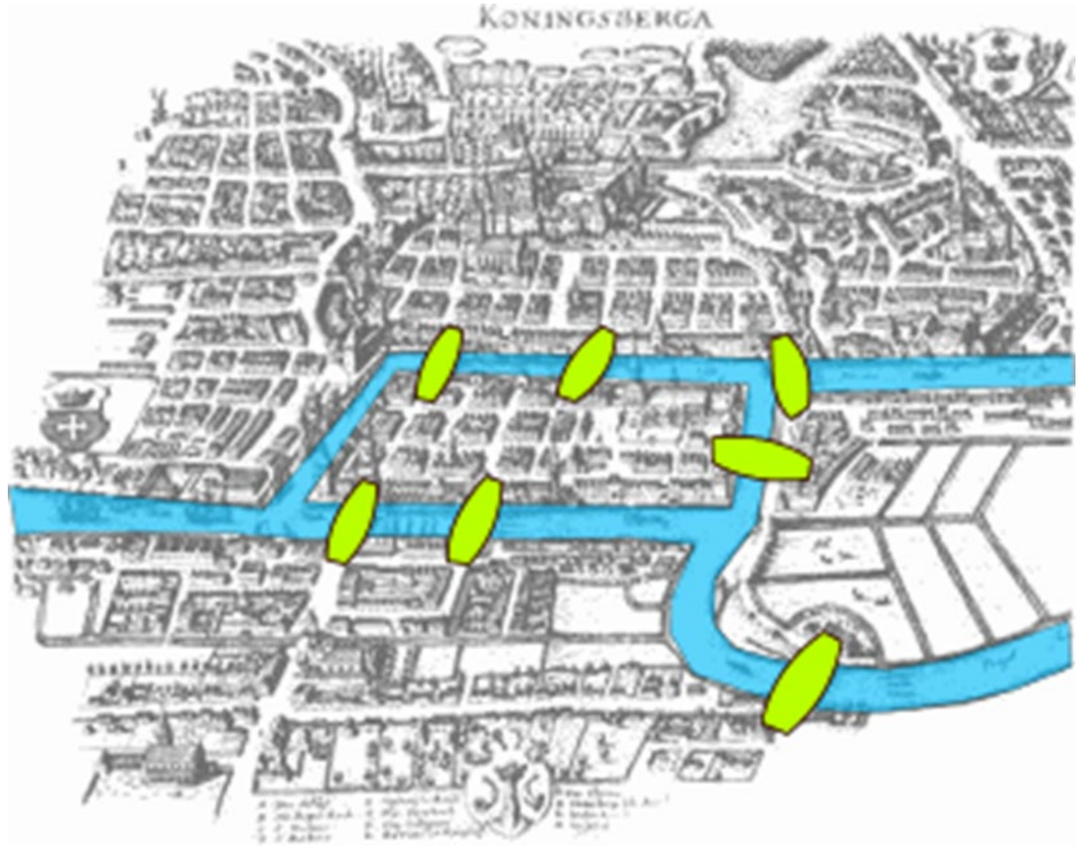
Chap IV :

Théorie des graphes

I. Exemples d’applications

I.1 Le problème des ponts de Königsberg

La ville de Königsberg en 1736 était une ville de Prusse orientale traversée par un fleuve sur lequel il y avait une île et qui se divisait en deux branches. La ville était ainsi divisée en quatre régions reliées entre elles par 7 ponts.
Le problème que se posaient les habitants de Königsberg était « Est-il possible de prendre une marche dans la ville qui nous fasse passer par chaque pont exactement une fois chacun? ».

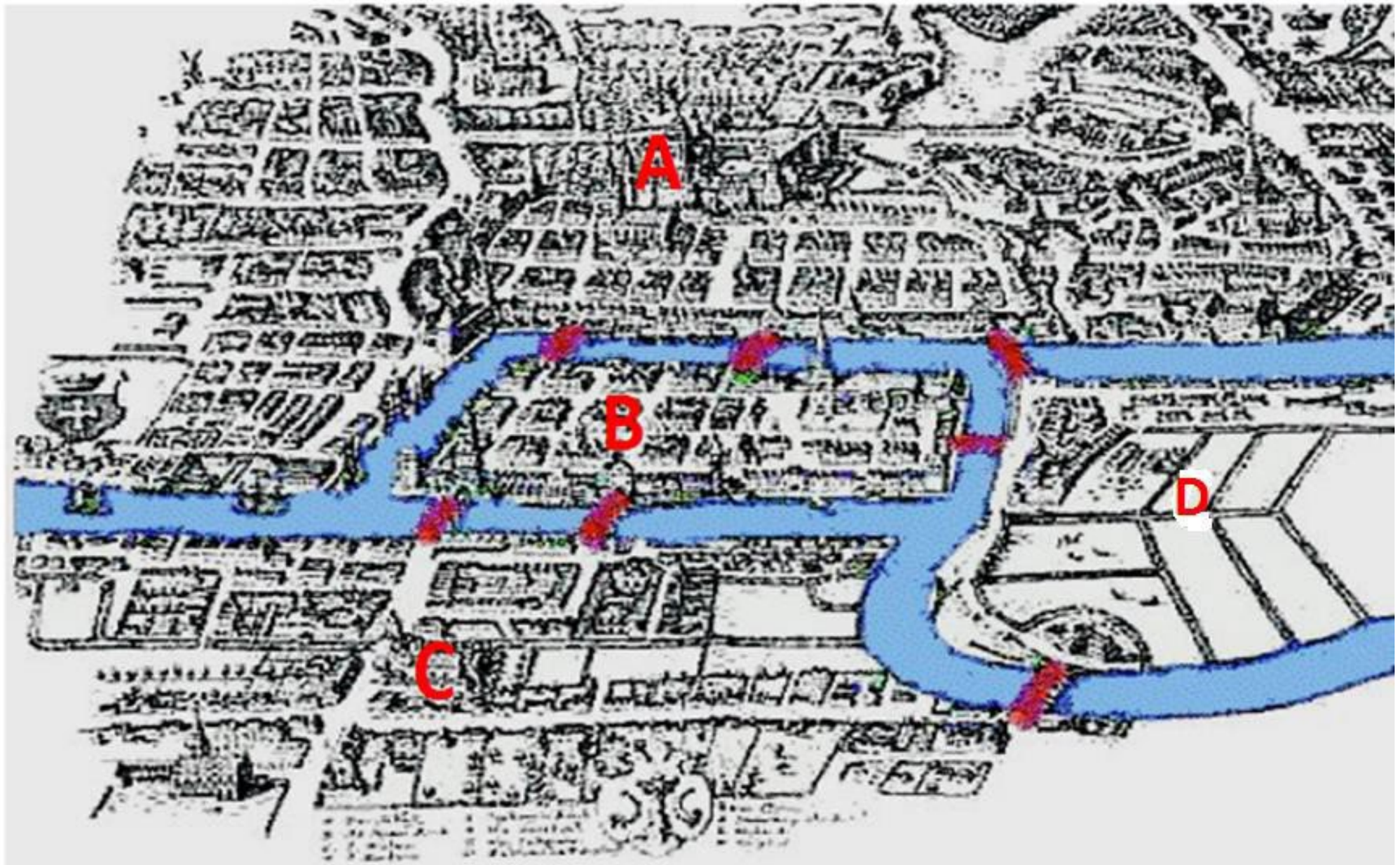


L’article d’Euler est considéré par plusieurs comme l’article fondateur de la théorie des graphes.

Un graphe est un ensemble de sommets reliés par des lignes qu’on appelle arcs.

On peut tracer un graphe qui est équivalent au schéma d’Euler, il aura quatre sommets (représentant les régions) et sept lignes (représentant les ponts).

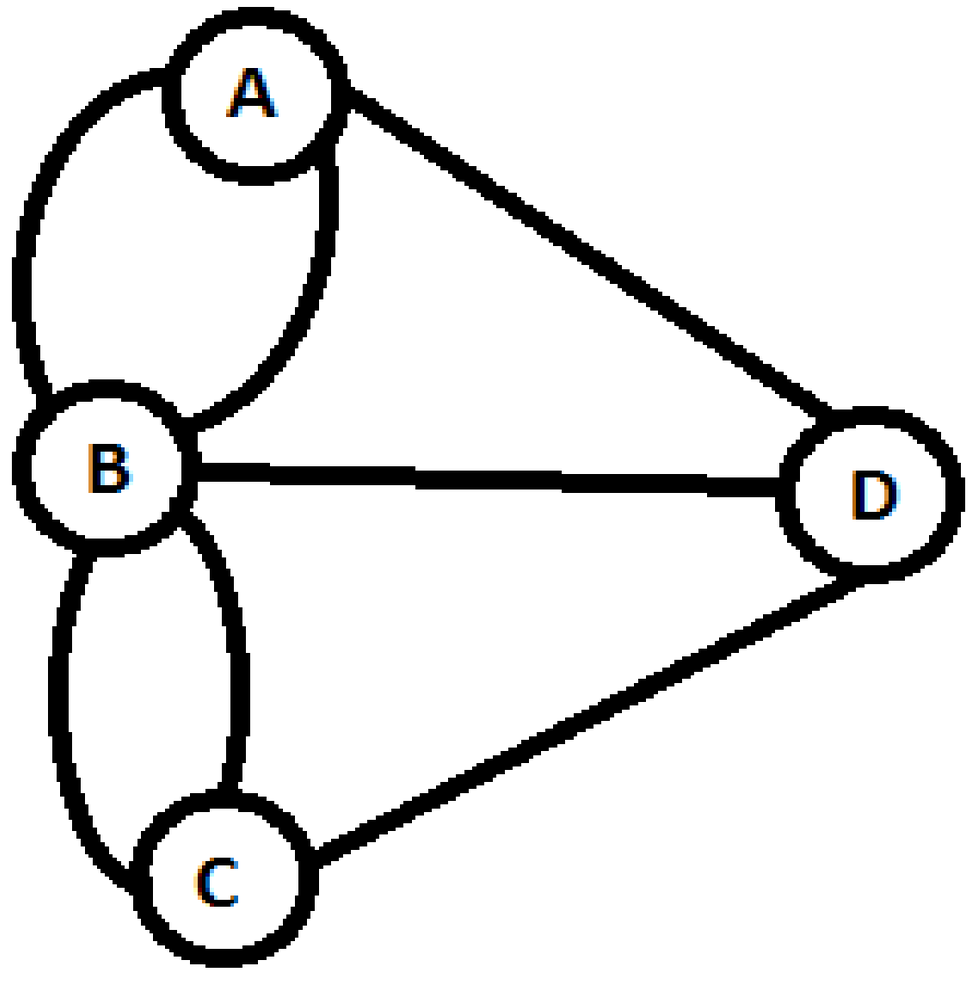
Retour à Konigsberg



Sous forme de graphe

Les sommets = quartiers

Les arcs = Les ponts



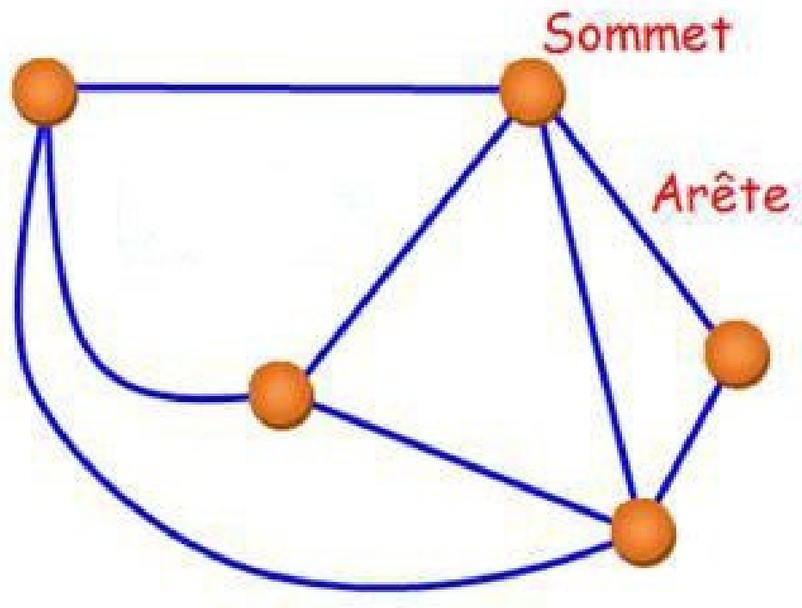
La solution moderne utilise un graphe dont les sommets représentent les régions et dont les arcs représentent les ponts.

Il suffit, comme Euler a fait de calculer le nombre de ponts touchant chaque région.
On divise alors les régions en deux catégories : les régions de passage et les régions extrémales.
Les régions extrémales sont celles où débute et où termine le parcours. Il y en aura au plus deux. Les régions de passages sont toutes les autres régions et puisqu'à chaque passage dans une région de passage on utilise deux ponts, un pour y arriver, un autre pour la quitter, ces régions toucheront toutes un nombre pairs de ponts.

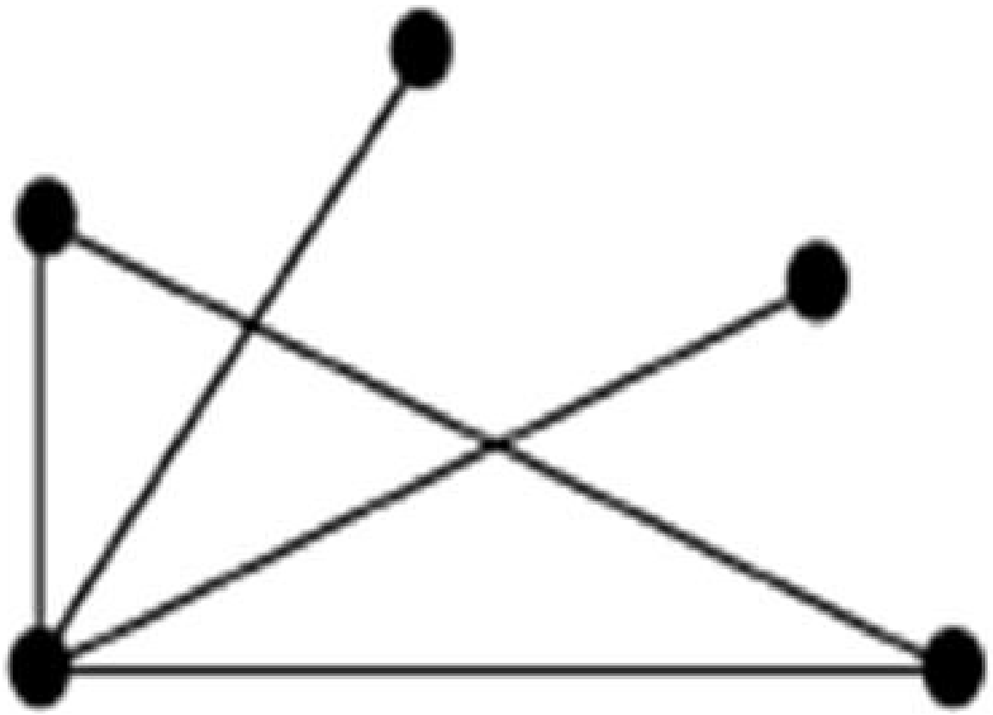
Donc, outre au plus deux régions extrémales, toutes les régions doivent toucher un nombre pair de ponts. Dans le cas du problème de Königsberg, quatre régions touchent un nombre impair de ponts, le parcours est donc impossible

I.1. Définitions et terminologie:

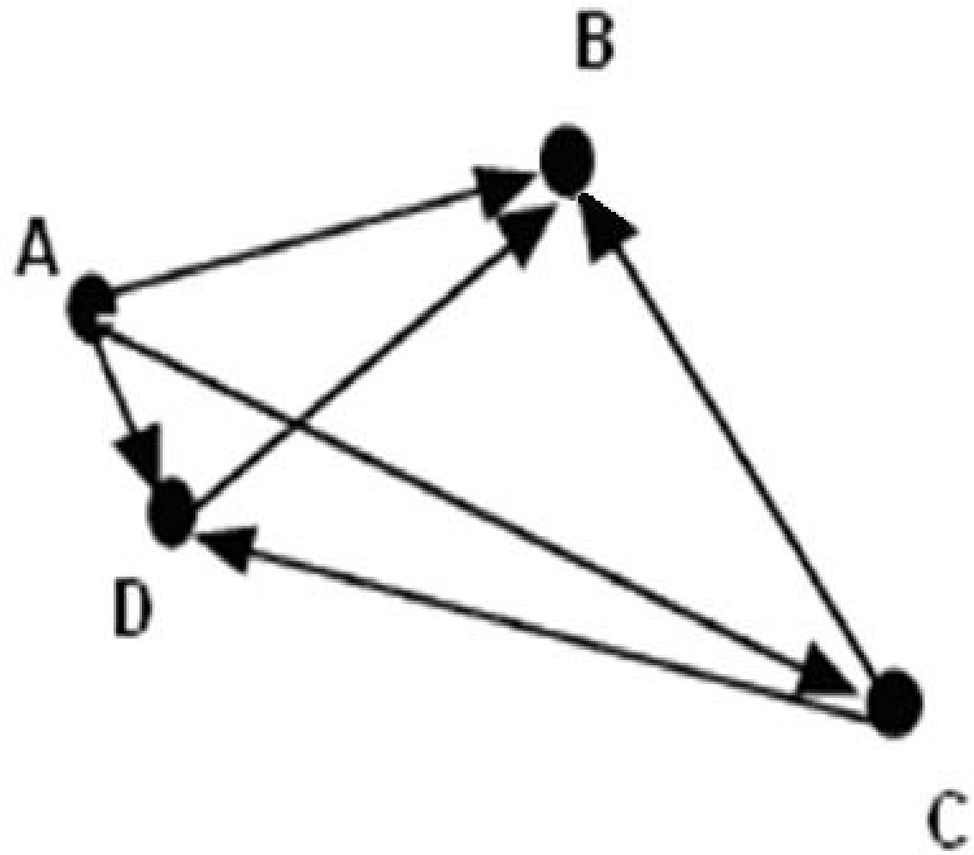
Un graphe est la donnée d'un certain nombre de points du plan, appelés ***sommets***, certains étant reliés par des segments de droites ou de courbes appelés ***arêtes***, la disposition des sommets et la forme choisie pour les arêtes n'intervenant pas.



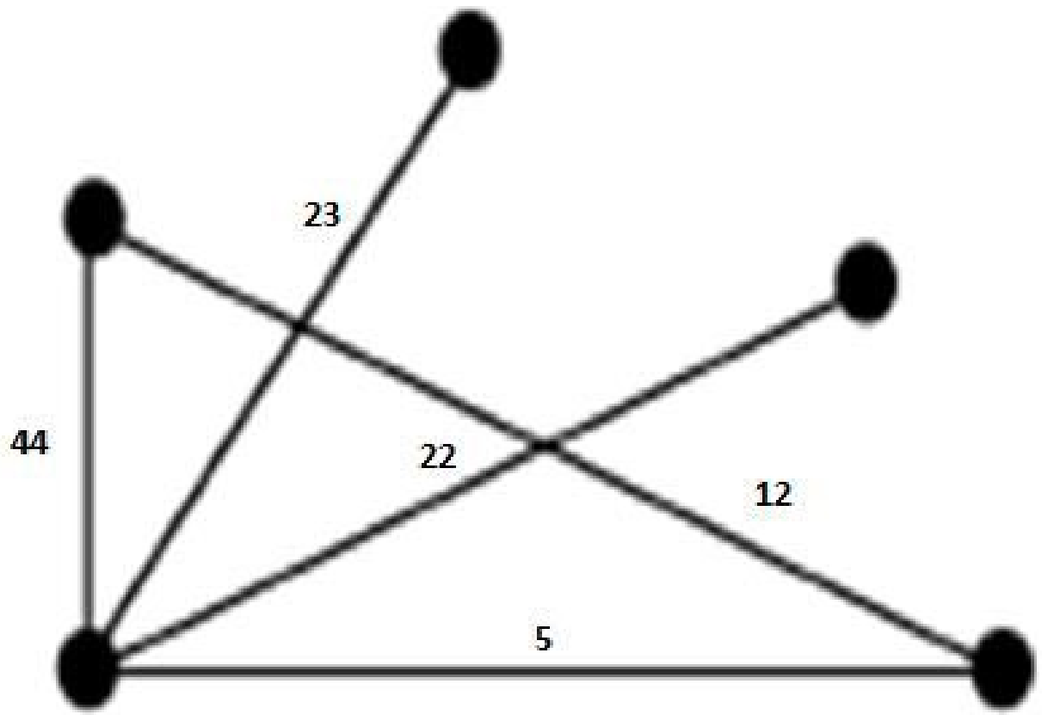
Graphe non orienté :



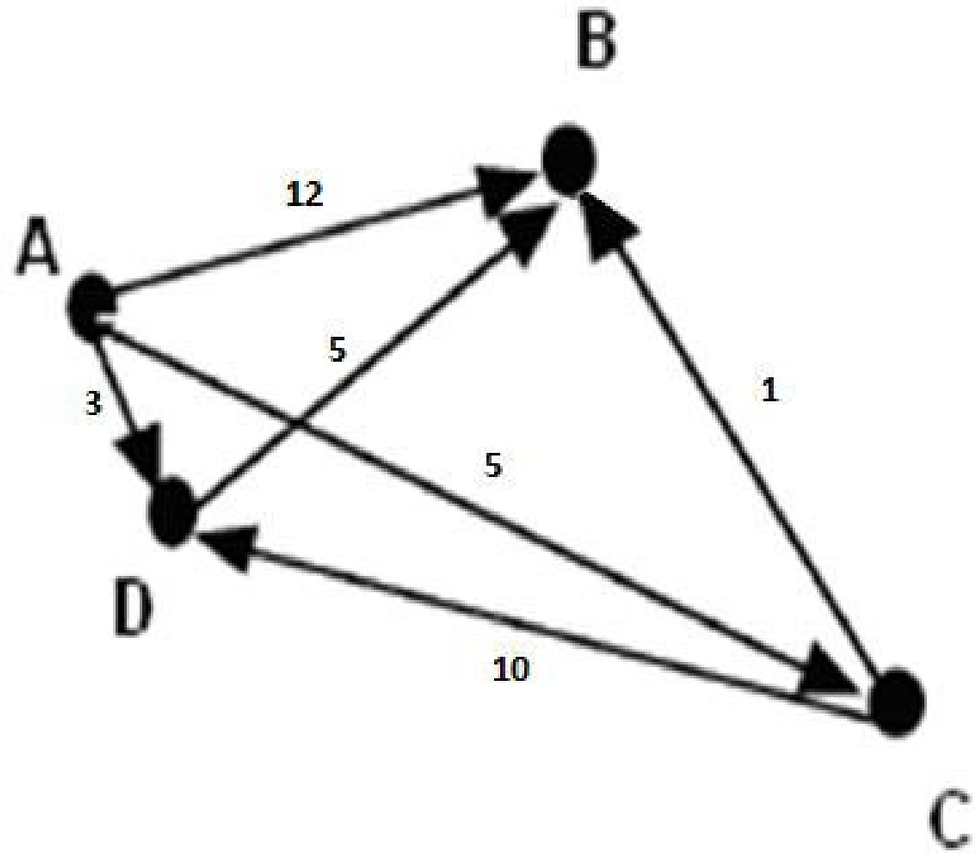
Graphe orienté :



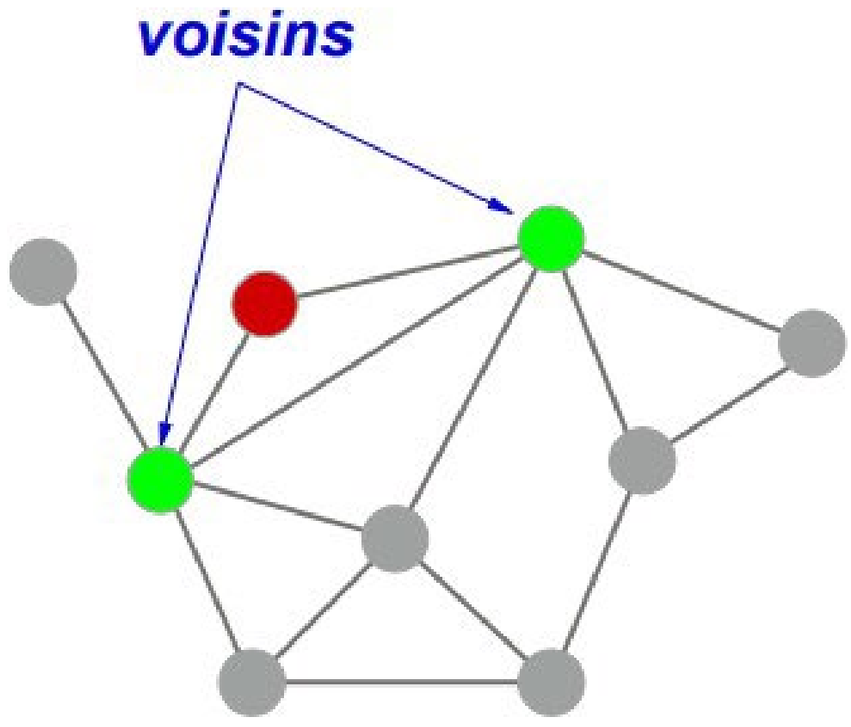
Graphe non orienté pondéré :



Graphe orienté pondéré:

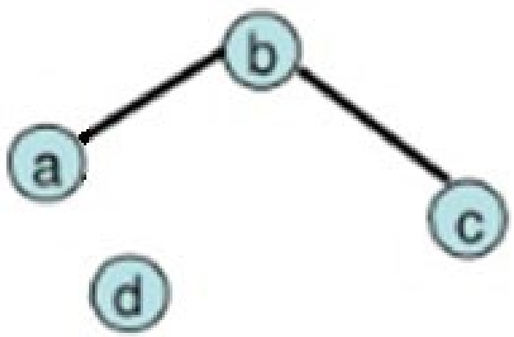


- Deux sommets d'un graphe sont dits *adjacents* s'il existe une arête (ou un arc) qui les relie.

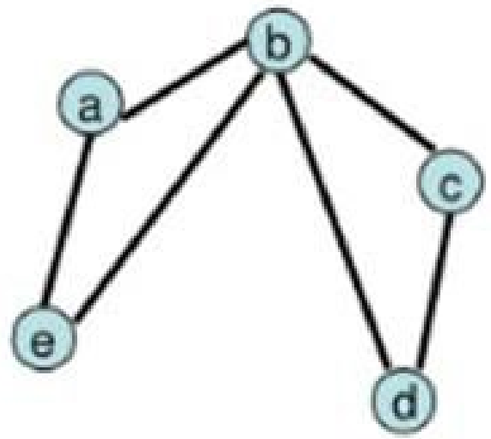


- L’ordre d’un graphe est le nombre de ses sommets.

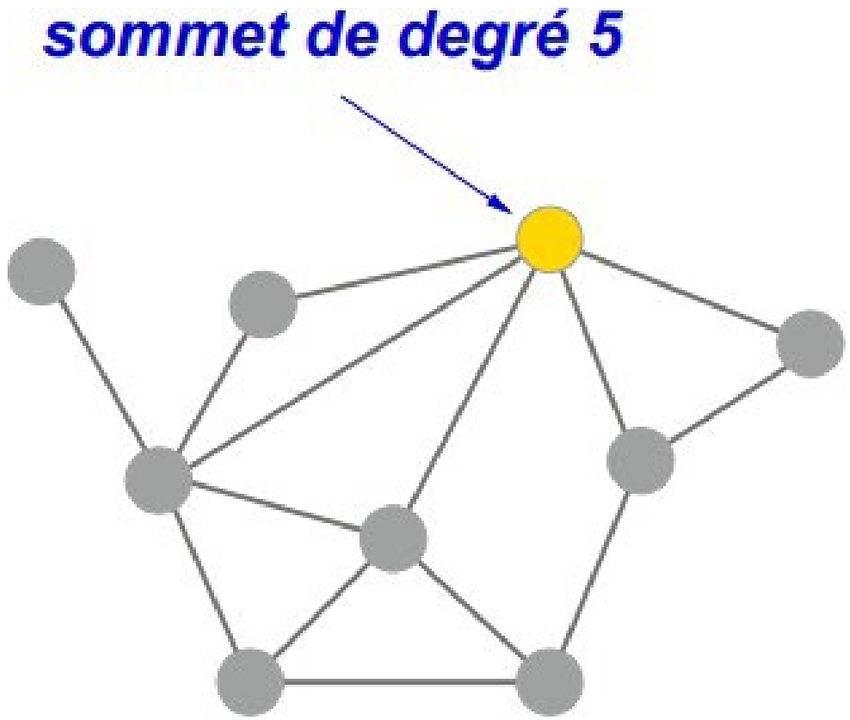
Graphe d’ordre 4



Graphe d’ordre 5

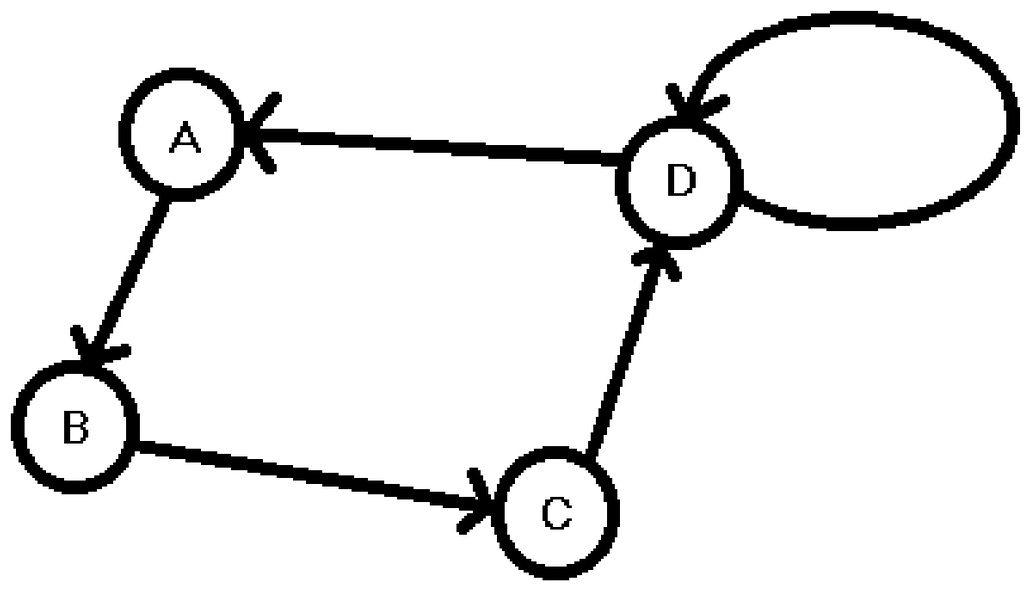


- Degré d’un sommet : nombre d’arêtes reliées à ce sommet.



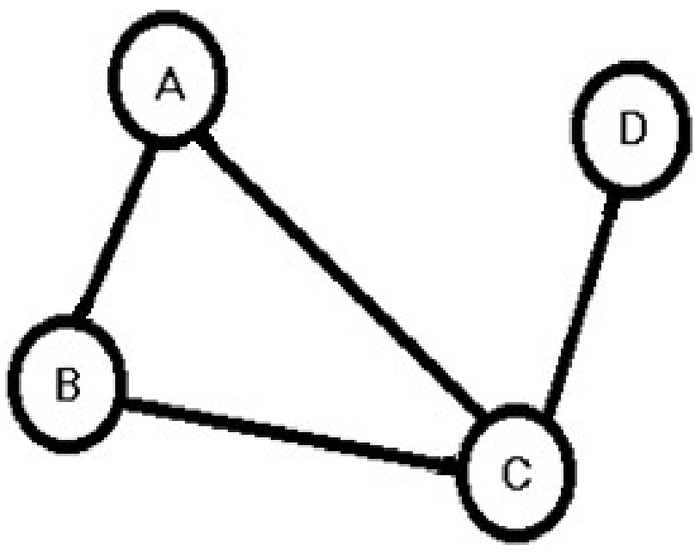
- ***Un arc boucle :*** est un arc qui part d'un sommet vers le même sommet.

L'arc <D,D> est une boucle

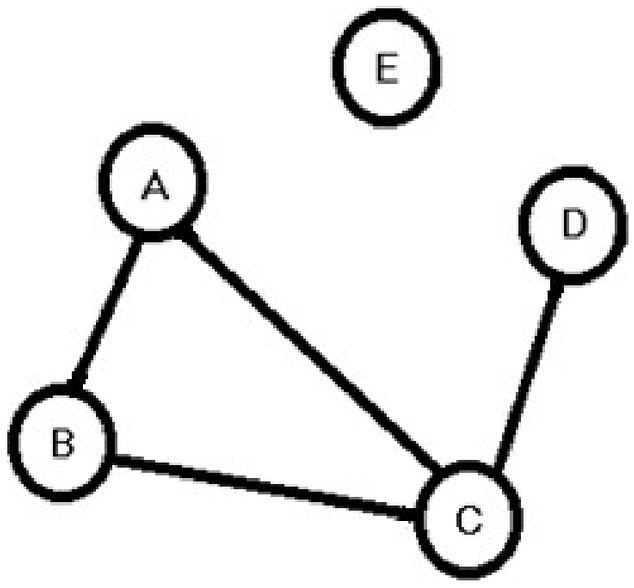


Graphe Connexe : Un graphe connexe est un graphe dont tout couple de sommets peut être relie par une chaine de longueur n>=1

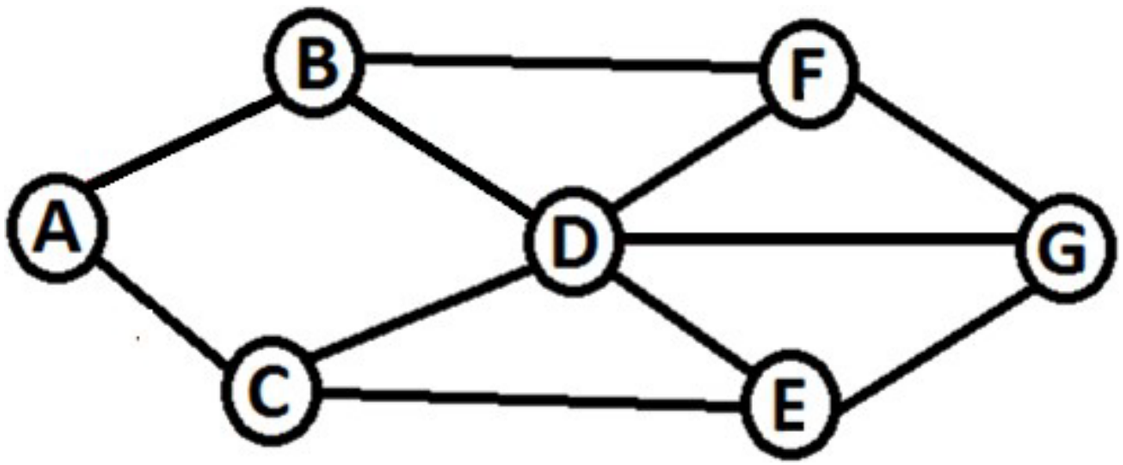
Graphe connexe :



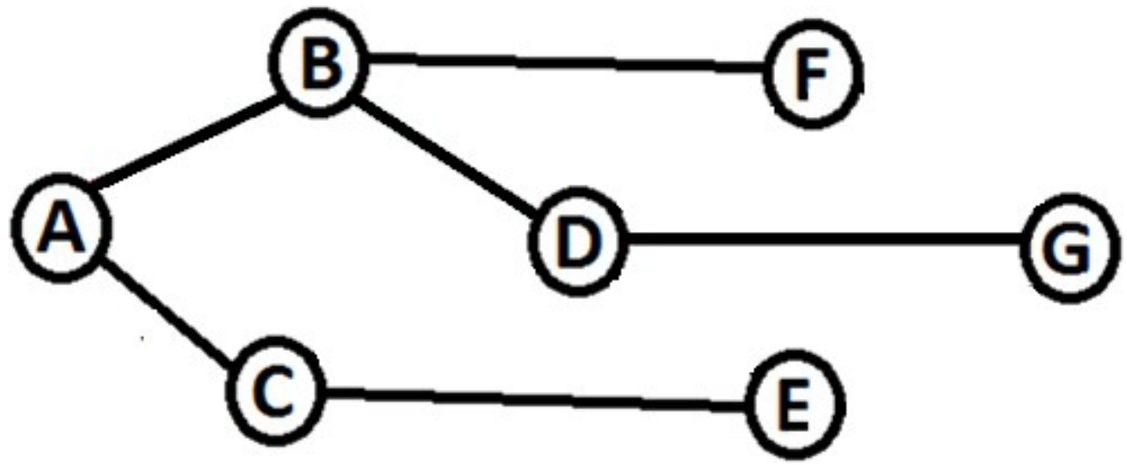
Graphe non connexe:



Un graphe est connexe s’il possède un arbre couvrant.



Graphe G



Un Arbre couvrant de G

<u>Algorithme de recherche d’un arbre minimal d’un graphe :</u>

Algorithme de Kuskal

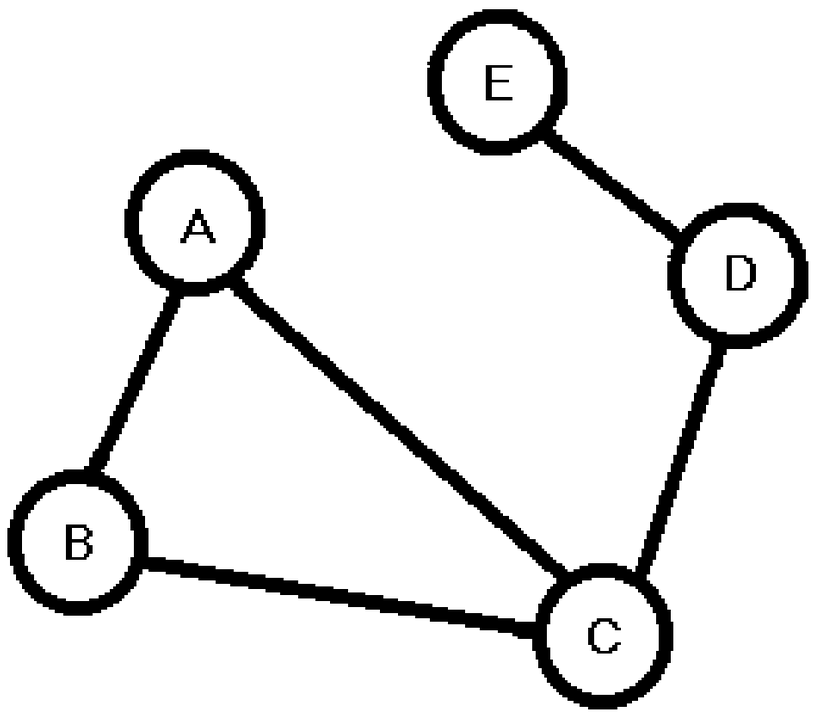
Algorithme de Prime

II. Implémentation des graphes

Deux implémentations possibles :

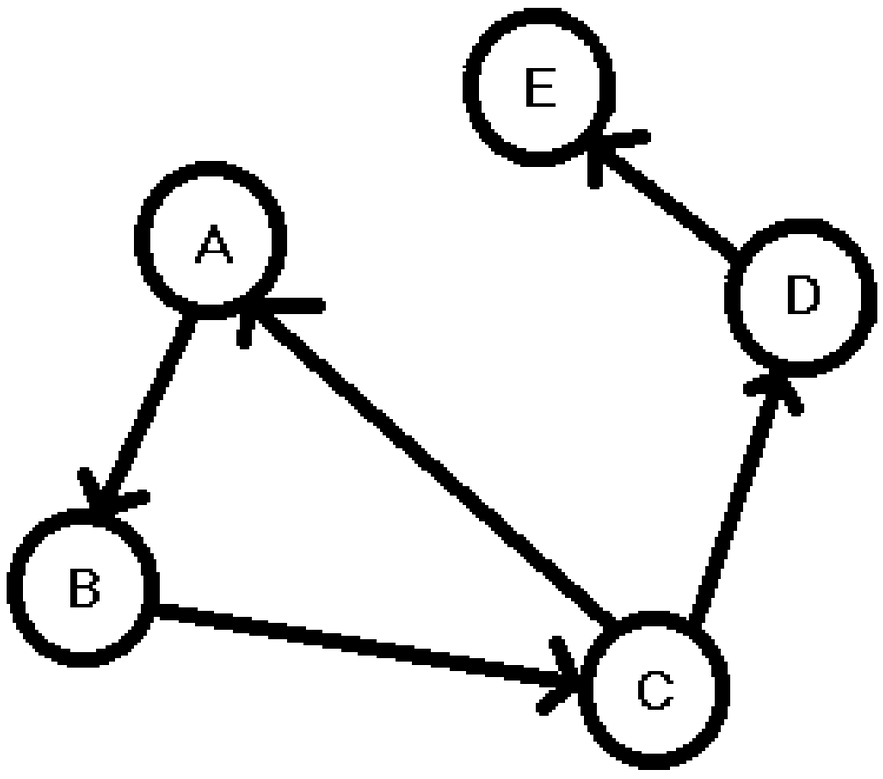
I.1. Implémentation par : Dictionnaire de précédence

Graphe non orienté :




```
G = {
        'A' : ['B', 'C'],
        'B' : ['A', 'C'],
        'C' : ['A', 'B', 'D'],
        'D' : ['C', 'E'],
        'E' : ['D']
}
```

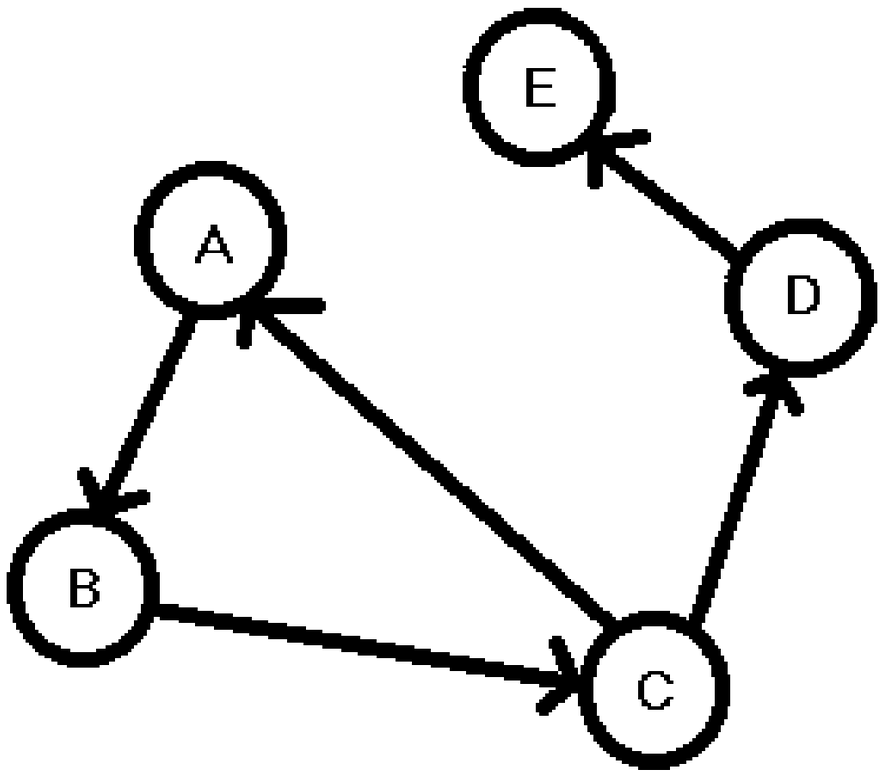
Graphe orienté :



G = {

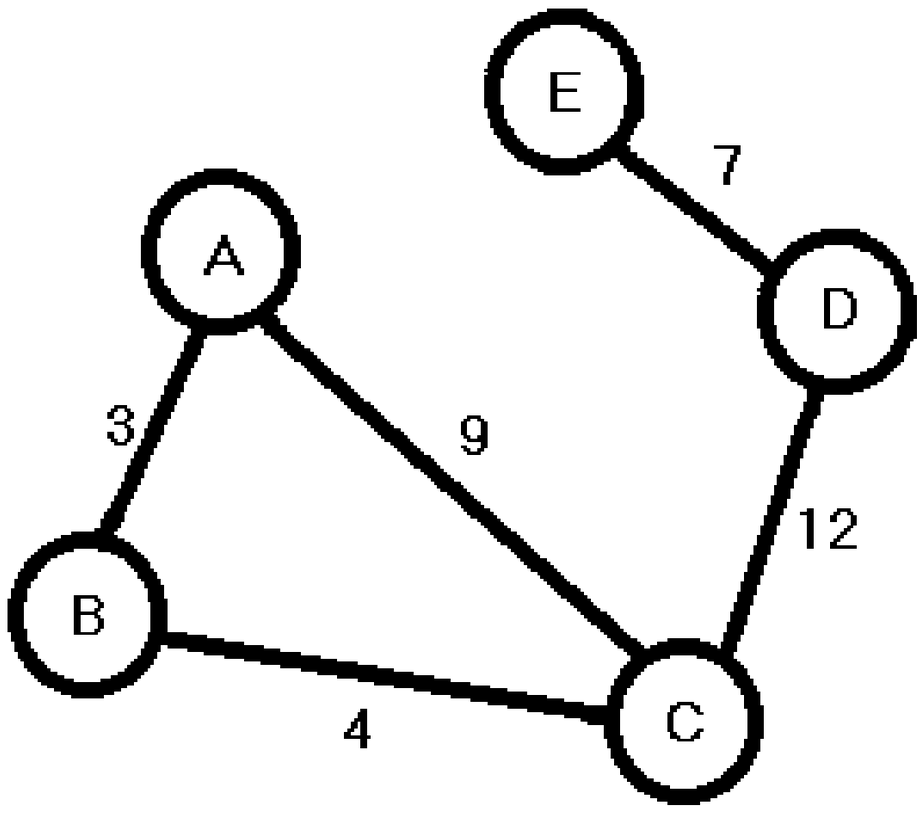
}

Graphe orienté :



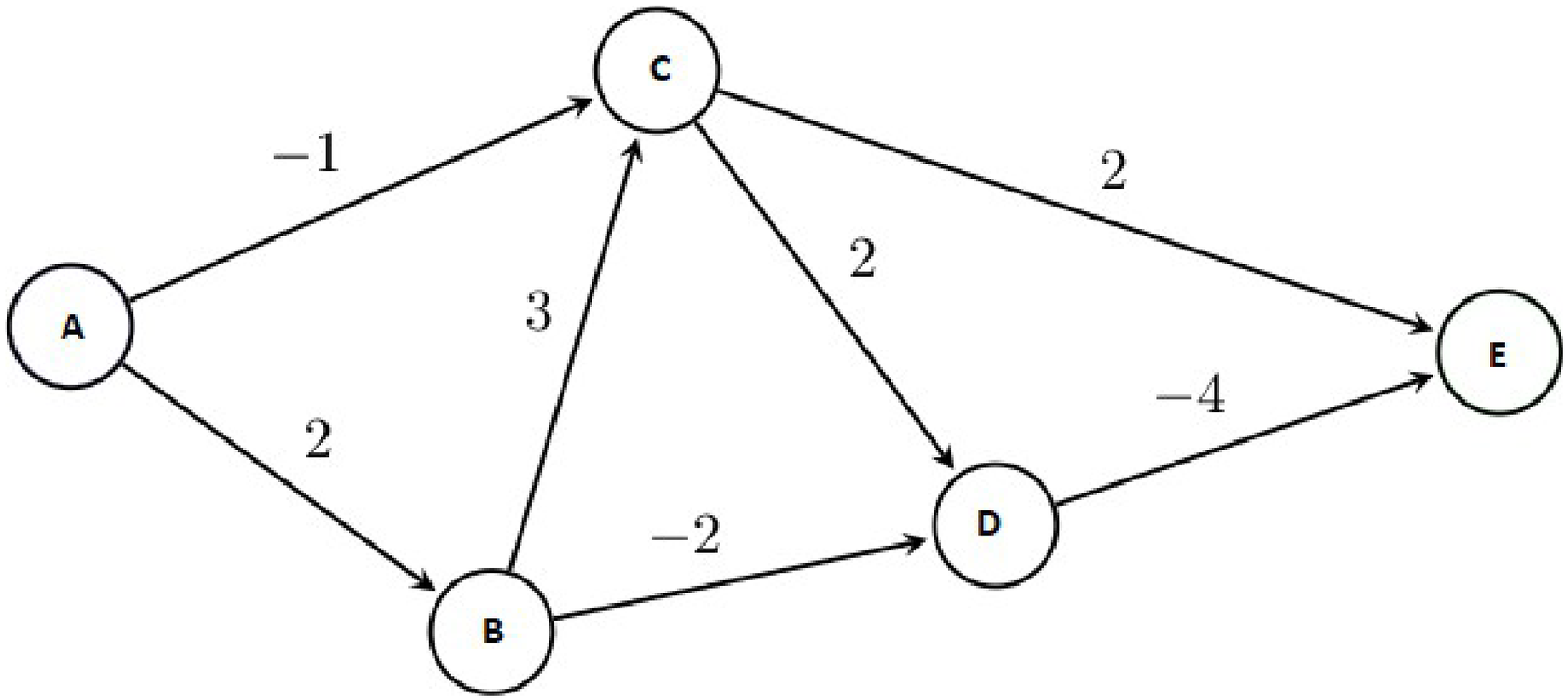

```
G = {
        'A' : ['B'],
        'B' : ['C'],
        'C' : ['A', 'D'],
        'D' : ['E'],
        'E' : []
}
```

Graphe non orienté et valué :

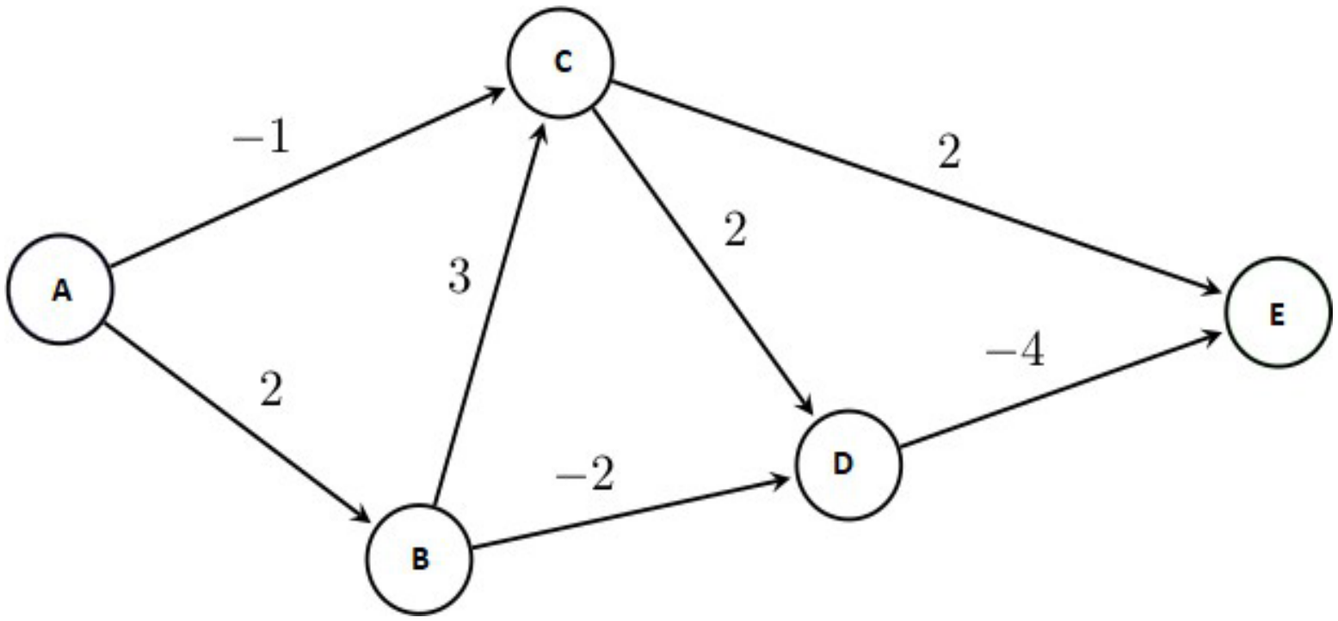



```
G = {
    'A' : [(3,'B'), (9,'C')],
    'B' : [(3,'A'), (4,'C')],
    'C' : [(9,'A'), (4,'B'), (12,’D')],
    'D' : [(7,’E'), (12,'C')],
    'E' : [(7,'D')]
}
```

Exercice : Donner l'implémentation avec le dictionnaire de précédence du graphe suivant.



Graphe non orienté et valué :



G = {

'A' : [(2,'B'), (-1,'C')],

'B' : [(3,'C’),(-2,D)],

'C' : [(2,'D'), (2,'E')],

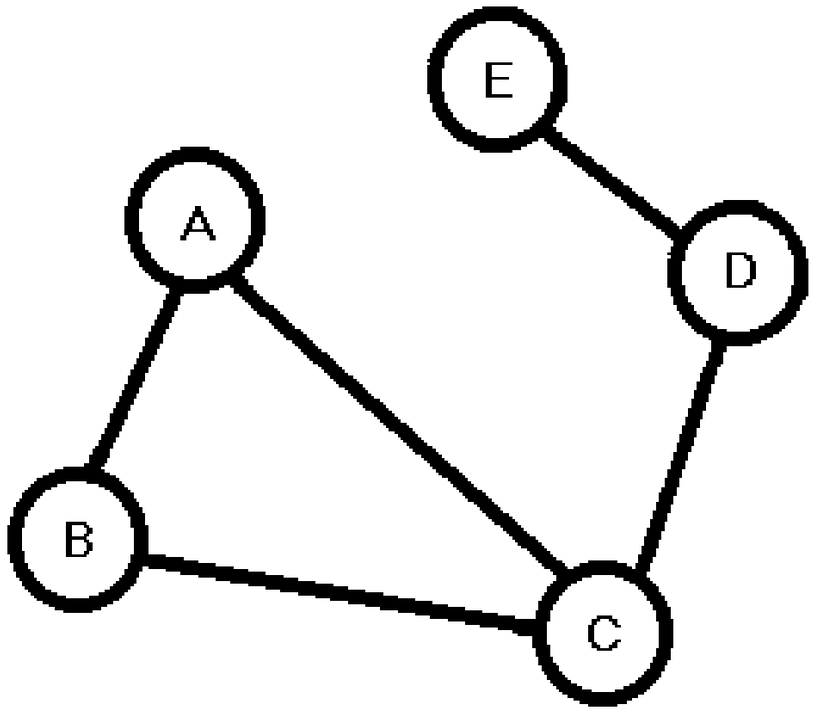
'D' : [(-4,’E')],

'E' : []

}

II.2. Implémentation par : Matrice d'adjacence

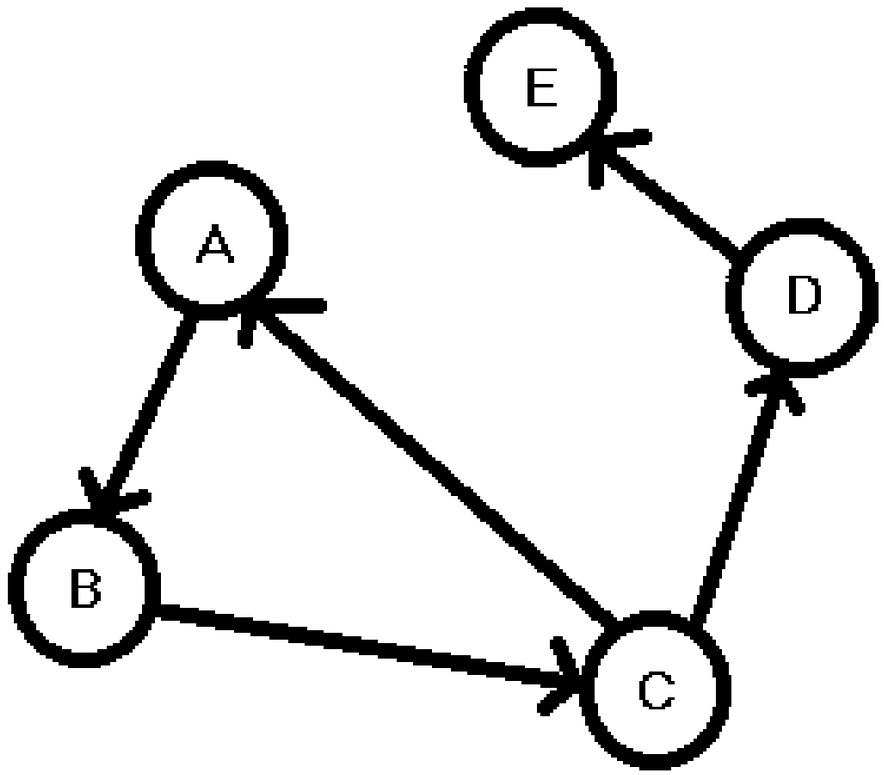
Graphe non orienté :



	A	B	C	D	E
A	0	**1**	**1**	0	0
B	**1**	0	**1**	0	0
C	**1**	**1**	0	**1**	0
D	0	0	**1**	0	**1**
E	0	0	0	**1**	0

Implémentation par :

Graphe orienté :

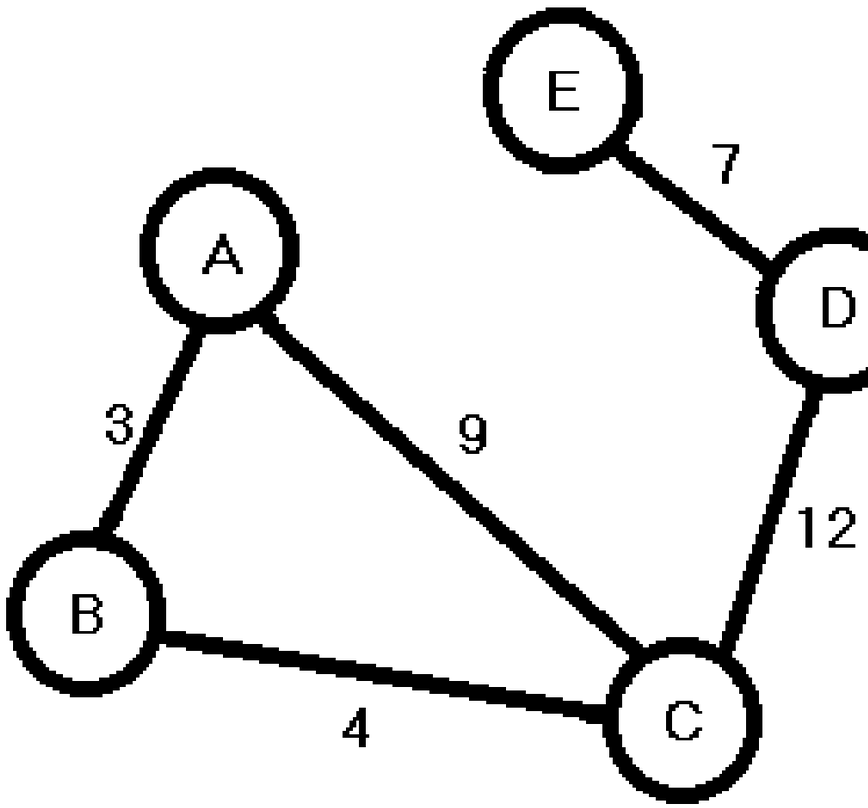


Matrice d'adjacence

	A	B	C	D	E
A	0	**1**	0	0	0
B	0	0	**1**	0	0
C	**1**	0	0	**1**	0
D	0	0	0	0	**1**
E	0	0	0	0	0

Exemple:

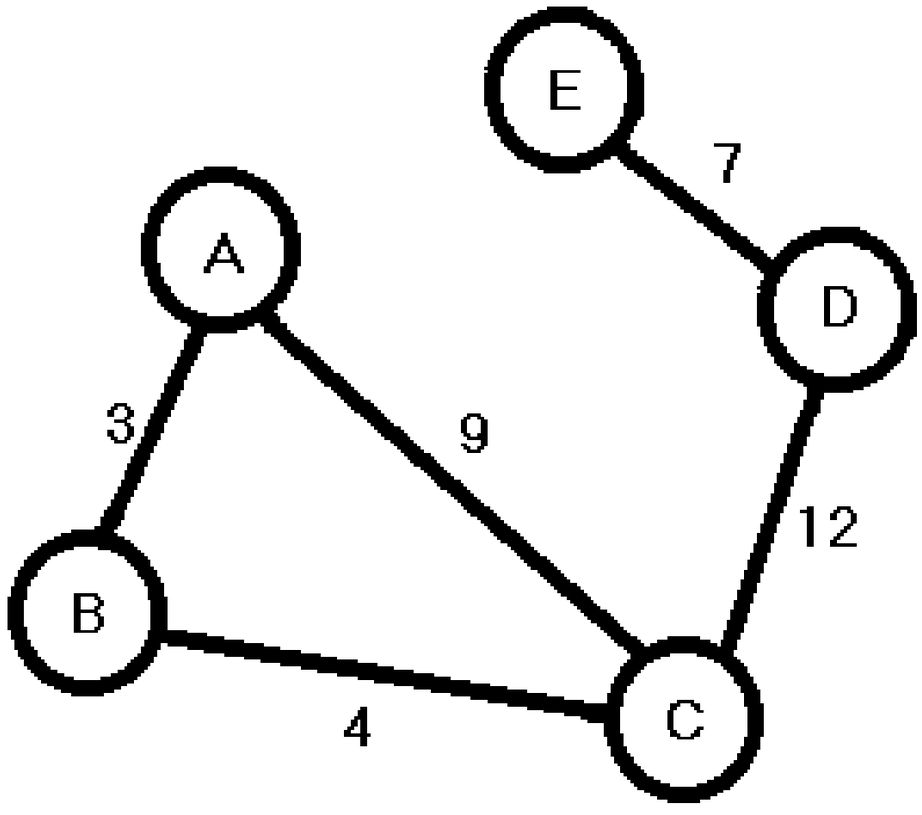
Graphe non orienté et valué :



Implémentation par :

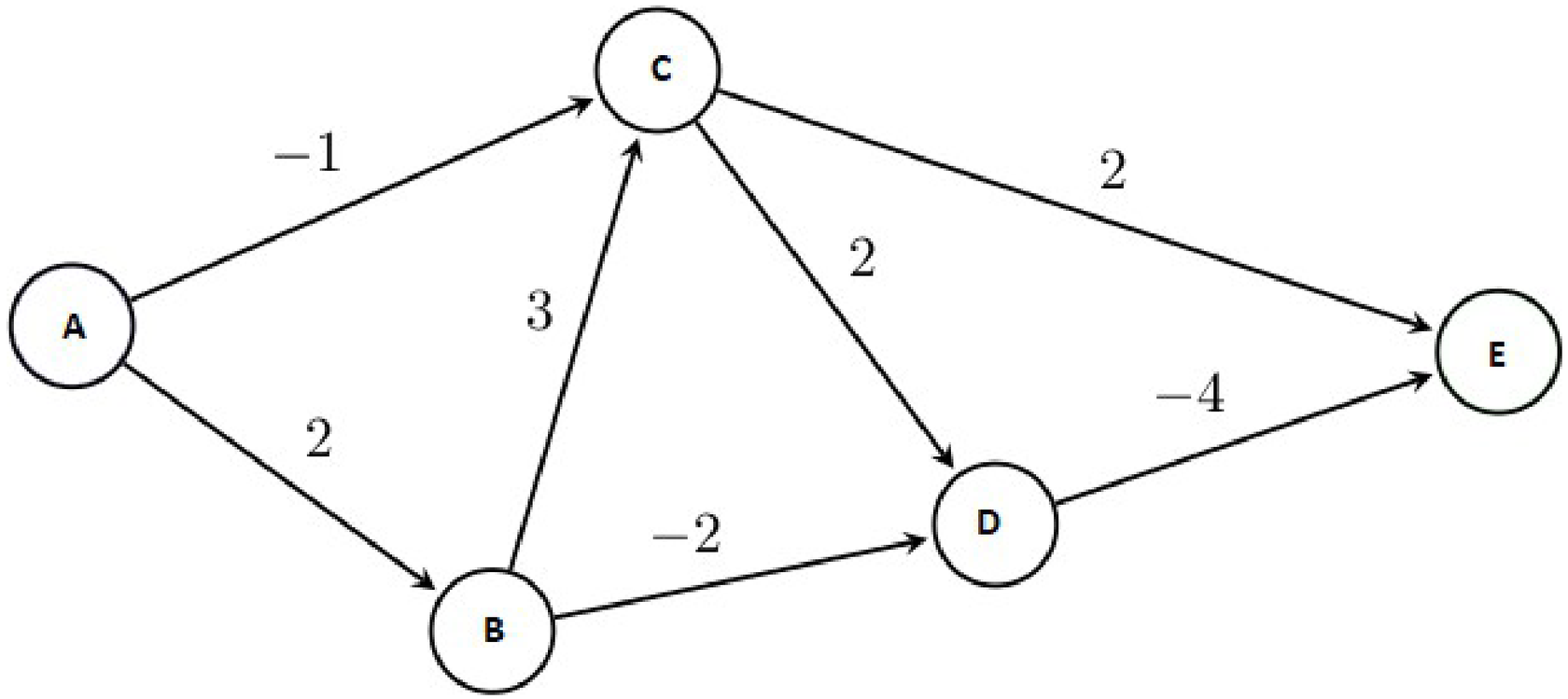
Matrice d'adjacence

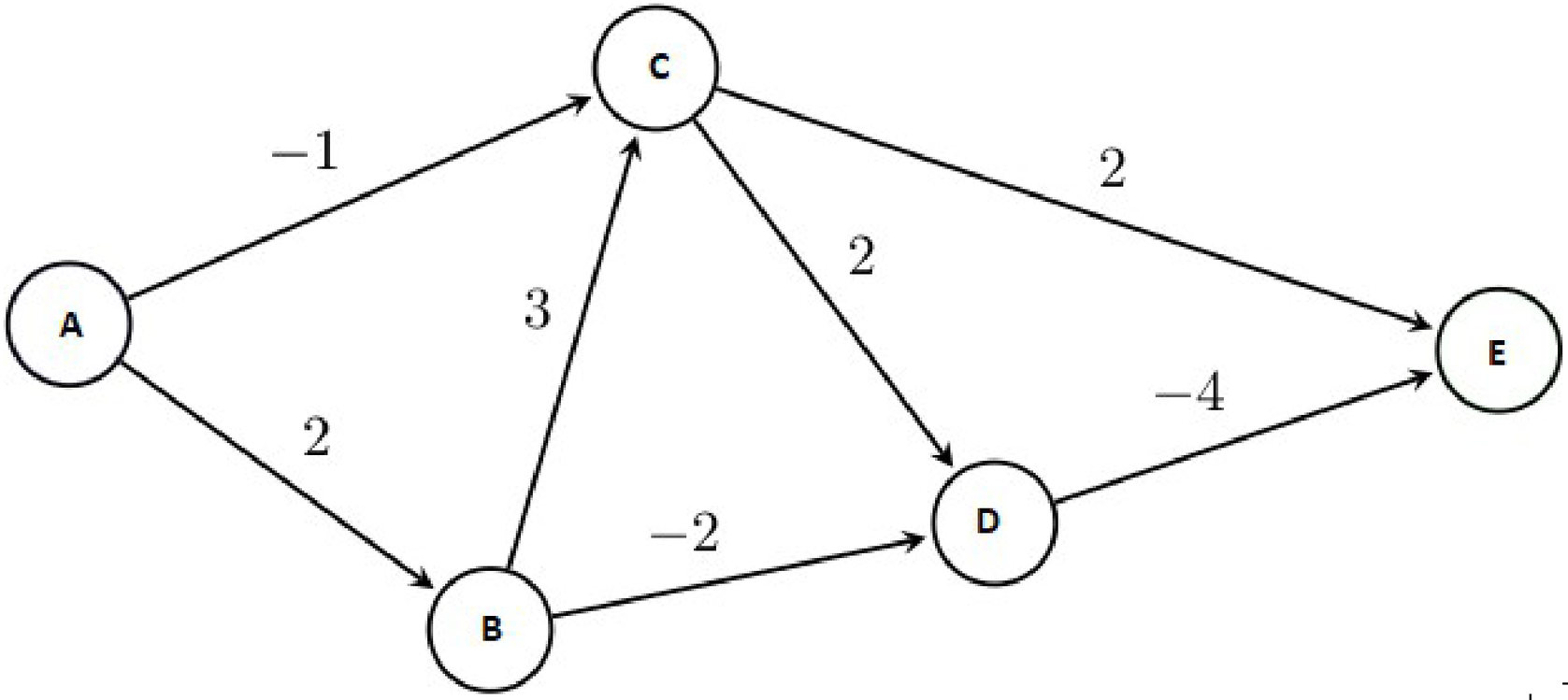
Graphe non orienté et valué :



	A	**B**	**C**	**D**	**E**
A	0	**3**	**9**	0	0
B	**3**	0	**4**	0	0
C	**9**	**4**	0	**12**	0
D	0	0	**12**	0	**7**
E	0	0	0	**7**	0

Exercice : Donner l'implémentation avec la matrice d'adjacence du graphe suivant.





	A	B	C	D	E
A	0			0	0
B		0		0	0
C			0		0
D	0	0		0	
E	0	0	0		0

III. Algorithmes sur les graphes

III.1. Parcours en largeur d'un graphe:

Ce parcours peut être utilisé pour :

- Chemin plus court entre deux sommets
- Vérifier si un graphe est connexe

Algorithme :

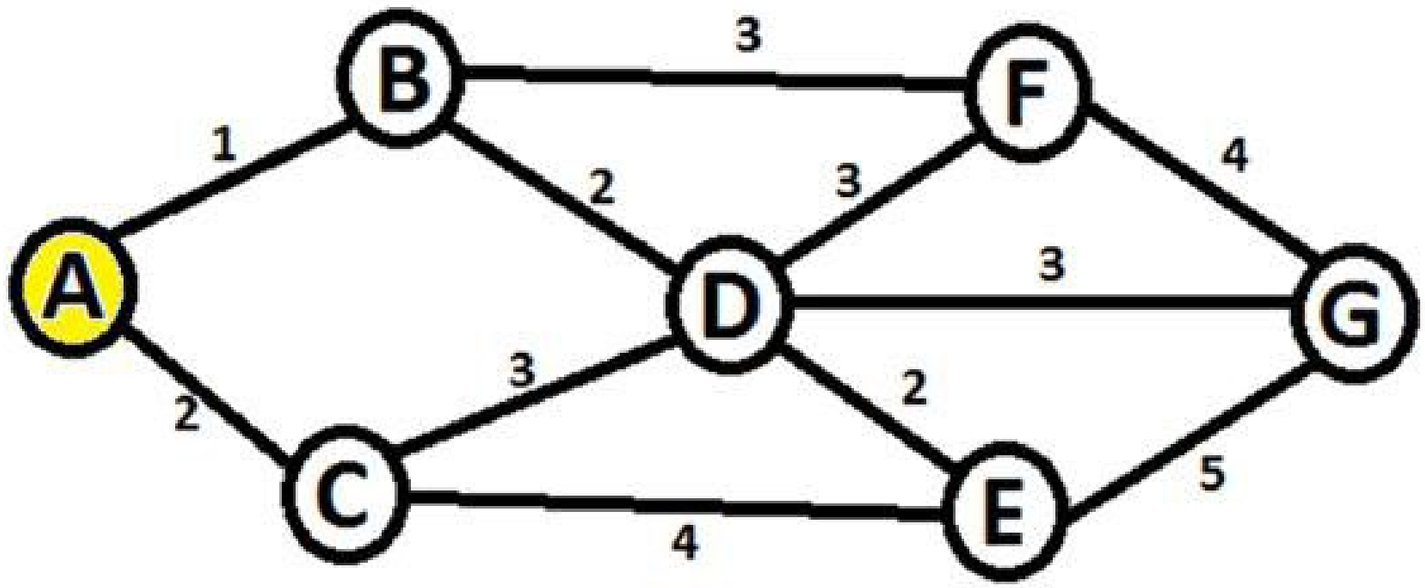
1 S est le sommet de départ

2 Cet algorithme liste d'abord les voisins de S pour ensuite les explorer un par un.

2- Répétez (1) pour chaque voisin.

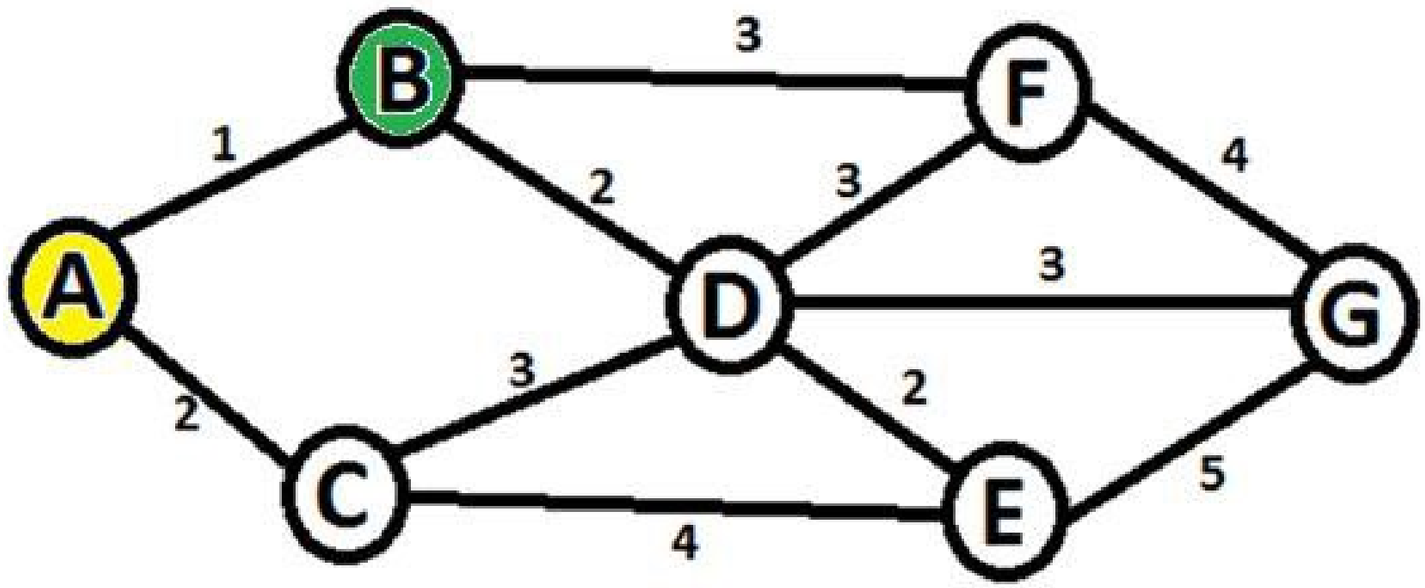
Parcours en largeur d'un

graphe : Exemple



Liste des sommets visités : A,

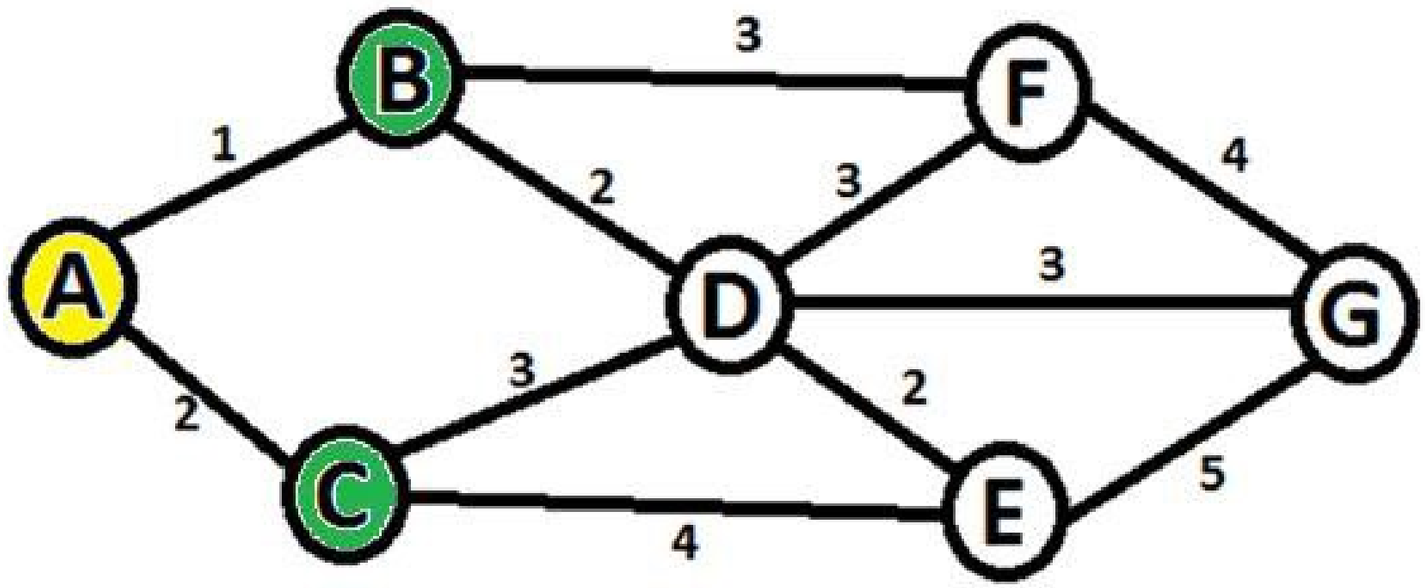
Parcours en largeur d'un graphe : Exemple



Liste des sommets visités : A, B

Parcours en largeur d'un

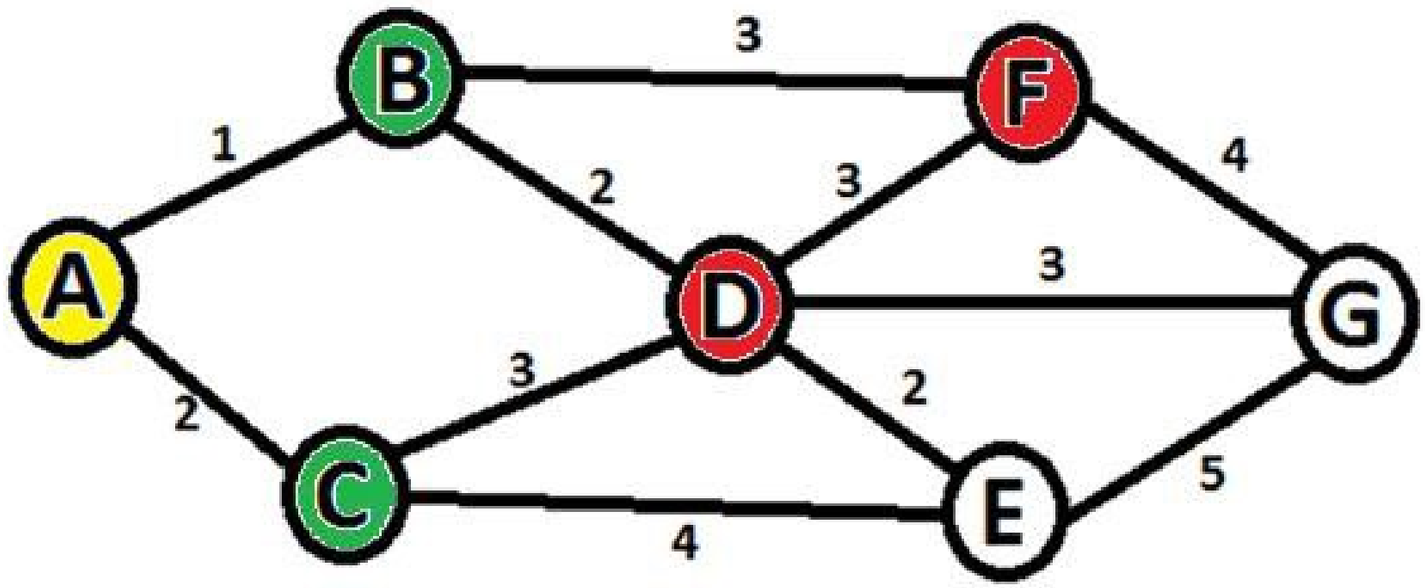
graphe : Exemple



Liste des sommets visités : A, B, C

Parcours en largeur d'un

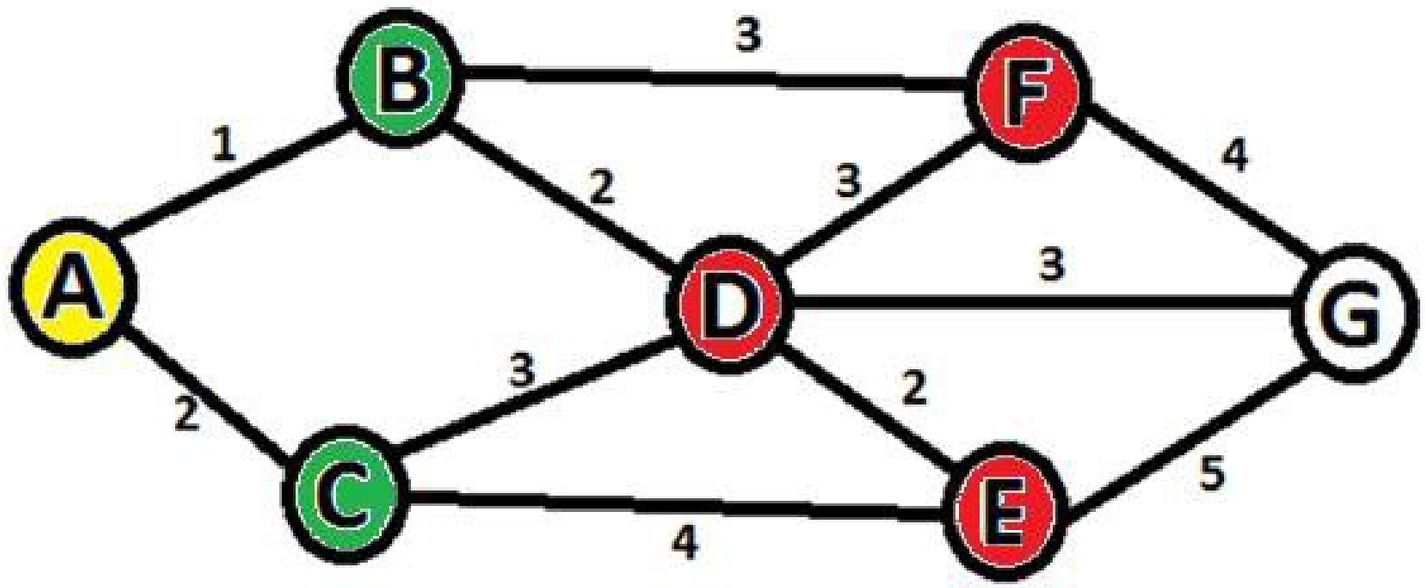
graphe : Exemple



Liste des sommets visités : A, B, C, F, D

Parcours en largeur d'un

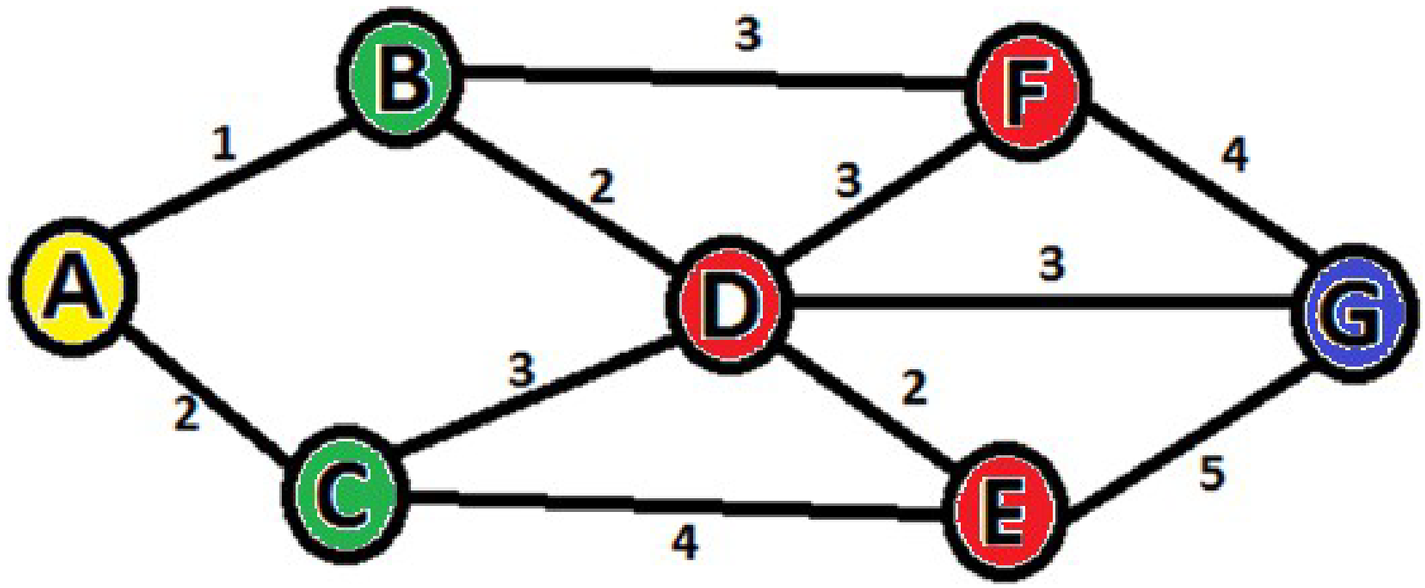
graphe : Exemple



Liste des sommets visités : A, B, C, F, D, E

Parcours en largeur d'un

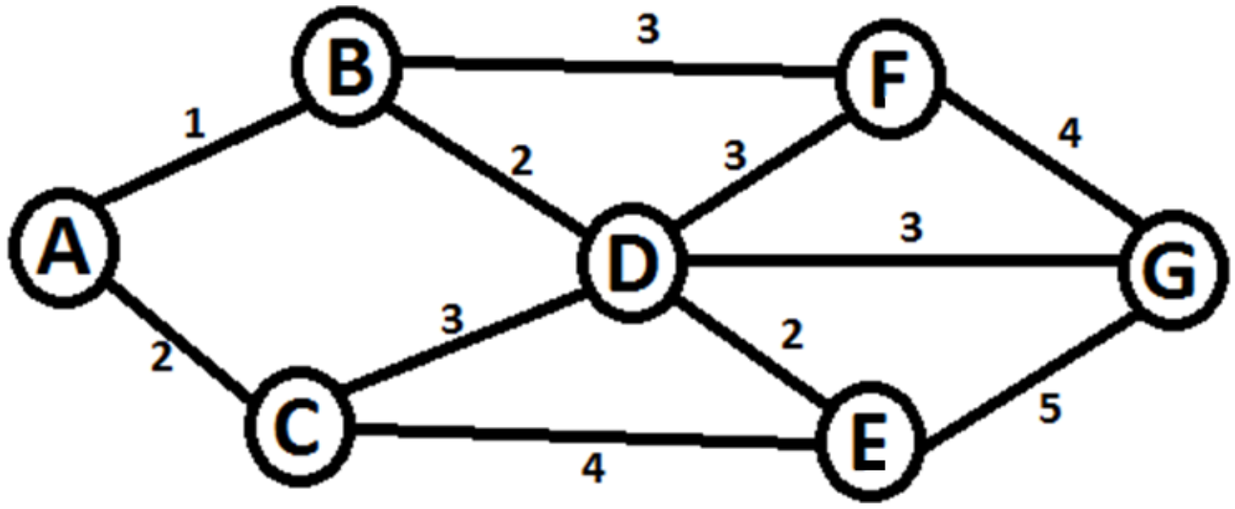
graphe : Exemple



Liste des sommets visités : A, B, C, F, D, E, G

Ecrire la fonction « parcours_largeur(G, d) » qui fait le parcours en largeur du graphe **G** passé en paramètre à partir du sommet de départ **d**.

On utilisera la représentation par dictionnaire de précédence d'un graphe.



Le résultat du parcours en largeur du graphe en dessus à partir de A sera : A, B, C, F, D, E, G

Solution

```
def parcours_largeur(G, d):
    visited = []
    file_to_visite = [d]
    while file_to_visite!=[]:
        s = file_to_visite.pop(0)
        if s in visited :
            continue
        visited.append(s)
        voisins = G[s]
        for dv, v in voisins:
            if v not in visited :
                file_to_visite.append(v)
return visited
```

III.2. Parcours en profondeur d'un graphe

Ce parcours peut être utilisé pour :

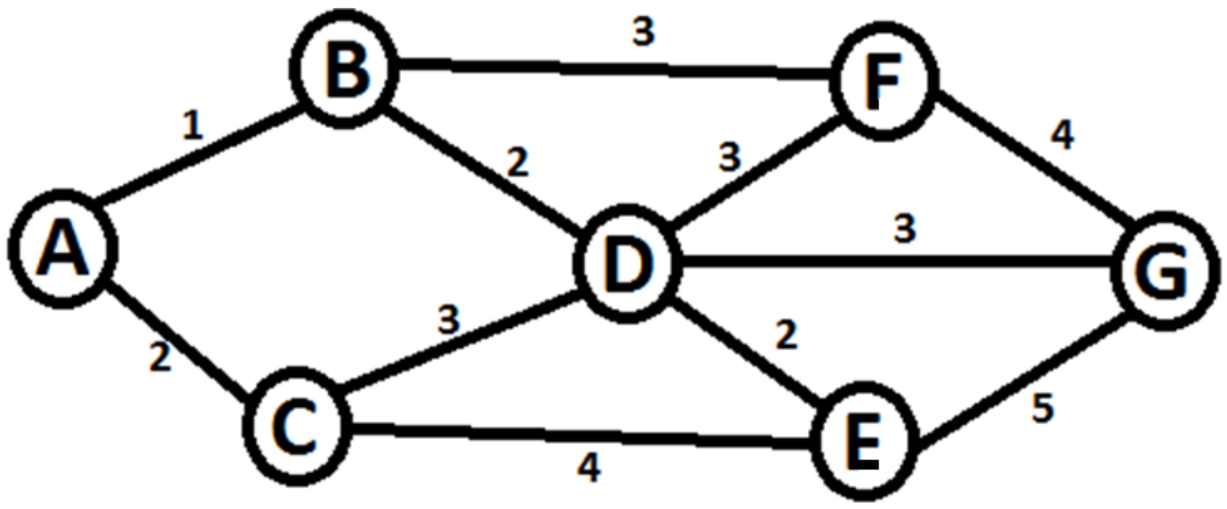
- Chemin plus court entre deux sommets
- Vérifier si un graphe est connexe

Algorithme :

Recherche qui progresse à partir d'un sommet S en s'appelant récursivement pour chaque sommet voisin de S :
Pour chaque sommet, il prend le premier sommet voisin jusqu'à ce qu'un sommet n'aie plus de voisins (ou que tous ses voisins soient marqués), et revient alors au sommet père.

Ecrire la fonction « parcours_profondeur(G, d) » qui fait le parcours en profondeur du graphe **G** passé en paramètre à partir du sommet de départ **d**.

On utilisera la représentation par dictionnaire de précédence d'un graphe.



Le résultat du parcours en profondeur du graphe en dessus à partir de A sera : A, B, F ,G ,D, E, C

Solution

```
def  parcours_profondeur(G, d, visité):
    """  """

    visité.append(d)
    voisins = G.voisins(d)

    for poids, v in voisins :
        if v not in visité :
            parcours_profondeur(G, v, visité)
```

Pour tester :

```
L=[]
parcours_profondeur(GRAPHE, 'A', L)
print(L)
```

```
def  parcours_profondeur(G, d, visité):
    """  """

    visité.append(d)
voisins = G.voisins(d)

for poids, v in voisins :
    if v not in visité :
        parcours_profondeur(G, v, visité)
```

III.3. Algorithme de Dijkstra

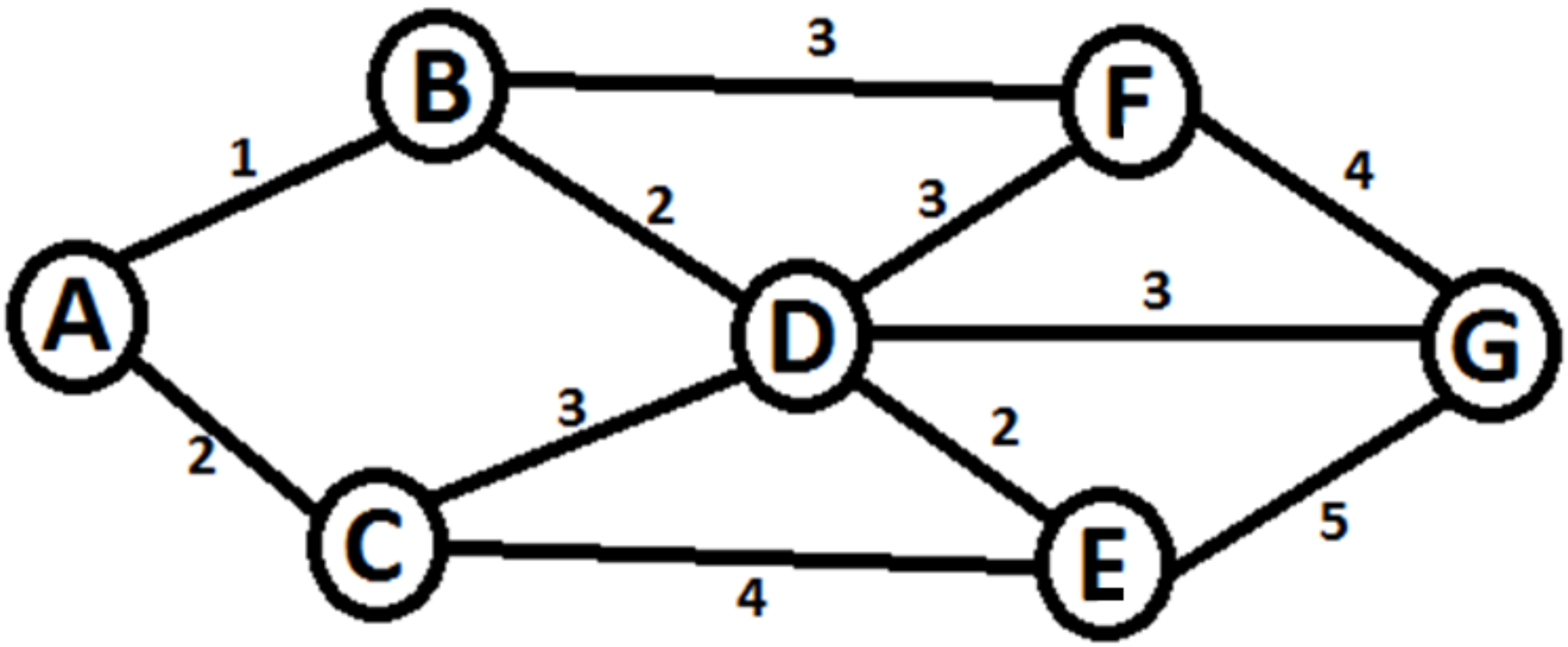
En théorie des graphes, l'**algorithme de Dijkstra** sert à résoudre le problème du plus court chemin.

Il s'applique à un graphe connexe dont le poids lié aux arêtes est un réel positif.

Applications :

1. Chemin le plus court entre deux villes
2. Chemin le plus rapide entre deux poins en ville.
3. Routage d’information optimal
4. ….

Exemple : Trouver le chemin optimal entre A et G



Algorithme de Dijkstra

Initialisation de l'algorithme :

Étape 1 : On affecte le poids 0 au sommet origine (E) et on attribue provisoirement un poids ∞ aux autres sommets.

Algorithme de Dijkstra

Répéter les opérations suivantes de l’étape 2 et 3 tant que le sommet de sortie (s) n'est pas affecté d'un poids définitif :

Étape 2 : Parmi les sommets dont le poids n'est pas définitivement fixé choisir le sommet X de poids *p* minimal. Marquer définitivement ce sommet X affecté du poids p(X).

Étape 3 : Pour tous les sommets Y qui ne sont pas définitivement marqués, adjacents au dernier sommet fixé **X** :

Calculer la somme *s* du poids de X et du poids de l'arête reliant X à Y.

Si la somme *s* est inférieure au poids provisoirement affecté au sommet Y, affecter provisoirement à Y le nouveau poids *s* et indiquer entre parenthèses le sommet X pour se souvenir de sa provenance.

Quand le sommet s est définitivement marqué

Le plus court chemin de E à S s'obtient en écrivant de gauche à droite le parcours en partant de la fin S.

Algorithme de Dijkstra

Etapes de l’algorithme :

	A	B	C	D	E	F	G
Etape 1							
Etape 2							
Etape 3							
Etape 4							
Etape 5							
Etape 6							
Etape 7							

Règles pour remplir les cases de chaque cellule:

Soit **e** le sommet de départ.

1. La valeur de chaque cellule est un couple : (d(v), p(v))

Où : d(v) est la distance minimale du sommet de départ **e** au somment **v** et p(v) représente le sommet précédent du chemin optimal reliant e et v.

2. d(e)=0 (e est le sommet de départ)
3. d(v)=∞ si la distance est non encore calculé

Algorithme de Dijkstra

Etapes de l'algorithme :

	A	B	C	D	E	F	G
Etape 1	0	∞	∞	∞	∞	∞	∞
Etape 2	X						
Etape 3	X						
Etape 4	X						
Etape 5	X						
Etape 6	X						
Etape 7	X						

Algorithme de Dijkstra

Etapes de l'algorithme :

	A	B	C	D	E	F	G
Etape 1	0	∞	∞	∞	∞	∞	∞
Etape 2	X	(1,A)	(2,A)				
Etape 3	X						
Etape 4	X						
Etape 5	X						
Etape 6	X						
Etape 7	X						

Algorithme de Dijkstra

Etapes de l'algorithme :

	A	B	C	D	E	F	G
Etape 1	0	∞	∞	∞	∞	∞	∞
Etape 2	X	(1,A)	(2,A)	∞	∞	∞	∞
Etape 3	X						
Etape 4	X						
Etape 5	X						
Etape 6	X						
Etape 7	X						

Algorithme de Dijkstra

Etapes de l'algorithme :

	A	**B**	**C**	**D**	**E**	**F**	**G**
Etape 1	0	∞	∞	∞	∞	∞	∞
Etape 2	X	<u>(1,A)</u>	(2,A)	∞	∞	∞	∞
Etape 3	X	X		(3,B)		(4,B)	
Etape 4	X	X					
Etape 5	X	X					
Etape 6	X	X					
Etape 7	X	X					

Algorithme de Dijkstra

Etapes de l'algorithme :

	A	**B**	**C**	**D**	**E**	**F**	**G**
Etape 1	0	∞	∞	∞	∞	∞	∞
Etape 2	X	<u>(1,A)</u>	(2,A)	∞	∞	∞	∞
Etape 3	X	X	(2,A)	(3,B)	∞	(4,B)	∞
Etape 4	X	X					
Etape 5	X	X					
Etape 6	X	X					
Etape 7	X	X					

Algorithme de Dijkstra

Etapes de l'algorithme :

	A	**B**	**C**	**D**	**E**	**F**	**G**
Etape 1	0	∞	∞	∞	∞	∞	∞
Etape 2	X	<u>(1,A)</u>	(2,A)	∞	∞	∞	∞
Etape 3	X	X	<u>(2,A)</u>	(3,B)	∞	(4,B)	∞
Etape 4	X	X	X	(3,B)	(6,C)		
Etape 5	X	X	X				
Etape 6	X	X	X				
Etape 7	X	X	X				

Algorithme de Dijkstra

Etapes de l'algorithme :

	A	B	C	D	E	F	G
Etape 1	0	∞	∞	∞	∞	∞	∞
Etape 2	X	<u>(1,A)</u>	(2,A)	∞	∞	∞	∞
Etape 3	X	X	<u>(2,A)</u>	(3,B)	∞	(4,B)	∞
Etape 4	X	X	X	(3,B)	(6,C)	(4,B)	∞
Etape 5	X	X	X				
Etape 6	X	X	X				
Etape 7	X	X	X				

Algorithme de Dijkstra

Etapes de l'algorithme :

	A	**B**	**C**	**D**	**E**	**F**	**G**
Etape 1	0	∞	∞	∞	∞	∞	∞
Etape 2	X	<u>(1,A)</u>	(2,A)	∞	∞	∞	∞
Etape 3	X	X	<u>(2,A)</u>	(3,B)	∞	(4,B)	∞
Etape 4	X	X	X	<u>(3,B)</u>	(6,C)	(4,B)	∞
Etape 5	X	X	X	X	(5,D)	(4,B)	(6,D)
Etape 6	X	X	X	X			
Etape 7	X	X	X	X			

Algorithme de Dijkstra

Etapes de l'algorithme :

	A	B	C	D	E	F	G
Etape 1	0	∞	∞	∞	∞	∞	∞
Etape 2	X	<u>(1,A)</u>	(2,A)	∞	∞	∞	∞
Etape 3	X	X	<u>(2,A)</u>	(3,B)	∞	(4,B)	∞
Etape 4	X	X	X	<u>(3,B)</u>	(6,C)	(4,B)	∞
Etape 5	X	X	X	X	(5,D)	<u>(4,B)</u>	(6,D)
Etape 6	X	X	X	X	(5,D)	X	(6,D)
Etape 7	X	X	X	X		X	

Algorithme de Dijkstra

Etapes de l'algorithme :

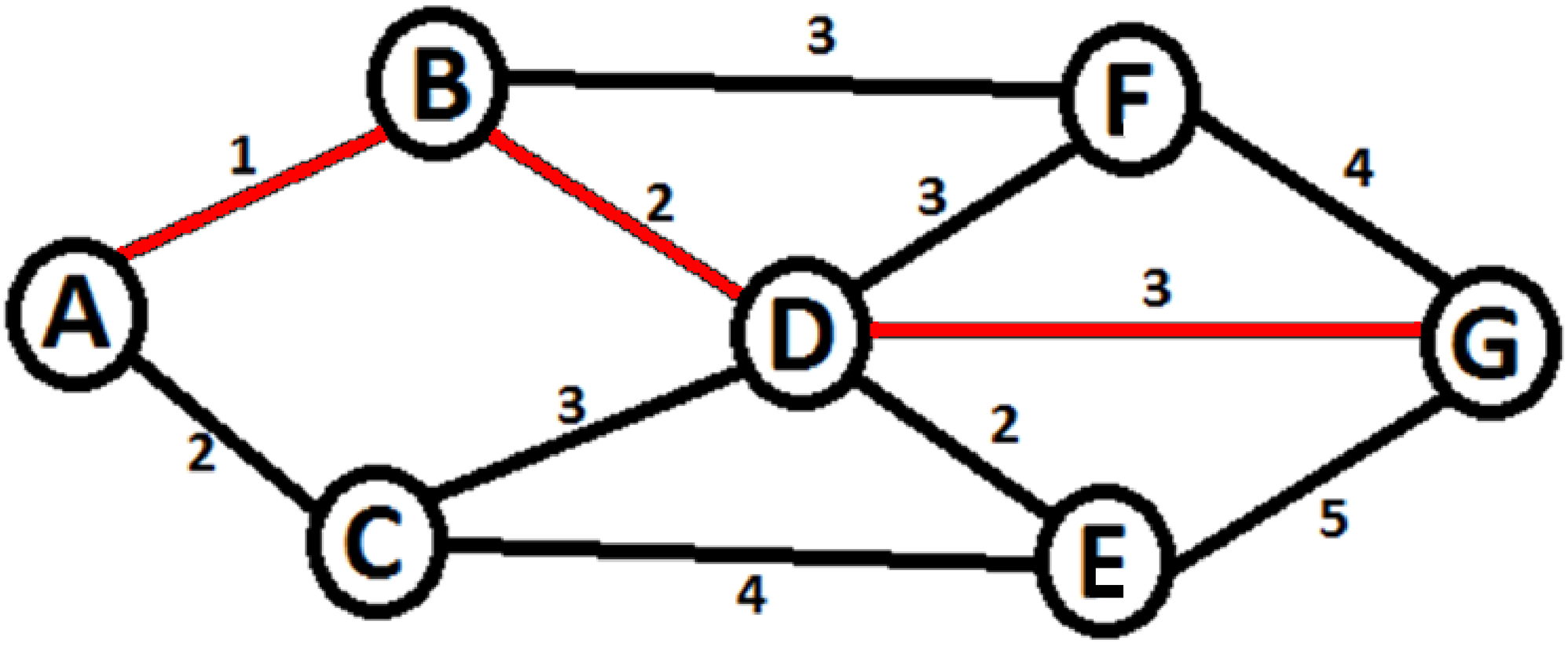
	A	**B**	**C**	**D**	**E**	**F**	**G**
Etape 1	0	∞	∞	∞	∞	∞	∞
Etape 2	X	<u>(1,A)</u>	(2,A)	∞	∞	∞	∞
Etape 3	X	X	<u>(2,A)</u>	(3,B)	∞	(4,B)	∞
Etape 4	X	X	X	<u>(3,B)</u>	(6,C)	(4,B)	∞
Etape 5	X	X	X	X	(5,D)	<u>(4,B)</u>	(6,D)
Etape 6	X	X	X	X	<u>(5,D)</u>	X	(6,D)
Etape 7	X	X	X	X	X	X	<u>(6,D)</u>

Algorithme de Dijkstra

Etapes de l'algorithme :

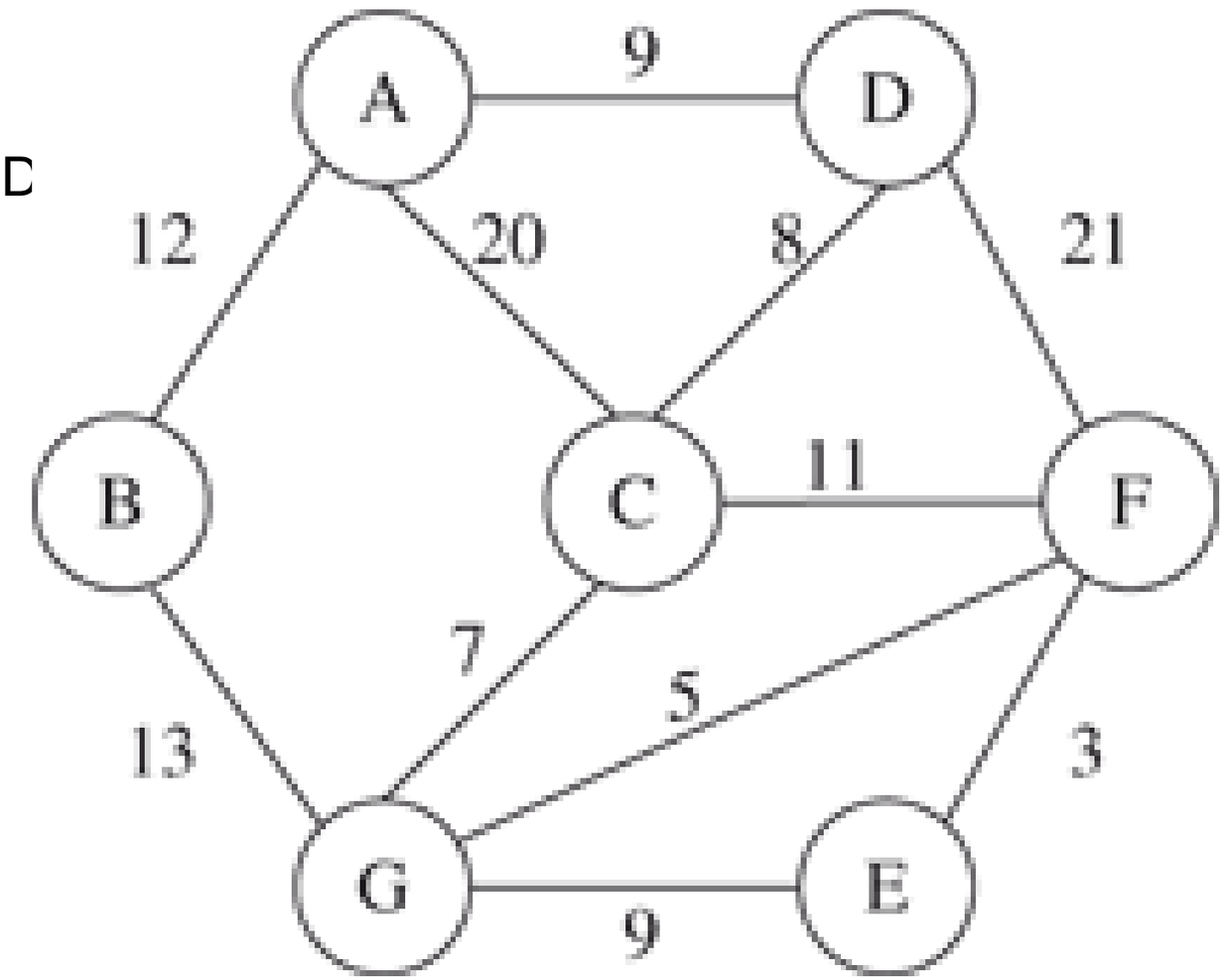
	A	**B**	**C**	**D**	**E**	**F**	**G**
Etape 1	0	∞	∞	∞	∞	∞	∞
...							
Etape 7	(0,A)	(1,A)	(2,A)	(3,B)	(5,D)	(4,B)	<u>(6,D)</u>

Le chemin optimal est : ABDG de coût égal à 6



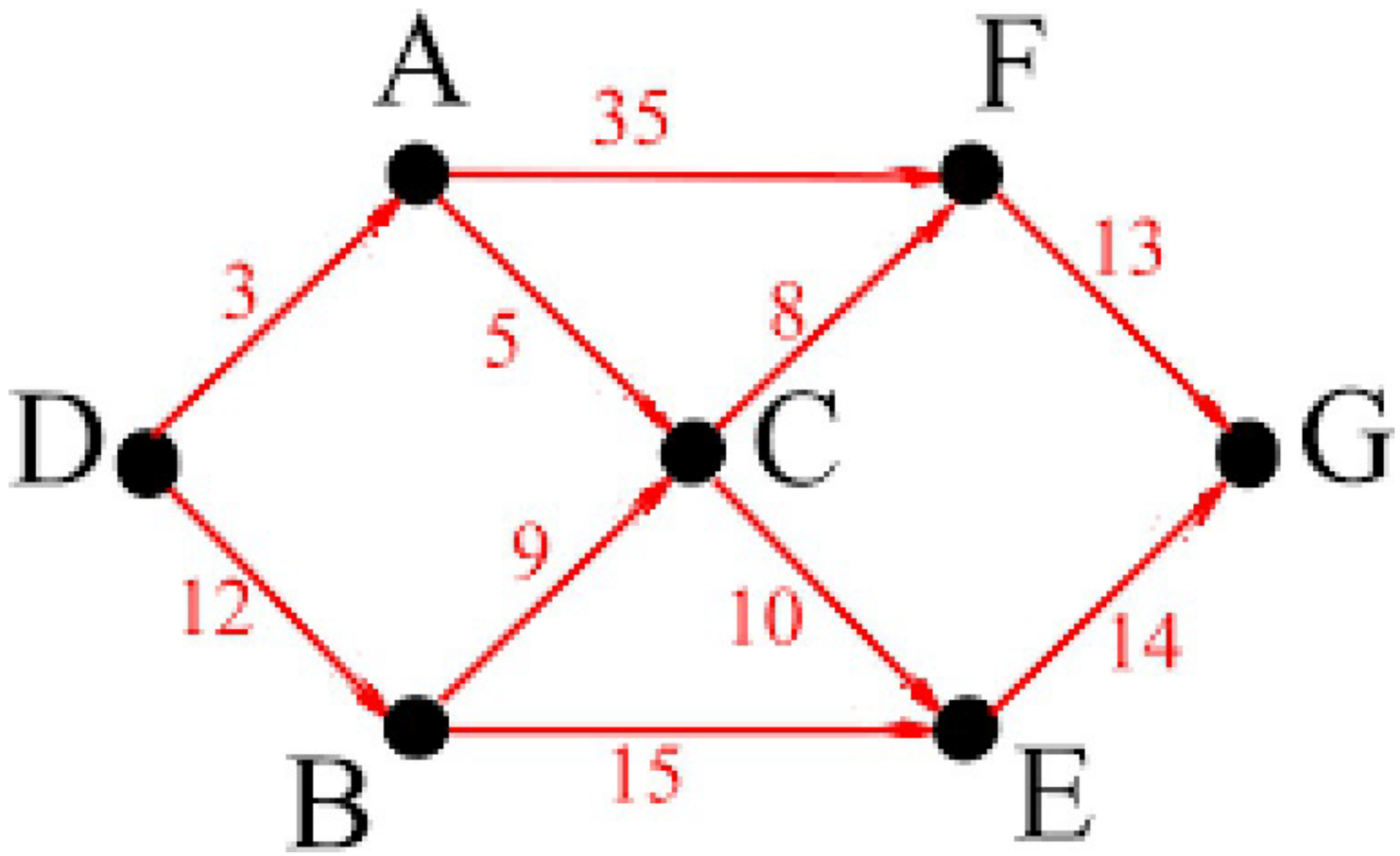
Exercice : Trouver le chemin optimal entre A et E

Algorithme de D



Exercice : Trouver le chemin optimal entre D et G

Algorithme de



Algorithme de Dijkstra

Etapes de l'algorithme :

	A	B	C	D	E	F	G
Etape 1	0	∞	∞	∞	∞	∞	∞
...							
Etape 7	(0,A)	(1,A)	(2,A)	(3,B)	(5,D)	(4,B)	<u>(6,D)</u>

En python on doit calculer deux dictionnaires :

D ={ 'A':0, 'B':1, 'C':2, 'D':3, 'E':5, 'F':4, 'G':6 } les distances

P ={ 'B':'A', 'C':'A', 'D':'B', 'E':'D', 'F':'B', 'G':'D' } dictionnaire des précédents

Programme de Dijkstra

```
def index_min_file(file):
    id = 0
    for i in range(1, len(file)):
        if file[i][0]< file[id][0] :
        id=i
    return id
```

```
def dijkstra(G, d, f):
    # Dictionnaires des distances minimales
    D={d:0}
    # Dictionnaire des précédents
    P={}
    # Liste des sommets visités
    visited = []
    # Liste des sommets en attente de traitement
    file = [(0,d)]
    while file!=[] :
        # On traite le sommet à distance minimale
        id = index_min_file(file)
        ds, s= file.pop(id)
        # Traiter le sommet s
        voisins =G[s]
        for dv, v in voisins:
            if v in visited:
                continue
            dn = dv + ds
            if v not in D or dn < D[v] :
                D[v]=dn
                file.append((dn, v))
                P[v]=s
        visited.append(s)
        if f in visited :
            break
    # Calculer le chemin optimale
    path = [f]
    x=f
    while x!=d:
        x=P[x]
        path.insert(0,x)
    return D[f], path
```

Exercice :

Implémenter l’algorithme de Dijsktra en utilisant la représentation du graphe par une matrice d’adjacence.

Des étudiants A, B, C, D, E et F doivent passer des examens dans différentes disciplines, chaque examen occupant une demi-journée :

– Algorithmique : étudiants A et B.

– Compilation : étudiants C et D.

– Bases de données : étudiants C, E, F et G.

– Java : étudiants A, E, F et H.

– Architecture : étudiants B, F, G et H.

On cherche à organiser la session d'examen la plus courte possible.