Introduction aux graphes

1 Introduction : les sept ponts de Königsberg

Le problème des sept ponts de Königsberg est connu pour être à l'origine de la théorie des graphes. Résolu par Leonhard Euler en 1735, ce problème se présente de la façon suivante :

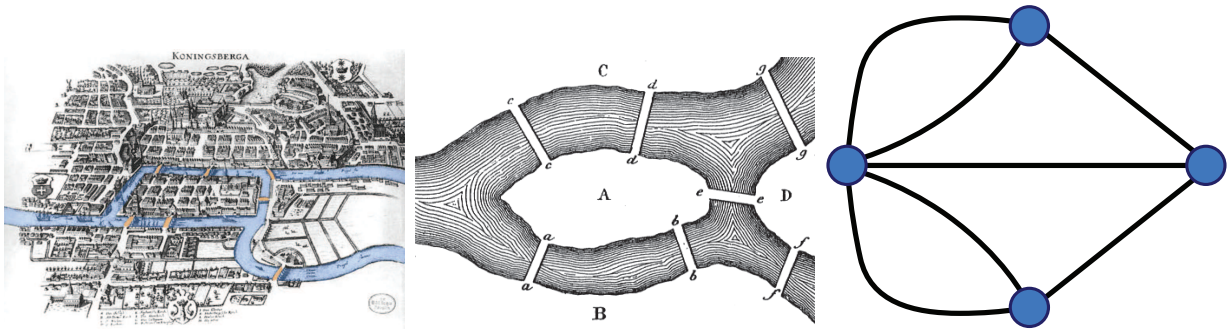


FIGURE 1 – Plan de la ville avec les ponts surlignés, et sa représentation graphique

La ville de Königsberg (aujourd'hui Kaliningrad) est construite autour de deux îles situées sur le Pregel et reliées entre elles par un pont. Six autres ponts relient les rives de la rivière à l'une ou l'autre des deux îles, comme représentés sur le plan ci-dessus. Le problème consiste à déterminer s'il existe ou non une promenade dans les rues de Königsberg permettant, à partir d'un point de départ au choix, de passer une et une seule fois par chaque pont, et de revenir à son point de départ, étant entendu qu'on ne peut traverser le Pregel qu'en passant sur les ponts.

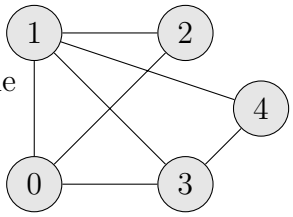
Les graphes sont utilisés dans de nombreux domaines pour modéliser les relations entre différentes entités : liaisons mécaniques entre solide, liens sur les réseaux sociaux, routes sur les réseaux de transport d'énergie, de bien ou de personnes, etc.

On remarquera que certains liens peuvent être symétriques (si une pièce est en liaison avec B alors B est en liaison avec A) ou asymétriques (A est abonné au compte de B ne veut pas dire que B est abonné à A).

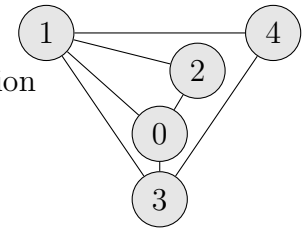
2 Graphes non orientés

2.1 Fondamentaux

Graphe 1 : exemple



Autre représentation



Définition 1 (Graphe non-orienté)**.** Un graphe non-orienté G est un couple constitué de deux ensembles (S, A) :

- un ensemble S de points nommés **sommets** ;
- un ensemble A de traits nommées **arêtes**.

Si on note $S = \{s_0, s_1, ...s_{n-1}\}$ et $A = \{a_0, a_1, ..., a_{p-1}\}$. Une arête a_i est un **ensemble** de 2 sommets $\{s_j, s_k\} \in S^2$ (l'ordre **n'a pas d'importance**).

Définition 2 (Vocabulaire de base des arêtes)**.** Soit $a_i = \{s_j, s_k\}$:

- les sommets s_j et s_k sont les **extrémités** de l'arête a_i ;
- si $s_j = s_k$, alors a_i est une **boucle**.

2.2 Type de graphe

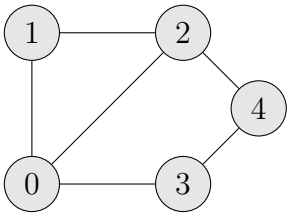
Graphe vide : Un graphe **vide** ne comporte que des sommets, l'ensemble de ses arêtes est vide ;

Graphe planaire : Un graphe est dit **planaire** si aucune de ces arêtes ne se croisent.

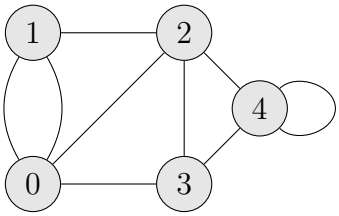
Graphe simple ou multigraphe : Un graphe est dit **simple** si au plus une arête relie deux sommets et s'il n'y a pas de **boucle** sur un sommet. Il est possible d'imaginer des graphes avec plusieurs arêtes reliant deux sommets ou avec une boucle. On appellera ces graphes des **multigraphes**.

Graphe connexe : Un graphe est **connexe** s'il est possible, à partie de n'importe quel sommet de rejoindre tous les autres sommets en suivant les arêtes. Un graphe non connexe se décompose en plusieurs sous graphes connexes.

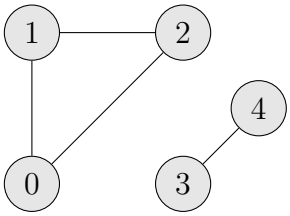
Graphe complet : Un graphe est **complet** si tous les sommets sont des voisins directs.



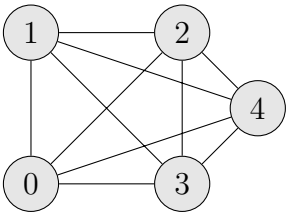
Graphe 2



Graphe 3

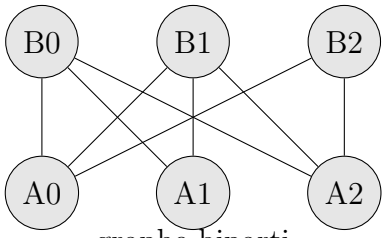


Graphe 4

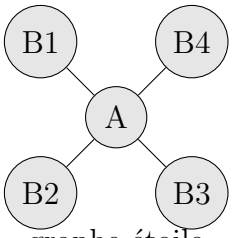


Graphe 5

Graphes bipartis et étoiles Un graphe est dit **biparti** si son ensemble de sommets V peut être partitionné en deux sous ensembles de sommets A et B de façon à ce que toute arrête ait une extrémité dans A et une extrémité dans B. Une **étoile** est donc un graphe biparti dont l'une des bipartitions n'a qu'un sommet.



graphe biparti



graphe étoile

2.3 Propriétés d'un graphe non orientés

Définition 3 (Ordre du graphe)**.** Le nombre n de sommets d'un graphe est l'**ordre de graphe**. On note $n = |S| = \#S$.

Définition 4 (Degré d'un sommet)**.** Le nombre d'arêtes dont s_i est une extrémité, ou autrement dit le nombre d'arêtes quittant s_i, est appelé **degré** de s_i et est noté $d(s_i)$.

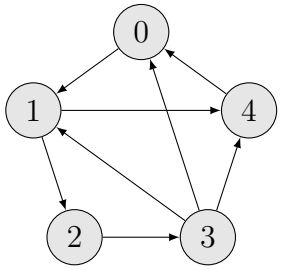
Définition 5 (Degré d'un graphe)**.** Le degré d'un graphe est le maximum de tous les degrés des sommets du graphe.

Définition 6 (Chaîne)**.** On appelle **chaîne** une liste d'arêtes successives $(a_0, a_1, ...a_{n-1})$ tel que l'une des extrémités d'une arête a_i soit l'une des extrémités de a_{i+1} $\forall i \in [\![0; n-2]\!]$.
La **longueur de la chaîne** est le nombre d'arêtes n constituant le chemin.

Définition 7 (Cycle)**.** Un **cycle** est une chaîne dont le sommet à une des extrémités de la dernière arête est aussi l'une des extrémités de la première arête.
Pour un graphe G ayant m arêtes, n sommets et p composantes connexes, on définit le nombre cyclomatiques (nombre de cycles indépendants) $\mu(G)$ du graphe par :

$$\mu(G) = m - n + p$$

3 Graphes orientés



Définition 8 (Graphe orienté)**.** Un graphe orienté G est un couple constitué de deux ensembles (S, A) :

- un ensemble S de points nommés **sommets** ;
- un ensemble A de flèches nommées **arcs**.

Si on note $S = \{s_0, s_1, ...s_{n-1}\}$ et $A = \{a_0, a_1, ..., a_{p-1}\}$. Un arc a_i est un **couple** $(s_j, s_k) \in S^2$ (l'**ordre est important**).

Définition 9 (Vocabulaire de base des arcs)**.** Soit $a_i = (s_j, s_k)$:

- le sommet s_j est l'**extrémité initiale** de a_i ;
- le sommet s_k est l'**extrémité terminale** de a_i ;
- s_k est le **successeur** de s_j ;
- s_j est le **prédécesseur** de s_k ;
- si $s_j = s_k$, alors a_i est une **boucle**.

Définition 10 (Degrés et adjacence)**.** Pour un graphe orienté, on fait la différence entre les arcs entrants et arcs sortants d'un sommet.

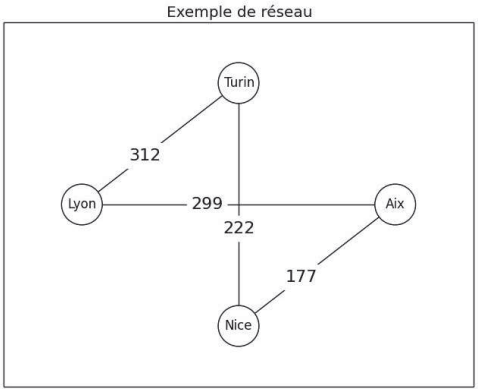
- Le nombre d'arcs dont s_i est l'extrémité initiale, ou autrement dit, le nombre d'arc sortant de s_i, est appelé **degré sortant** de s_i et est noté $d_+(s_i)$.
- Le nombre d'arcs dont s_i est l'extrémité terminale, ou autrement dit, le nombre d'arc entrant dans s_i, est appelé **degré entrant** de s_i et est noté $d_-(s_i)$.
- L'ensemble des sommets successeurs et prédécesseurs de s_i sont dits **adjacents**. Leur nombre est le **degré** de s_i : $d(s_i) = d_+(s_i) + d_-(s_i)$

Définition 11 (Chemin et circuit)**.**

- On appelle **chemin** une liste d'arcs $(a_0, a_1, ...a_{n-1})$ tel que l'extrémité terminale d'un arc a_i soit l'extrémité initiale de a_{i+1} $\forall i \in [\![0; n-2]\!]$.
- La **longueur d'un chemin** est le nombre d'arcs n constituant le chemin.
- Un **circuit** est un chemin dont l'extrémité terminale du dernier arc est l'extrémité initiale du premier.

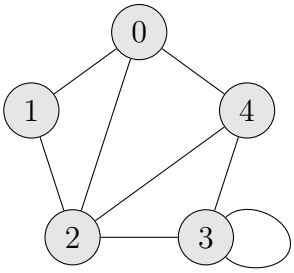
4 Autres types de graphes

Définition 12 (graphe pondéré)**.** On appelle graphe pondéré (ou valué) un graphe où les arêtes sont affectées d'un poids qui est un nombre réel. Il peut être orienté ou non. On considérera pour certains algorithmes le seul cas où le poids affecté est strictement positif. Cela représentera par exemple des situations de distances dans un réseau routier.



5 Implémentation en machine

Dans le cadre de ce cours, nous n'aborderons pas les graphes multi-arcs ou multi-arêtes.



Ce graphe $G = (S, A)$ est défini par :

- $S =$
- $A =$

5.1 Liste d'adjacence

Une liste d'adjacence correspondant à un graphe orienté est une liste de listes qui possède les propriétés suivantes :

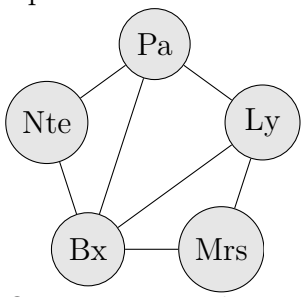
- elle est constituée de $\#S$ listes ;
- la liste i contient tous les successeurs du sommet s_i.

La liste `G=[[1,2,4],[0,2],[0,1,3,4],[2,3,4],[0,2,3]]` est une liste d'adjacence du graphe.

Pour un graphe non-orienté, chaque sous-liste est constitutée de tous les sommets adjacents au sommet considéré.

5.2 Dictionnaire d'adjacence

Une évolution ergonomique utilisé dans les graphes est de désigner un sommet par un nom particulier.



On pourrait utiliser une liste de couple (nom du sommet, liste des sommets adjacents). Mais, la manipulation d'une liste de listes de listes est trop lourde. Exemple :

```
G=[['Pa',['Nte','Bx','Ly']],['Nte',['Pa','Bx']],
['Bx',['Pa','Nte','Mrs','Ly']],['Mrs',['Bx','Ly']],['Ly',['Pa','Bx','Mrs']]]
```

Un dictionnaire d'adjacence correspondant à un graphe orienté est un dictionnaire d de listes qui possède les propriétés suivantes :

- il est constitué de $\#S$ listes ;
- $d[s_i]$ est une liste contenant les successeurs du sommet s_i.

On a donc la représentation du graphe :

```
G={'Pa':['Nte','Bx','Ly'],'Nte':['Pa','Bx'],
'Bx':['Pa','Nte','Mrs','Ly'],'Mrs':['Bx','Ly'],'Ly':['Pa','Bx','Mrs']}
```

L'important avantage comparé à un dictionnaire d'adjacence est la possibilité de nommer les sommets. On peut l'adapter de la même manière que la liste par adjacence pour représenter un graphe non-orienté.

5.3 Matrice d'adjacence

Une matrice d'adjacence correspondant à un graphe orienté est une matrice `M` qui possède les propriétés suivantes :

- il est de taille $\#S \times \#S$;
- $M[i][j] = 1$ s'il y a un arc d'extrémité initiale s_i et d'extrémité terminale s_j ;
- $M[i][j] = 0$ s'il n'existe pas un arc d'extrémité initiale s_i et d'extrémité terminale s_j.

On a donc la représentation du graphe :

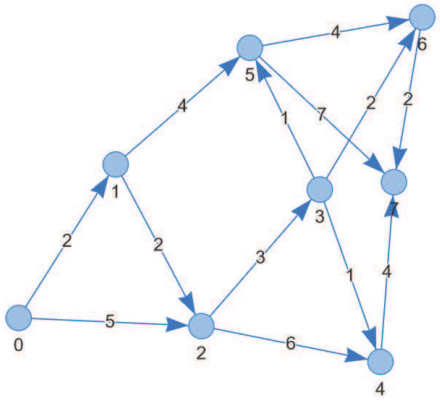
$$\mathbb{A}_G = \begin{pmatrix} 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}$$

On peut l'adapter pour représenter un graphe non-orienté : on remplace les arcs (s_i, s_j) par les arêtes $\{s_i, s_j\}$. On remarquera qu'un graphe non-orienté possède forcément une matrice d'adjacence symétrique.

Bien sûr, cette matrice peut-être codée avec une liste de listes ou un `array`.

6 Graphes pondérés

On peut enrichir la notion de graphe en attribuant des poids aux arcs et aux arêtes. Concrètement, ces poids peuvent représenter des distances, des temps de parcours, des coûts de transport, des coûts de construction, etc.



6.1 Vocabulaire

Définition 13 (Arcs et arêtes pondérés)**.** Un arc pondéré est un couple (arc, poids).
Une arête pondérée est un couple (arête, poids).

Définition 14 (Graphes pondérés)**.** Un graphe orienté $G = (S, A)$ pondéré est un couple de sommets et d'arcs pondérés.

Un graphe orienté $G = (S, A)$ non-pondéré est un couple de sommets et d'arêtes pondérées.

6.2 Matrice de pondération

Une matrice de pondération est une matrice d'adjacence adaptée :

- $M[i][j] = w$ s'il y a un arc de poids w d'extrémité initiale s_i et d'extrémité terminale s_j ;
- $M[i][j] = 0$ (ou $M[i][j] = \infty$) s'il n'existe pas un arc d'extrémité initiale s_i et d'extrémité terminale s_j.

On a donc la représentation du graphe

$$\mathbb{A}_G = \begin{pmatrix} \infty & 2 & 5 & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 2 & \infty & \infty & 4 & \infty & \infty \\ \infty & \infty & \infty & 3 & 6 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 1 & \infty & 2 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 4 \\ \infty & \infty & \infty & 1 & \infty & \infty & 4 & 7 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix}$$

Parcours de graphes

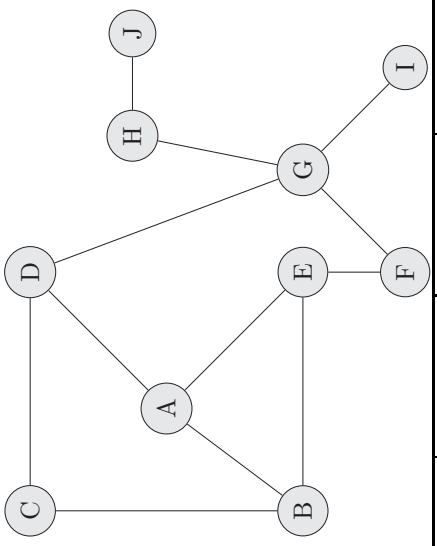
1 Objectifs

-
-

2 Parcours en largeur

2.1 Principe

2.2 Exemple



2.3 Analyse

2.4 Algorithmes

L'algorithme du parcours en largeur présenté ci-dessus peut être représenté sous le pseudo-code suivant :

Algorithme 1 : Algorithme de parcours de graphe en largeur : Version 1

```
1  Initialisation;
2  Initialiser la liste file avec le nœud départ ;
3  Initialiser la liste noeuds_connus avec le nœud départ ;
4  tant que file n'est pas vide faire
5  |  noeud_courant ← Défiler(file) ;
6  |  pour chacun des voisins v de noeud_courant faire
7  |  |  si v n'appartient pas à la liste noeuds_connus alors
8  |  |  |  Ajouter v à la liste noeuds_connus;
9  |  |  |  Enfiler v à la fin de la liste file;
10 |  |  fin
11 |  fin
12 fin
13 return noeuds_connus
```

Le précédent algorithme était illustré avec trois listes (dont la file des sommets à explorer). On peut également envisager une version où l'on ne considère que deux listes/files : la liste des sommets explorés et la file des sommets à explorer.

Algorithme 2 : Algorithme de parcours de graphe en largeur : Version 2

```
1  Initialisation;
2  Initialiser la liste file avec le nœud départ ;
3  Initialiser la liste noeuds_explores ;
4  tant que file n'est pas vide faire
5      noeud_courant ← Défiler(file) ;
6      Ajouter à la liste noeuds_explores le noeud_courant;
7      pour chacun des voisins v de noeud_courant faire
8          si v ne fait pas partie de la liste noeuds_explores alors
9              Enfiler v à la fin de la liste file;
10         fin
11     fin
12 fin
13 return noeuds_explores
```

3 Parcours en profondeur

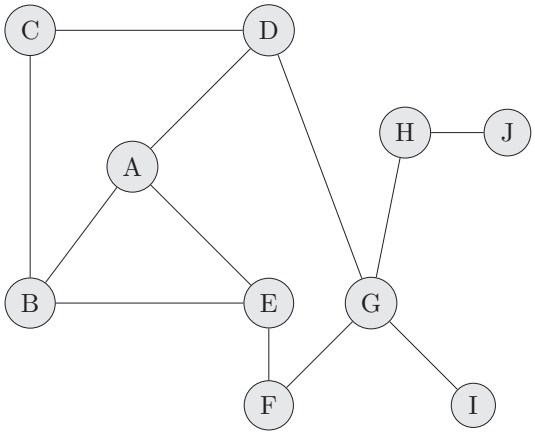
3.1 Principe

3.2 Algorithme

Algorithme 3 : Algorithme de parcours de graphe en profondeur itératif

```
1  Initialisation;
2  Initialiser la pile avec le nœud de départ;
3  Initialiser la liste noeuds_connus;
4  tant que la pile n'est pas vide faire
5      nœud courant ← Dépiler(pile) ;
6      si il existe un voisin du nœud courant non connu alors
7          Ajouter le voisin à la liste noeuds_connus ;
8          Empiler le nœud courant ;
9          Empiler ce voisin ;
10         sinon
11             Ajouter le nœud courant à la liste exploré
12         fin
13     fin
14 fin
15 return noeuds_connus
```

3.3 Exemple



3.4 Version recursive

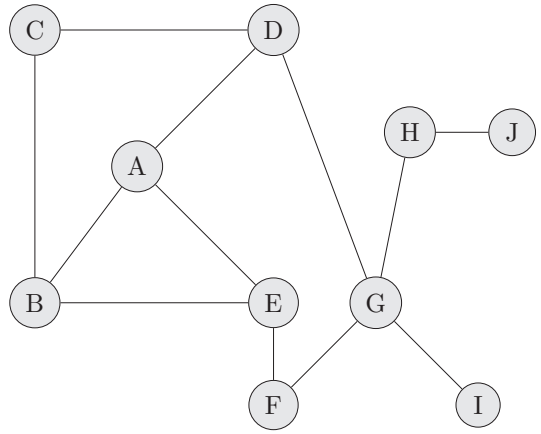
Le pseudo-code de la version de parcours de graphe en profondeur est alors la suivante :

Algorithme 4 : Algorithme de parcours de graphe en profondeur récursif

```
1 Initialisation;
2 Initialiser la liste noeuds_connus avec le nœud départ ;
3 Visiter(noeud_depart,noeuds_connus);
4 return noeuds_connus
```

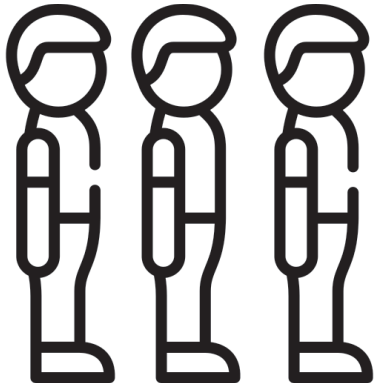
Algorithme 5 : Fonction récursive Visiter(u,noeuds_connus)

```
1 si u n'est pas dans la liste noeuds_connus alors
2 |  Ajouter u à la liste noeuds_connus;
3 fin
4 pour chaque sommet v voisin de u non connu faire
5 |  Visiter(v,noeuds_connus);
6 fin
```



4 Complément : Files & Piles

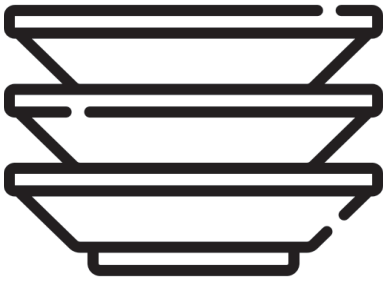
4.1 Files



Une file est une collection particulièrement simple, mais d'usage limité.
Fonctions associées :

- Créer un file vide
- Ajouter un élément : enfiler
- Extraire un élément : défiler
- Test : la file est vide ?

4.2 Piles



Une pile est aussi une collection d'usage limité.
Fonctions associées :

- Créer un pile vide
- Ajouter un élément : empiler
- Extraire un élément : dépiler
- Test : la pile est vide ?

4.3 Exemple d'implémentation en python

Un module propose un type dedié qui se rapproche des piles et files :
`from collections import deque`

- `d = deque([])` crée une collection vide ;
- `d.append(x)` et `d.appendleft(y)` permettent d'ajouter à d ou bien x à droite ou bien y à gauche ;
- `x = d.pop()` et `y = d.popleft()` permettent d'extraire le dernier élément x ou le premier y.
- `if d :` la condition est vrai si `d` existe et est non vide ;

Plus court chemin

entre deux sommets d'un graphe

1 Introduction

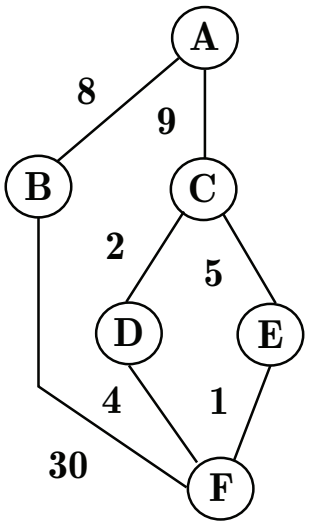
1.1 Objectifs

La problème de recherche de chemin du plus court chemin dans un graphe peut se traduire par trois niveaux de complexité :

-
-
-

Ce problème peut se poser pour des graphes orientés ou non orientés, pondérés ou non pondérés. Dans ce cours, nous nous limiterons aux graphes orientés, pondérés à pondération positives ou nulles.

1.2 Exemple



Matrice d'adjacence :

$$\mathtt{M} = \begin{bmatrix} & 8 & 9 & & & \\ 8 & & & & & 30 \\ 9 & & & 2 & 5 & \\ & & 2 & & & 4 \\ & & 5 & & & 1 \\ & 30 & & 4 & 1 & \end{bmatrix}$$

2 Algorithme de Dijkstra

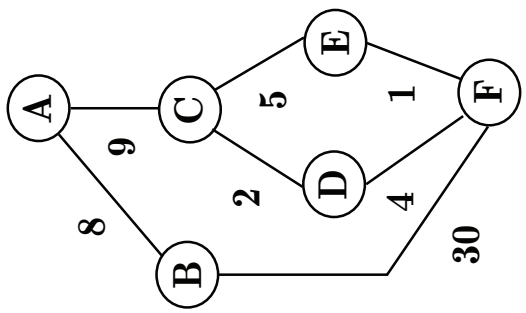
2.1 Principe

2.2 Déroulé

2.3 Exemple

Matrice d'adjacence :

$$M = \begin{bmatrix} & 8 & 9 & & & & 30 \\ 8 & & & 2 & 5 & & 4 \\ 9 & & & 2 & & 4 & 1 \\ & 2 & 2 & & 5 & & \\ & 5 & & 5 & & 1 & \\ & & 4 & & 1 & & \\ 30 & 4 & 1 & & & & \end{bmatrix}$$



3 Analyses

1 Dictionnaires

Un dictionnaire est une collection aux propriétés atypiques dont nous ferons un usage dans certains cas très particuliers. A la différence d'une liste où les éléments sont accessibles via leur index, les éléments d'un dictionnaire sont accessibles via leur « clé ».

0	2001
1	'odyssée'
2	'espace'
3	'Thomas'
4	'Pesquet'

Jean	0612345678
Josephine	0623456789
Jamel	0634567890
Jacques	0645678901
Jimmy	0656789012

Les clés doivent être constantes non modifiables (types élémentaires, `str`, `tuple`, mais pas une liste). Les valeurs peuvent être tout type d'objet.

1.1 Manipulation de dictionnaires

Usage	Syntaxe
créer un dictionnaire vide	`d = {}`
créer un dictionnaire non vide	`d = {'a' : 12, 'droit' : 'au but', 42 : [1,2,3]}`
lire une valeur sur la base de sa clé	`d['a']`
insérer une paire clé-valeur	`d[True]=888`
modifier une valeur sur la base de sa clé	`d[True]=889`
supprimer un élément sur la base de sa clé	`del(d[42])`

1.2 méthodes sur les dictionnaires

- `len(d)` renvoie le nombre d'élément du dictionnaire ;
- `key in d` teste si la clé `key` est présente dans le dictionnaire `d` ;
- `d.keys()` renvoie la liste des clefs ;
- `d.values()` renvoie la liste des valeurs ;
- `d.items()` renvoie la liste des couples (clé,valeur) ;
- `d.copy()` renvoie une copie du dictionnaire d.

1.3 Parcours d'un dictionnaire

Affichage des clés	Exemple

`for cle in dico.keys():`
`  print(cle)`
	`a`
	`droit`
	`True`

`for cle in dico:`
`  print(cle)`
	`a`
	`droit`
	`True`

Affichage des valeurs	Exemple

`for cle in dico:`
`  print(dico[cle])`
	`12`
	`au but`
	`889`

`for val in dico.values():`
`  print(val)`
	`12`
	`au but`
	`889`

Affichage des clés et des valeurs	Exemple

`for cle,val in dico.items():`
`  print(cle,val)`
	`a 12`
	`droit au but`
	`True 889`