3, 5, 6 et 7.
— les **nœuds**, qui sont tous les autres sommets : 2 et 4 dans cet exemple.

II Parcours d'un graphe

On cherche à parcourir tous les sommets d'un graphe simple non orienté et connexe, par exemple de façon à trouver une information sur des pages web. On peut proposer deux façons de parcourir le graphe : un parcours « en profondeur » ou un parcours « en largeur ». On note `taille` le nombre de sommets (taille de la matrice d'adjacence).

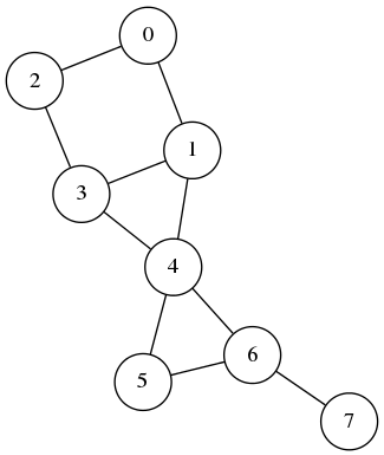
1. Parcours en profondeur

Dans un parcours en profondeur, on part du sommet de départ, on choisit un arête et on essaie d'aller le plus loin possible dans le graphe ; lorsqu'on est bloqué, on remonte à la dernière bifurcation rencontrée et on explore une autre piste.
Le parcours en profondeur utilise une pile `P` et se déroule de la façon suivante :
— on empile le sommet de départ dans `P`
— tant que la pile `P` n'est pas vide, si le sommet de la pile `P` possède un voisin qui n'a pas encore été examiné, on l'empile et on recommence ; s'il n'a pas de voisin, on dépile la pile `P` et on recommence
Pour réaliser ce parcours, on a donc besoin, en plus de la pile `P`, d'une liste `parcourus` pour mémoriser les sommets que l'on a déjà examinés.

```
def profondeur(depart) :
    P = CreerPile()
    parcourus = [depart]
    Empiler(P, depart)
    while not EstVide(P) :
        S = Sommet(P)
        i = 0
        trouve = False
```

```
        while i < taille and not trouve :
            if G[S][i] != 0 and not i in parcourus :
                trouve = True
            i += 1
        if trouve :
            Empiler(P,i-1)
            parcourus.append(i-1)
        else :
            Depiler(P)
    return parcourus
```

La fonction précédente renvoie la liste `parcourus` qui contient les indices des sommets dans l'ordre où ils ont été découverts par la recherche en profondeur.



La matrice d'adjacence :

```
G = np.array([[0,1,1,0,0,0,0,0],
              [1,0,0,1,1,0,0,0],
              [1,0,0,1,0,0,0,0],
              [0,1,1,0,1,0,0,0],
              [0,1,0,1,0,0,1,1,0],
              [0,0,0,0,1,0,1,0],
              [0,0,0,0,1,1,0,1],
              [0,0,0,0,0,0,1,0]])
taille = 8
```

`profondeur(3)` renvoie `[3, 1, 0, 2, 4, 5, 6, 7]` alors que `profondeur(6)` renvoie `[6, 4, 1, 0, 2, 3, 5, 7]`.

2. Parcours en largeur

Dans un parcours en largeur, on part du sommet de départ et on examine tous les sommets qui sont à une distance 1 du départ, puis ceux à distance 2, ...

Pour le parcours en largeur, on utilise une **file**, une autre structure de donnée linéaire qui suit le protocole FIFO , « premier rentré, premier sorti » (*first in, first out*) : on enfile les éléments à la fin de la file et on les récupère en partant du premier.

Pour créer la structure de file, qui n'existe pas sous Python, on peut se contenter d'utiliser une liste : on ajoutera les éléments à la fin de la liste par la méthode `append` et on enlèvera la tête de la file par slicing.

Le principe du parcours en largeur est le suivant :

- on enfile le sommet de départ.
- tant que la file n'est pas vide : si la tête de file possède des voisins qui n'ont pas encore été visités, on les enfile tous ; sinon on défile la tête de file.

Un code réalisant ce parcours est le suivant :

```
def largeur(depart) :
    F = [depart]
    vus = [depart]
    while len(F) > 0 :
        T = F[0]
        i = 0
        voisin = False
        while i < taille :
            if G[T][i] != 0 and i not in vus :
                voisin = True
                F.append(i)
                vus.append(i)
            i += 1
        if not voisin :
            F = F[1:]
    return vus
```

L'exécution de `largeur(3)` renvoie `[3, 1, 2, 4, 0, 5, 6, 7]` et celle de `largeur(6)` renvoie `[6, 4, 5, 7, 1, 3, 0, 2]`.

III Algorithme de Dijkstra

1. Position du problème : recherche du plus court chemin

On s'intéresse au problème suivant : on dispose d'une carte où sont placées n villes, numérotées de 0 à $n-1$, et les distances des routes qui les relient. Pour modéliser une telle carte, on utilisera un tableau `carte` à deux dimensions rempli de la façon suivante :

$$\texttt{carte}[i][j] = \begin{cases} d_{i,j} & \text{si les villes } i \text{ et } j \text{ sont reliées par une route de longueur } d_{i,j} \\ -1 & \text{si aucune route ne relie directement ces deux villes} \end{cases}$$

On peut commencer par remarquer que ce tableau est symétrique par définition.
On cherche à relier une ville de départ quelconque à une ville d'arrivée quelconque par le plus court chemin possible.
Le principe de fonctionnement de l'algorithme de Dijkstra est le suivant : on parcourt les différentes villes de la carte en cherchant à chaque étape, parmi les villes que l'on a pas encore parcourues, la ville la plus proche de la ville de départ. Il s'agit donc, pour chacune des villes de la carte, de déterminer le plus court chemin permettant d'y accéder depuis la ville de départ ; ceci jusqu'à ce que l'on trouve le plus court chemin ralliant la ville d'arrivée.

Pour mémoriser les résultats de cette recherche, on va créer trois listes de longueur n :
- La liste `distances` qui comporte les distances les plus courtes trouvées entre la ville de départ et les différentes villes de la carte : `distances[i]` est la valeur de la plus courte distance trouvée jusqu'ici pour relier la ville de départ et la ville `i`.
- La liste `parcourues` qui référence les villes que l'on a déjà examinées : `parcourues[i]` vaut `True` ou `False` selon qu'on a déjà examiné les routes qui partent de la ville `i` ou non.
- La liste `antécédents` qui permettra de reconstruire le chemin le plus court : `antecedents[i]` est le numéro de la ville à partir de laquelle on arrive dans la ville `i` dans le chemin le plus court trouvé jusqu'ici.

2. Initialisation des listes

Avant de commencer, on initialise les trois listes de la façon suivante :
- Le liste `distances` est initialisée avec `distances`[depart]=0 : on part de la ville `depart` et la distance parcourue depuis la ville de départ est 0. Les autres éléments de la liste sont initialisés avec la valeur -1 qui servira à marquer qu'une ville n'a pas encore été visitée.
- La liste `parcourues` est initialisée avec `parcourues[depart]=True`, toutes les autres coordonnées valent `False` : on part de la ville `depart` et on n'a pas encore visité les autres.
- La liste `antecedents` est initialisée avec toutes les coordonnées égales à `None` :

```
distances = [-1] * n
parcourues = [False] * n
antecedents = [None] * n

def Initialisation(depart) :
distances[depart] = 0
parcourues[depart] = True
```

3. Fonctionnement de l'algorithme

Depuis la ville de départ

On commence par déterminer les villes que l'on peut rejoindre directement depuis la ville `depart` et on met à jour les valeurs des distances correspondantes : les villes que l'on peut rejoindre depuis la ville `depart` sont les villes d'indice `i` telles que `carte`[depart][i] > 0 ; pour ces indices `i`, on change les valeurs de la liste `distances` en effectuant `distances[i] = carte[depart][i]`. Dans le même temps, on met à jour la liste des antécédents : pour les indices `i` précédents, on fait `antecedents[i] = depart` ce qui signifie que l'on arrive dans ces villes en venant directement de la ville `depart`.
On cherche ensuite parmi ces villes laquelle est la plus proche de la ville `depart`, c'est donc une ville d'indice `i` pour laquelle on a `distance[i] > 0` (une ville dont on vient de mettre à jour la distance à la ville de départ) et pour laquelle la valeur de `distances[i]` est la plus petite (parmi les valeurs > 0) ; s'il existe plusieurs indices pour lesquels la distance à la ville de départ est minimale, on en choisit un. On met alors à jour la liste `parcourue` par `parcourues[i] = True` pour l'indice `i` précédent, ce qui signifie que l'on s'est déplacé dans cette ville `i`.

Et ensuite...

Si on est arrivé à une ville `i` : on détermine les villes que l'on peut relier directement depuis la ville `i` et qui n'ont pas encore été visitées, c'est-à-dire les villes d'indice `j` pour lesquelles `carte[i][j] > 0` et `parcourues[j] == False`. Pour

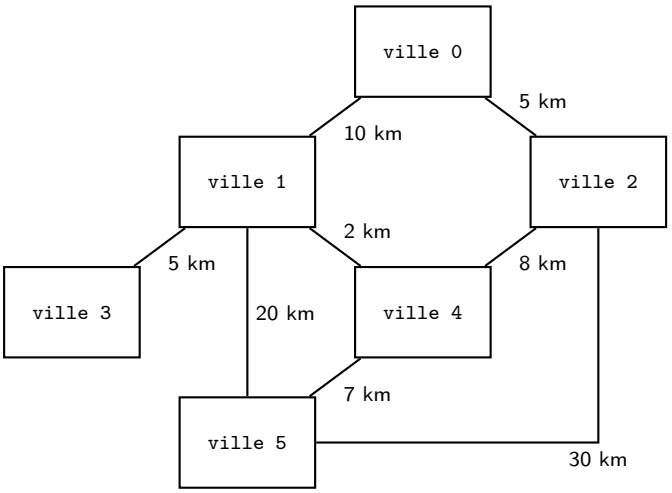
ces villes, on met à jour la liste `distances` : si une ville `j` n'a pas encore été examinée, `distances[j]` vaut alors -1, on change la valeur de cette liste par `distances[j] = carte[i][j]`. Si la ville `j` a déjà été examinée, ce qui signifie que l'on avait déjà relié cette ville `j` à la ville `depart` par un autre chemin, on compare les deux chemins : si `distance[j] > distance[i] + carte[i][j]`, ce qui signifie que le nouveau chemin est plus court que celui que l'on avait avant, on change la valeur de `distances[j]` en `distances[j] = distances[i] + carte[i][j]` ; dans le cas contraire, on ne fait rien pour cet indice. On met à jour la liste des antécédents : `antécédents[j] = i` pour toutes les villes dont on a modifié la valeur correspondante de la liste `distances`.
On cherche alors la ville pour laquelle la distance à la ville `depart` est la plus courte et qui n'a pas encore été visitée : c'est une ville d'indice `k` pour laquelle `parcourues[k] = False` et `distances[k]` est minimale. On se « déplace » à la ville `k`, ce que l'on repère par `parcourues[k] = True`, avant de recommencer depuis la ville `k`.

4. Fin de l'algorithme

L'algorithme se termine quand la ville d'arrivée, la ville `arrivee`, est la ville que l'on devrait examiner, c'est-à-dire la ville qui n'a pas encore été visitée, qui est joignable depuis une des villes de la carte et dont la distance à la ville de départ est minimale.
Dans ce cas, `distances[arrivee]` est la distance minimale à parcourir pour rejoindre la ville d'arrivée. Il reste à retrouver le chemin de longueur minimale. Pour cela, on utilise la liste des antécédents pour remonter le chemin : on part de la ville `arrivee`, on remonte à la ville précédente qui est `antecedents[arrivee]` (disons k), la ville précédente est alors `antecedents[k]` et ainsi de suite jusqu'à retomber sur la ville `depart` ; il reste alors seulement à remettre les villes traversées dans le bon ordre.

5. Un exemple illustré

Pour illustrer le fonctionnement de l'algorithme, considérons la « carte routière » suivante avec 6 villes, numérotées de 0 à 5. On cherche le plus court chemin pour aller de la ville 0 à la ville 5.

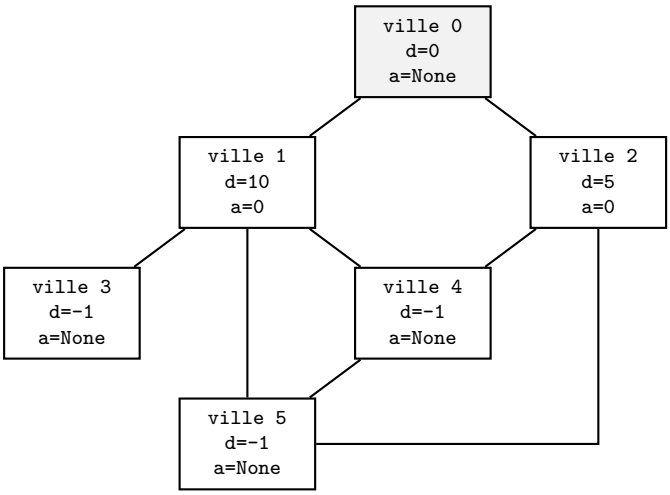


Le tableau `carte` utilisé pour représenter cette situation est le suivant :

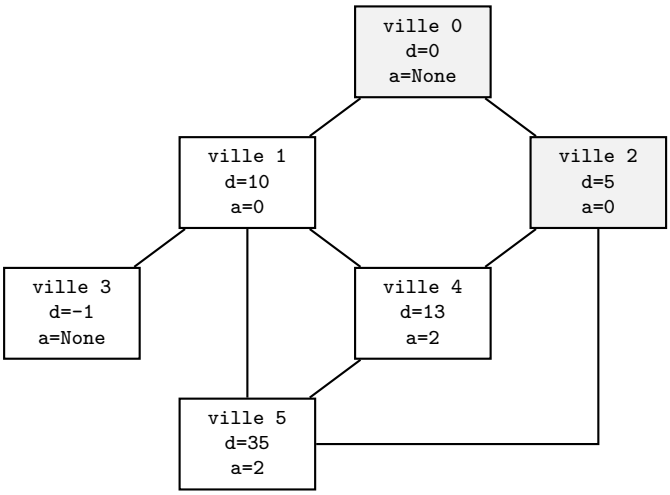
	0	1	2	3	4	5
0	0	10	5	-1	-1	-1
1	10	0	-1	5	2	20
2	5	-1	0	-1	8	30
3	-1	5	-1	0	-1	-1
4	-1	2	8	-1	0	7
5	-1	20	30	-1	7	0

On représente la progression de l'algorithme en marquant pour chaque ville les valeurs des listes `distances` et `antecedents` dans la case de la ville (`d=?,a=?`). Les cases grisées sont les cases qui ont déjà été parcourues.

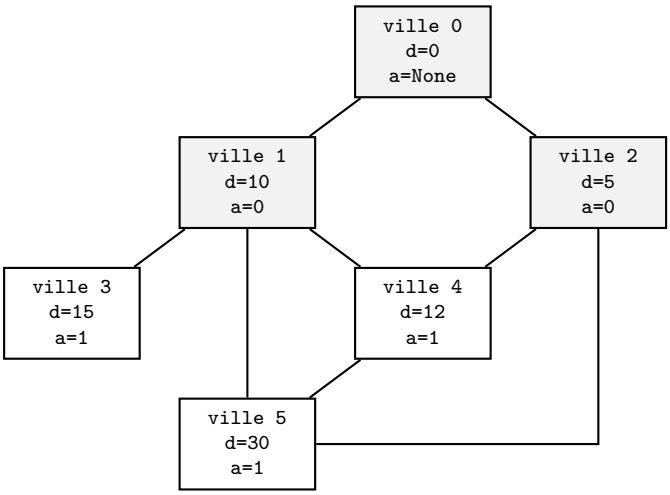
On met à jour les distances des villes 1 et 2 qui sont accessibles depuis la ville 0 :



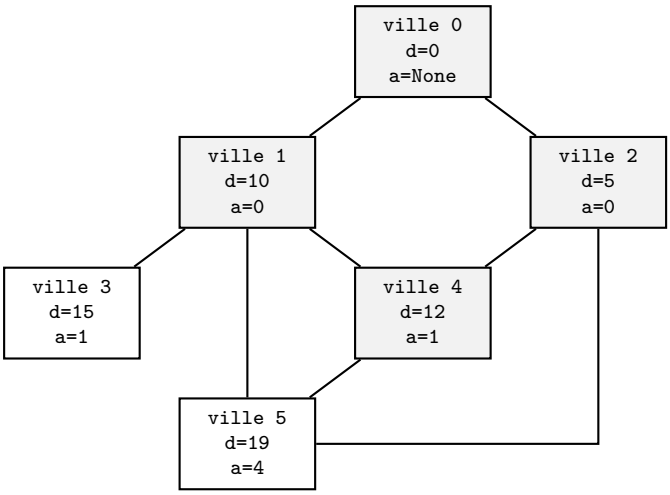
On se déplace à la ville 2 et on met à jour les distances des villes 4 et 5 qui sont accessibles depuis la ville 2 :



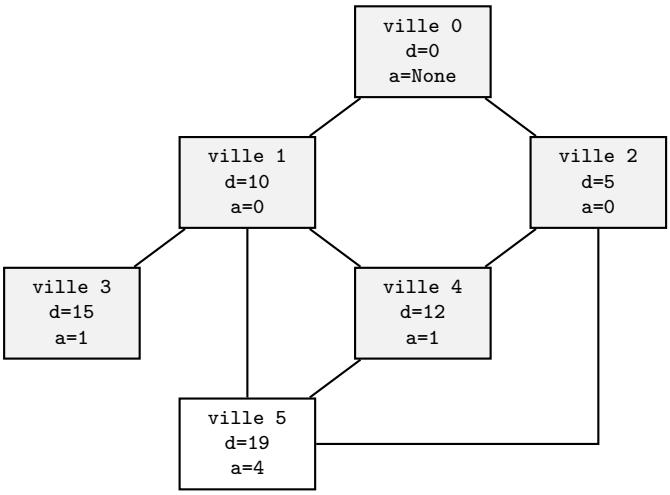
On se déplace à la ville 1 et on met à jour les distances des villes 3, 4 et 5 qui sont accessibles depuis la ville 1 ; on trouve un chemin plus court pour aller aux villes 4 et 5 ; on modifie leurs antécédents :



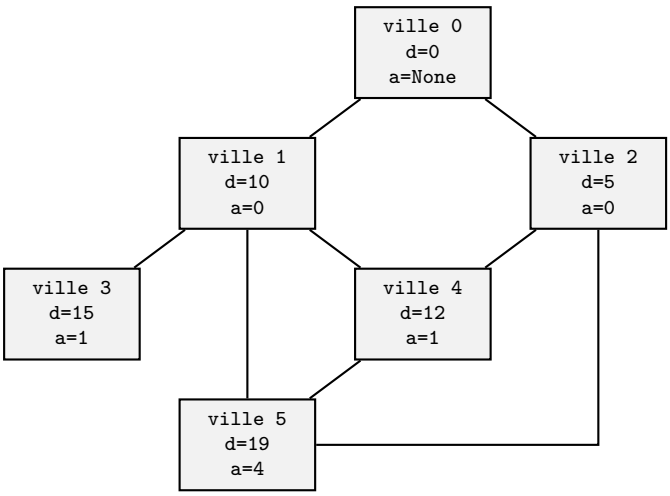
On se déplace à la ville 4 et on met à jour la distance à la ville 5 qui est accessible depuis la ville 4 ; on trouve un chemin plus court pour aller à la ville 5 et on modifie son antécédent :



On se déplace à la ville 3 qui n'ouvre pas de nouvelle route :



On se déplace à la ville 5 donc l'algorithme se termine. Le plus court chemin mesure 19 km et on « remonte » les antécédents : on arrive en 5 depuis 4, où on arrive depuis la ville 1, où on arrive depuis 0. Le plus court chemin est donc 0,1,4,5 :



1. En appliquant l'algorithme décrit plus haut, déterminer le plus court chemin permettant d'aller de la ville 2 à la ville 3.

6. Le code

Les fonctions auxiliaires

On va programmer trois fonctions auxiliaires avant l'algorithme proprement dit. La première permet de déterminer les villes qui sont directement accessibles depuis une ville donnée, la deuxième met à jour la liste des `distances` à partir d'une ville donnée, et la troisième permet de choisir la ville suivante à examiner.

2. Écrire une fonction `accessible(ville)` qui renvoie la liste des villes que l'on peut rejoindre depuis une ville d'indice `ville` : il s'agit de déterminer les villes d'indices `i` pour lesquelles `carte[ville][i] > 0`.
3. Écrire une fonction `MajDistances(ville)` permettant de mettre à jour les distances à la ville `départ` des villes accessibles depuis une ville d'indice `ville`. Elle modifie les listes `distance` et `antecedents` quand c'est nécessaire. On peut remarquer qu'il est inutile de faire un test supplémentaire pour vérifier si une ville accessible `i` a déjà été parcourue ou non. En effet, pour une telle ville `i`, `distances[i]` est déjà la plus courte distance depuis la ville `départ` donc on aura forcément `distances[i] < distances[ville] + carte[ville][i]`.
4. Écrire une fonction `suivant(ville)` qui renvoie la ville suivante à examiner. Il s'agit de trouver parmi les villes non encore parcourues, une ville dont la distance à la ville de départ est minimale.

Le code principal

5. Il ne reste plus qu'à faire une boucle tant que la ville « suivante » n'est pas la ville d'arrivée.
   Écrire une fonction `Dijkstra(depart,arrivee)` renvoyant la distance la plus courte entre la ville de départ et la ville d'arrivée ainsi que le chemin le plus court depuis la vile de départ.