ITC2

Graphes

C. MULLAERT

Lycée Saint-Louis

Année 2024-2025

Rappels sur les graphes

- Un **graphe** G est un couple $G = (S, A)$ où:
  - S est un ensemble non vide. Les éléments de S sont appelés les **sommets** ou les **noeuds** du graphe G.
  - A est un ensemble de couples (x, y) ou de paires $\{x, y\}$ avec $x, y \in S$. Les éléments de A sont appelés les **arêtes** du graphe G.
- Si les éléments de A sont des paires, comme la paire $\{x, y\}$ est égale à la paire $\{y, x\}$, on dit que G est un **graphe non orienté**. Dans ce cas, les arêtes n'ont pas de sens de parcours.
- Si les éléments de A sont des couples, comme le couple (x, y) est en général différent du couple (y, x), on dit que G est un **graphe orienté**. Dans ce cas, les arêtes ont un sens de parcours et les arêtes sont également appelées **arcs**.

- Si deux sommets sont reliés par une arête, on dit qu'ils sont **adjacents** ou **voisins**. Ces deux sommets sont alors appelés les **extrémités** de l'arête.
- Une arête de la forme $\{x, x\}$ est appelée une **boucle**.

Degré

Soit $G = (S, A)$ un graphe et s un sommet de G.

- Si G est un graphe non orienté, alors le **degré** de s noté $d(s)$ est le nombre d'arêtes dont s est une extrémité, en comptant les boucles pour 2 arêtes.
- Si G est un graphe orienté.
  - Le **degré sortant** de s noté $d_+(s)$ est le nombre d'arêtes dont s est l'extrémité initiale.
  - Le **degré entrant** de s noté $d_-(s)$ est le nombre d'arêtes dont s est l'extrémité finale.

Connexité

Soit $G = (S, A)$ un graphe.

- Soient s_0 et s_n des sommets de G. Un **chemin** de s_0 à s_n est une suite de sommets adjacents $(s_0, s_1, \ldots, s_n)$ reliant s_0 et s_n. Le nombre d'arêtes du chemin $(s_0, s_1, \ldots, s_n)$ est alors n et est appelé la **longueur du chemin** $(s_0, s_1, \ldots, s_n)$
- La **distance** entre deux sommets est la longueur minimale d'un chemin reliant ces deux sommets. Si ces deux sommets ne peuvent être reliés, on considère alors que la distance est infinie.
- Un **cycle** est un chemin reliant un sommet à lui même.
- On dit qu'un graphe est **connexe** si, pour tout couple de sommets, il existe un chemin reliant ces deux sommets.

Pondération d'un graphe

On dit qu'un graphe est:

- **pondéré** si un nombre appelé **poids** est associé à chaque arête du graphe,
- **étiqueté** si un texte appelé **étiquette** est associé à chaque arête du graphe.

Un graphe pondéré ou étiqueté permet d'avoir des informations supplémentaires. Par exemple, si on considère un réseau routier dont les sommets sont les villes, on peut prendre comme poids la distance entre chaque ville et comme étiquette le nom de la route reliant les villes.

Un graphe peut être représenté par des listes d'adjacences, c'est-à-dire en précisant pour chacun des sommets la liste de ses voisins pour un graphe non orienté (resp. de ses fils pour un graphe orienté).
On peut alors représenter un graphe à l'aide d'une lite de listes : chaque élément de la liste est une liste contenant un sommet et la liste de ces voisins.

```
G=[["A",["B"]], ["B",["C","E"]], ["C",[]],...]
```

ou un dictionnaire : les clés sont les sommets et les valeurs correspondent aux clés sont les listes des voisins.

```
G={"A":["B"], "B":["C","E"], "C":[], ...}
```

Si le graphe est pondéré, on rajoute les poids :

```
G=[["A",[("B",3)]],["B",[("C",1),("E",2)]],["C",[]],...]
```

```
G = {"A":[("B",3)], "B":[("C",1),("E",2)], "C":[], ... }
```

Les choses sont simplifiées si les sommets sont numérotés.

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

On peut alors également représenter le graphe par une matrice d'adjacence c'est à dire une liste de listes T telle que $T[i][j]$ soit égale à 1 s'il existe une arête de i vers j et 0 sinon.

Dans le cas d'un graphe pondéré ou étiqueté, on remplace les 1 par l'information sur l'arête reliant i à j.

Pour parcourir tous les sommets accessibles à partir d'un sommet, on peut faire

- un parcours en profondeur : à 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.
- un parcours en largeur : à 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

La recherche d'un élément dans une liste n'est pas optimale, il serait préférable d'utiliser une structure qui permette une recherche efficace. Pour cela on utilise un dictionnaire qui permettra de stocker les sommets déjà vus (visités ou en attente)

```
def profondeur(G,s0) :
    visite=[]
    attente=[s0]
    dejavus={s0:True}
    while len(attente)>0 :
        s=attente.pop()
        visite.append(s)
        for v in G[s] :
            if v not in dejavus :
                attente.append(v)
                dejavus[v]=True
    return(visite)
```

```
def largeur(G,s0) :
    visite=[]
    attente=deque([s0])
    dejavus={s0:True}
    while len(attente)>0 :
        s=attente.popleft()
        visite.append(s)
        for v in G[s] :
            if v not in dejavus :
                attente.append(v)
                dejavus[v]=True
    return(visite)
```

L'algorithme de Dijkstra permet d'obtenir un chemin le plus court entre deux sommets d'un graphe en utilisant le fait que si un plus court chemin entre deux sommets D et A passe par un sommet I, alors la partie de ce chemin entre D et I est un plus court chemin de D à I, et la partie entre I et A est un plus court chemin entre I et A.

```
def Dijkstra(G,a,b):
    Dist_fin={a:0}
    Dist_encours={a:0}
    Pred={}
    s=a
    while s!=b :
        Mise_a_jour(G,s,Dist_fin,Dist_encours,Pred)
        s=Cle_min(Dist_fin,Dist_encours)
        if s==None :
            return "pas de chemin"
        Dist_fin[s]=Dist_encours[s]
    ch=deque([b])
    p=b
    while p!=a :
        p=Pred[p]
        ch.appendleft(p)
    return list(ch),Dist_fin[b]
```

```
def Mise_a_jour(G,s,Dist_fin,Dist_encours,Pred):
    for v in range(len(G)):
        if G[s][v]!=0 :
            if v in Dist_encours :
                if not v in Dist_fin :
                    d=Dist_encours[v]
                    d2=Dist_fin[s]+G[s][v]
                    if d>d2:
                        Dist_encours[v]=d2
                        Pred[v]=s
            else :
                Dist_encours[v]=Dist_fin[s]+G[s][v]
                Pred[v]=s
```

```python
def Cle_min(Dist_fin,Dist_encours):
    min=float("inf")
    cle_min=None
    for cle in Dist_encours :
        if cle not in Dist_fin and Dist_encours[cle]<min:
            min=Dist_encours[cle]
            cle_min=cle
    return cle_min
```

L'algorithme A* est une variante de celui de Djikstra.
À la place de chercher le sommet le plus proche de l'origine, on cherche le sommet de moindre coût, le coût c(s) d'un sommet s étant égal à d(s) + h(s), où :

- d(s) est la distance de s au point de départ s0 présente dans la liste des distances,
- h(s) est une heuristique, c'est-à-dire une fonction définie sur l'ensemble des sommets, qui permettra de définir une direction à privilégier.

Il suffit de changer `Cle_min` en :

```
def Cle_min_H(Dist_fin,Dist_encours):
    min=float("inf")
    cle_min=None
    for cle in Dist_encours :
        if cle not in Dist_fin and Dist_encours[cle]+h[cle]
            min=Dist_encours[cle]
            cle_min=cle
    return cle_min
```

L'algorithme de Floyd-Warshall consiste à déterminer l'ensemble des plus courts chemins entre toute paire de sommets d'un graphe en utilisant la programmation dynamique.

On numérote les sommets de 0 à $n-1$ et pour tout $(i,j,k) \in [\![0, n-1]\!]^2$, on note $D_k(i,j)$ la distance entre les sommets i et j en ne passant que par les sommets intermédiaires de 0 à $k-1$.

Par convention cette distance sera égale à ∞ s'il n'existe pas de chemin reliant les sommets i et j en ne passant que par les sommets de 0 à $k-1$.
On a alors :

$$D_{k+1}(i,j) = \min\left(D_k(i,j), D_k(i,k) + D_k(k,j)\right)$$

```
def FW(G):
    n=len(G)
    D=[[ inf for j in range (n)] for i in range (n)]
    for s in range(n):
        for v in range(n):
            if G[s][v]!=0:
                D[s][v]= G[s][v]
        D[s][s]=0
    for k in range (n):
        for i in range (n):
            for j in range (n):
                D[i][j]= min(D[i][j],D[i][k]+D[k][j])
    return D
```

Dans l'algorithme précédent, il n'y a pas nécessité de faire de copies de D avant de la modifier mais au k-ème passage dans la boucle $D[i][j]$ ne sera pas forcément égal à $D_k(i,j)$ mais vérifiera

$$D_n(i,j) \leq D[i][j] \leq D_k(i,j)$$

On peut bien sûr adapter pour trouver un plus court chemin d'un sommet à un autre...

```
def FW_chemin(G,a,b):
    n=len(G)
    D=[[ inf for j in range (n)] for i in range (n)]
     chemin=[[None for j in range (n)] for i in range (n)]
    for i in range(n):
        for j in range(n):
            if G[i][j]!=0 :
                D[i][j]= G[i][j]
                chemin[i][j]=[]
         chemin[i][i],D[i][i]=[],0
    for k in range (n):
        for i in range (n):
            for j in range (n):
                if D[i][k]+D[k][j]<D[i][j] :
                  D[i][j]= D[i][k]+D[k][j]
                  chemin[i][j]=chemin[i][k]+[k]+chemin[k][j]
    if chemin[a][b]==None:
        return "pas de chemin"
    return [d]+chemin[d][a]+[a]
```