Stratégies et Techniques de Résolution de Problèmes

Chapitre III : algorithmes des Graphes

S7 - ECL – 2A – MI

2022-2023

Alexandre Saidi

I- TAS ou file de priorité (Heap, priority queue)

- Utilisée comme outil d'optimisation (cf. complexité) de multiples algorithmes et comme outil d'extraction ordonnée d'éléments (tri) d'une collection.

TAS ou File de priorité :

- Structure employée dans les algorithmes sur les graphes (cf. complexité).
- Dans les systèmes d'exploitation (file d'attente de ressources, file d'exécution, ...)
  - Une forme particulière de file d'attente.
  - Deux des opérations principales sur un TAS :

    Insertion : insertion à sa place ordonnée ≡ enfiler dans une file

    DeleteMin : extraire le plus petit/grand élément ~ défiler.

Relation d'ordre : info(tout nœud) < info(ses fils). (ou symétriquement '>')

I.1- Implantation et structure de données

Rappel : pour manipuler le minimum (resp. max) d'une telle séquence, on peut utiliser :

1- Une **liste** : *Insert_en_tête* (O(1)) et *DeleteMin* (O(N)).

2- Mieux vaut une l**iste triée** :

Insert est O(N) et Θ(N/2) et *DeleteMin* est O(1)

cf. un système d'exploitation avec modification permanente des files.

../..

3- Un Arbre Binaire Ordonné Horizontalement (ABOH) avec une moyenne de O(log N) pour les deux opérations de base (si ABOH **équilibré**).

Problème : après plusieurs Insert/*DeleteMin*, l'arbre sera déséquilibré (voir dans ce cas un ABOH équilibré et compact = AVL).

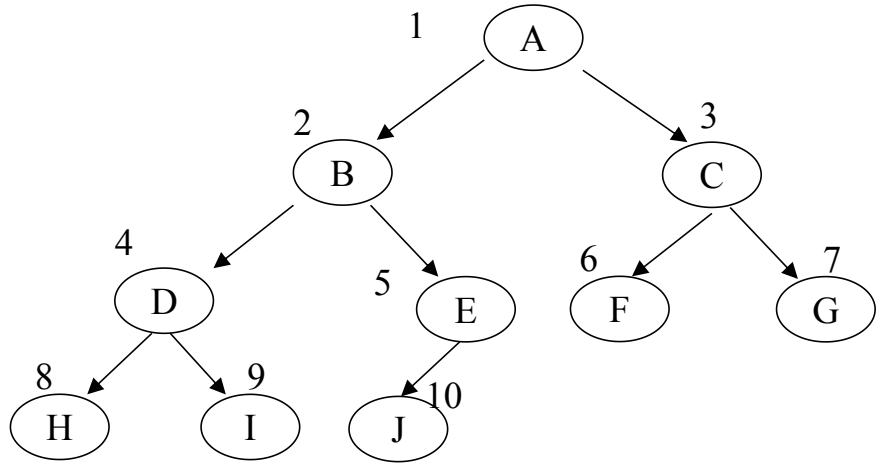
Question de représentation économe en mémoire (une table ?)

4- Une structure de données spécifique et simple avec une complexité O(log N) :

⇨**TAS**

On utilisera ce dernier choix.

I.2- TAS Binaire (Binary Heap)



- File de priorité **binaire**

C'est un arbre binaire **complet**.

Il est complètement rempli avec exception possible sur le niveau **le plus bas** qui est rempli **de gauche à droite** (*complete binary tree*).

N.B. : un **Arbre Binaire Complet** (*ABC*) non vide de hauteur *h* possède $[2^{h-1},\ 2^h-1]$ nœuds (avec par convention : **h(vide)=0** et **h=1** pour un seul élément à la racine) .

$h=\lceil log_2 N \rceil$ (la partie entière+1)

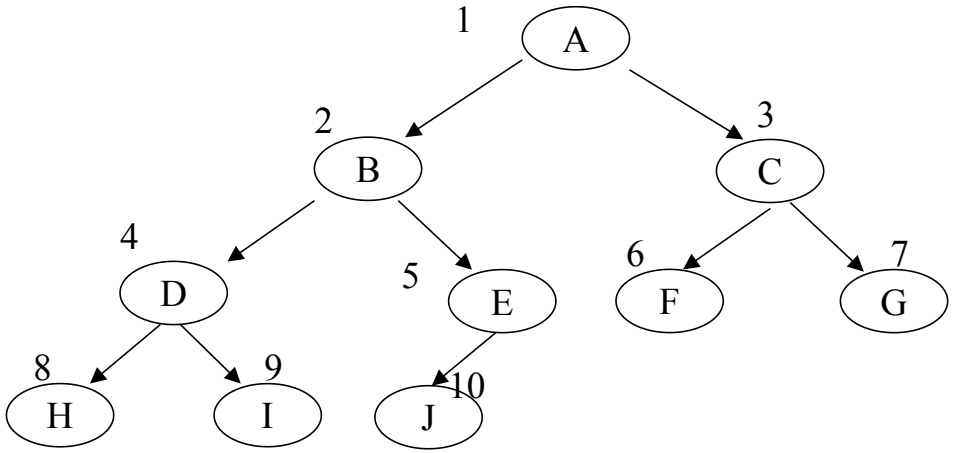
Si $N=2^{h-1}$, le dernier niveau a un seul élément (en bas à gauche).

Si $N=2^h-1$, le dernier niveau est complet.

N.B. : la **représentation abstraite** d'un Tas = un arbre binaire complet (ABC)

AVEC une **représentation interne** par un tableau (vecteur).

- pour tout nœud à la position **i**, les 2 fils seront en **2i** et **2i+1** (si N=1, i=1)
- pour un nœud en position **j**, le parent est en position $\left\lfloor \frac{j}{2} \right\rfloor$



⇨Pas besoin d'une liste;

les opérations de parcours de l'arbre sont simples / rapides.

⇨ Un inconvénient possible : borner par avance la taille max du vecteur.

- Le tableau représentant le TAS ci-dessus :

indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	..

indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	..
valeur	*	A	B	C	D	E	F	G	H	I	J	...			

Remarques sur la hauteur (h) et le nombre d'éléments (N) :

Rappel : un ABC non vide de hauteur *h* possède $[2^{h-1},\ 2^h-1]$ nœuds

Convention alternative utilisée :

si on note $h' = h-1$, on aura :

si $h'= 0$, nombre d'éléments = 1

$h'= 1$ = 2 à 3 $(2^1,\ 2^2-1)$ éléments

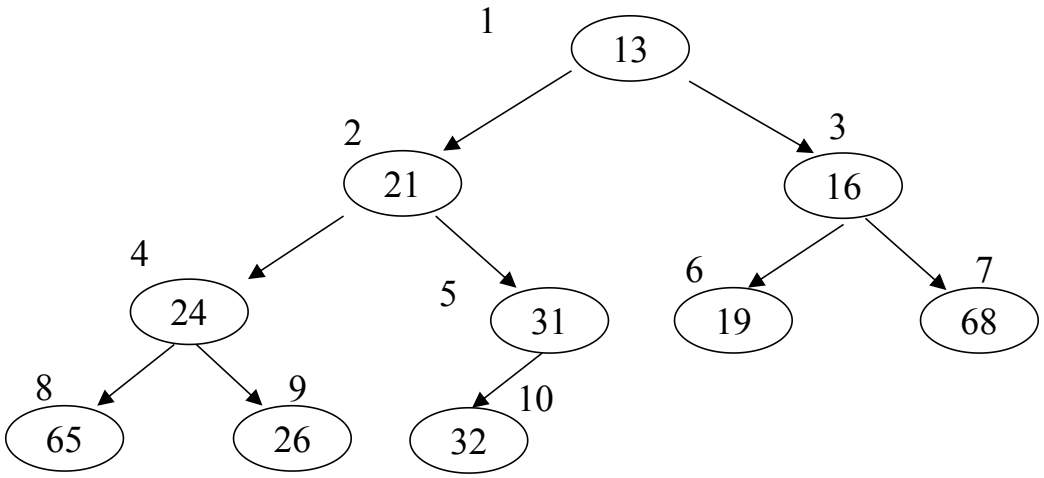
$h'= 2$ = 4 à 7 $(2^2,\ 2^3-1)$ éléments

$h'= k$ = 2^k à $2^{k+1}-1$ (selon le remplissage du niveau k)

I.3- Propriétés d'un Tas

- Un **min_heap** : le plus petit élément est au sommet de l'arbre (à la racine) ;
- Un **max_heap** : la racine contient le maximum.
- Dans un **min_heap** :

  Tout nœud **n** est plus petit que ses descendants :



Tout nœud **n ≤ minimum(fils-gauche, fils_droit)**

- Relation d'ordre sur les nœuds (à définir) surtout si info(noeud) complexe.
- La relation d'ordre ci-dessus est inversée pour un max_heap .
- Dans la suite, on envisage un **min_heap**.

- Dans un TAS (binaire), tout nœud a exactement 2 fils

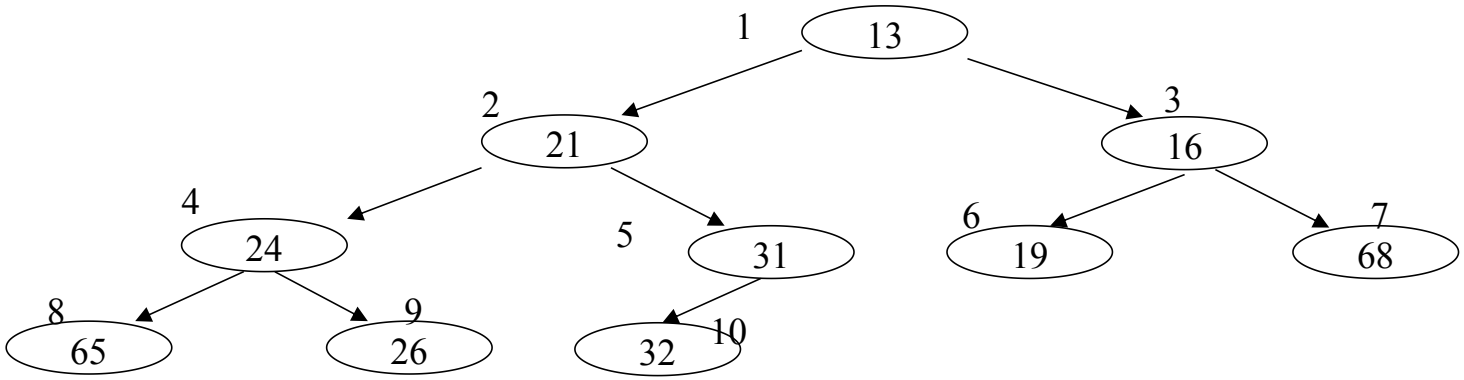
  (sauf pour les parents du dernier niveau qui peuvent en avoir 1).

- Trouver le minimum est rapide O(1).

  Mais, après un retrait, il faut réorganiser l'arbre.

- Un nœud peut contenir une information complexe (e.g. ville × habitants × ....)

  Dans ce cas, il faut définir la relation d'ordre (la clef pour les comparaisons).



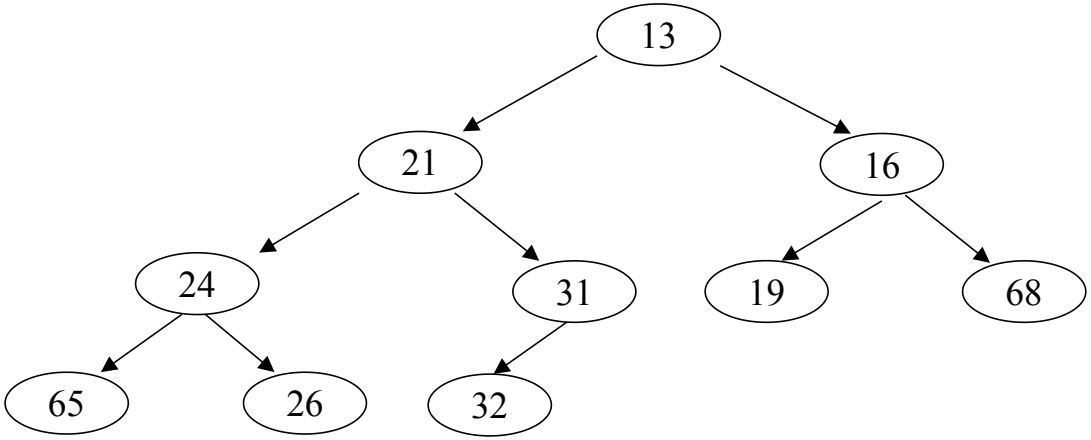
I.4- Les opérations

I.5- Insertion

Pour insérer un élément X, on le place dans la prochaine place libre (au dernier niveau pour respecter ABC) puis l'on rétablit la relation d'ordre dans l'arbre.

Pour rétablir la relation d'ordre :

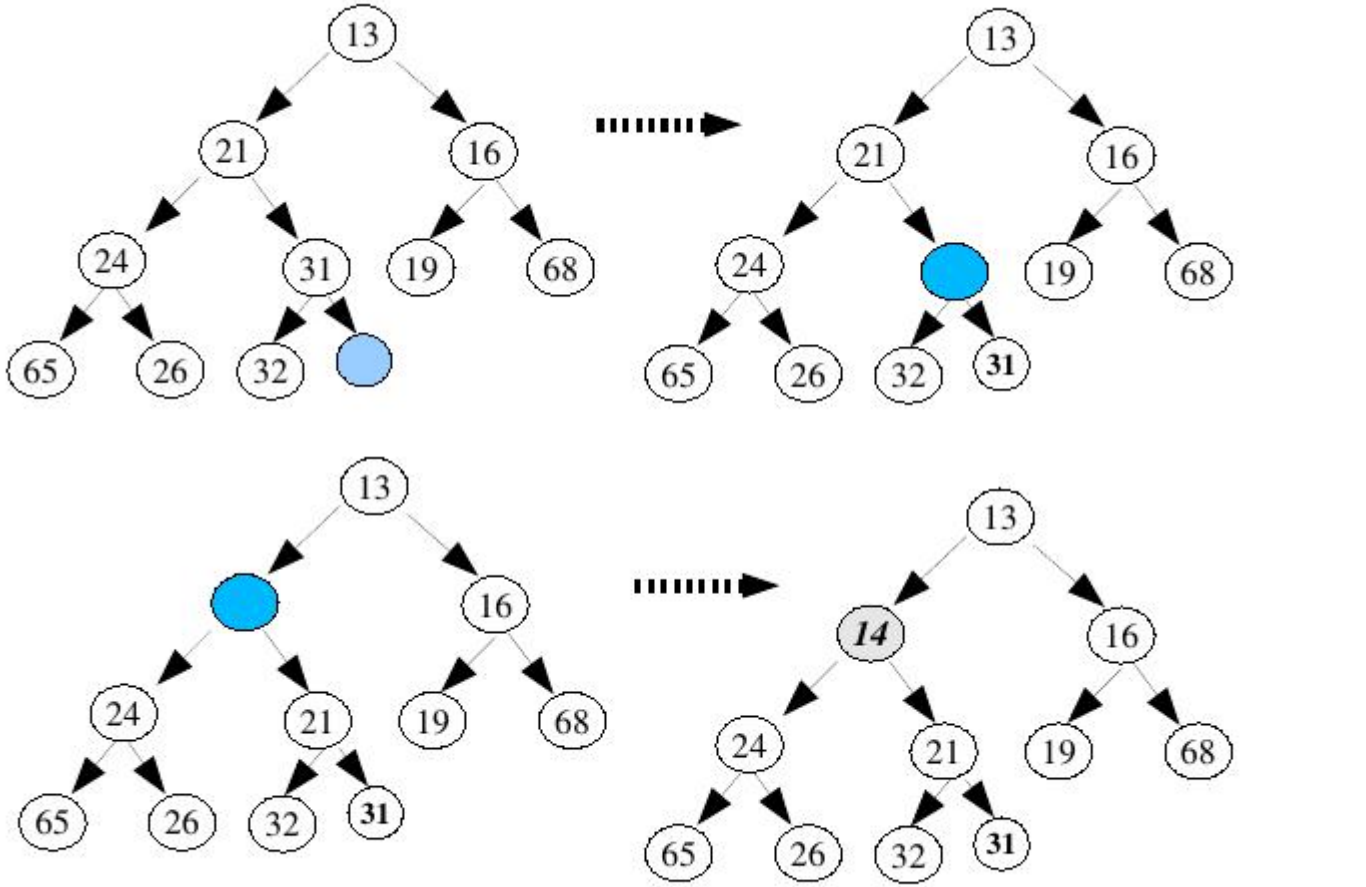
- Si X est bien placé (en toute dernière place) alors fini.
- Sinon, on permute avec son père (on descend le père; on remonte X)



On continue cette opération jusqu'à placer X à sa place.

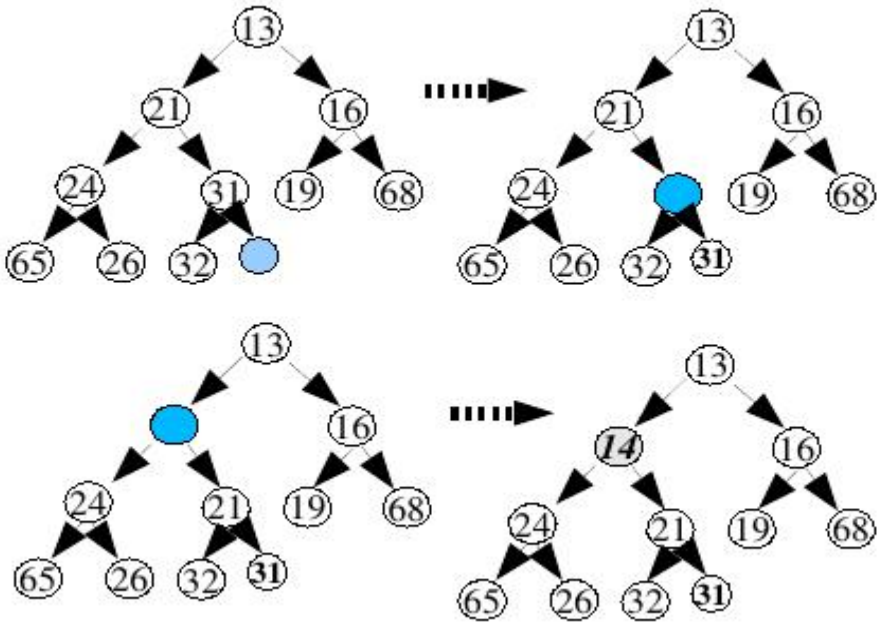
Exemple : Insertion de 14

(Le cercle vide bleu en dernière place est initialement réservée pour 14)



Remarques techniques :

- Pour placer 14, au lieu de permuter avec son "père" (3 affectations), on peut simplement "descendre le père".



- La relation d'ordre '<' peut être étendue à '≤' pour accepter des doublons.

I.5.1- Complexité de l'algorithme Insert

- L'arbre associé ayant un niveau h, on fera, au pire, *h* percolations (move_up)

  O(log(N)).

I.5.2- L'algorithme d'insertion

```
Action insert(X) :                                  // X est du type Element
    Si heap_plein() Alors exception "plein" Finsi   // erreur, il faut faire davantage de place dans tableau sous-jacent
    taille_actuelle += 1                            // taille_actuell : nb_elements
    hole = taille_actuelle                          // nb éléments+1 : on suppose avoir de la place disponible
    TQ (hole > 1 && X < tableau[hole/2])            // '<' ou '≤'   TQ : action de "remontée" du trou (descente du père)
        tableau[hole]=tableau[hole/2]
        hole = hole / 2
    tableau[hole] = X
Fin insert
```

Remarques techniques :

- L'action "remontée" (*percolateUp)* de ce code évite les permutations.

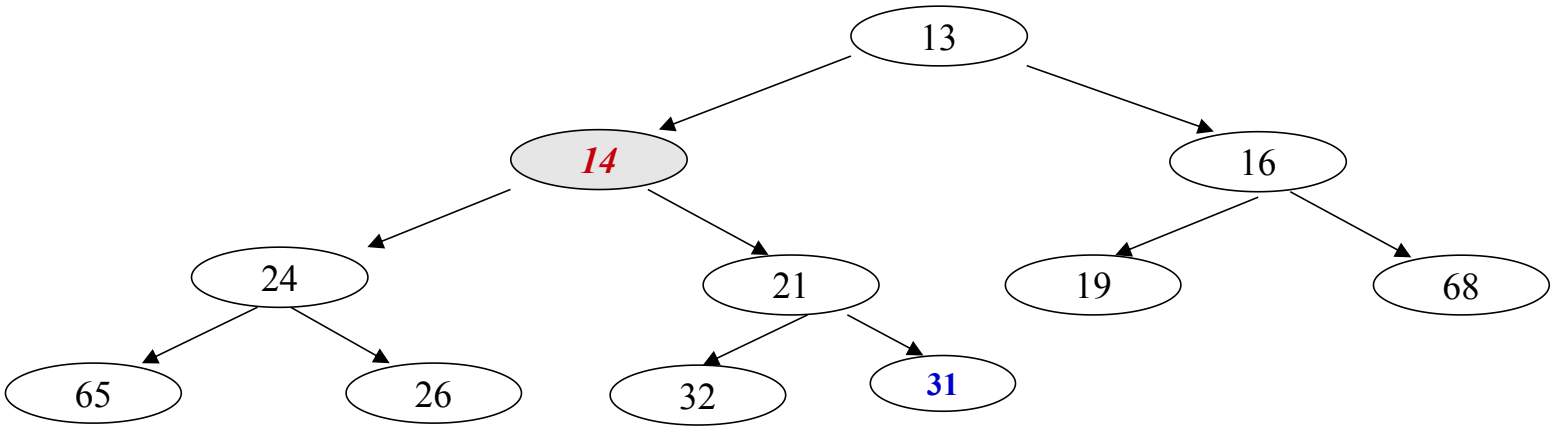
  Si on sépare la partie *percolateUp*, on sera obligé de faire des permutations.

- Si X est le minimum, il doit être placé au sommet.

Pour cela, la boucle s'arrête sur *hole*=1 et l'insertion se fera dans *tableau[1]*.

Le premier élément est donc ici à l'indice 1 → tableau[0] est libre

Dans l'exemple :



indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14 ...
valeur	*	13	14	16	19	21	24	26	31	32	65	68			

- Pour la relation 2i, 2i+1, la valeur placée dans tableau*[0]* est appelé **sentinelle**.

Cette idée est proche de celle de "dummy" dans les listes.

I.6- L'opération DeleteMin

- Le retrait du minimum est immédiat ($\Theta(1)$) MAIS

  une fois le minimum retiré du TAS, un *trou* est créé à la racine et il faut réorganiser l'arbre pour respecter la relation d'ordre.

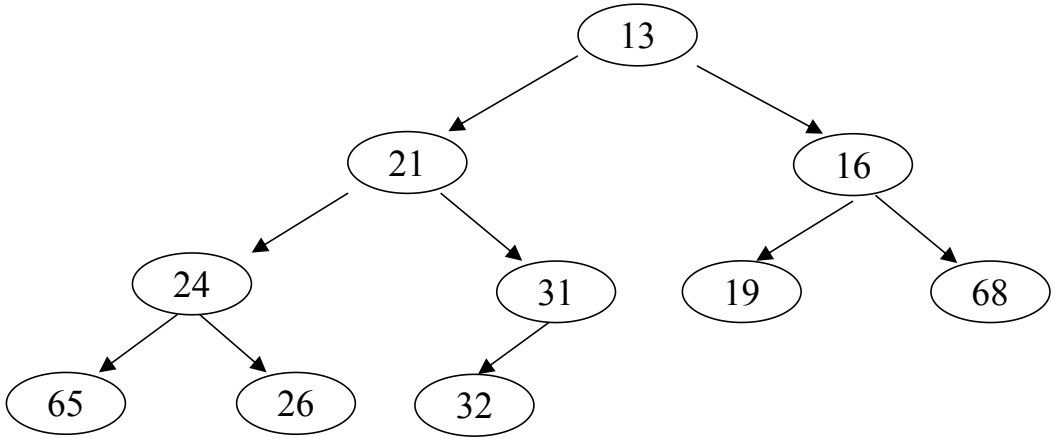
- Soit X le dernier élément du Tas (quelle que soit la valeur de X).

Puisque le Tas possède un élément en moins, X sera **certainement** déplacé.

On peut donc supprimer X en le plaçant au sommet (la place de l'ancien *min*).

Si X est à sa place : fini

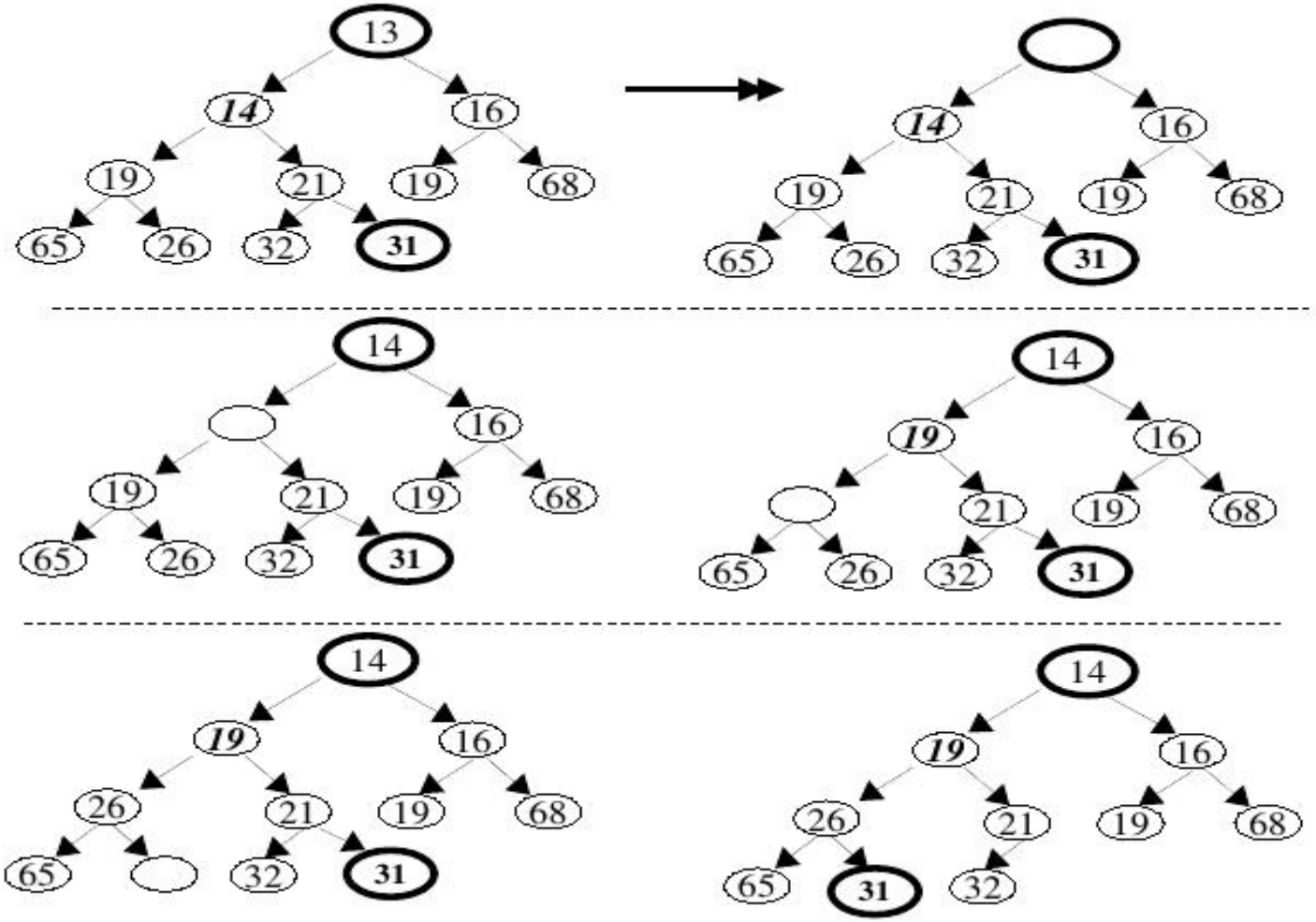
Sinon, on permute X et son **plus petit fils** et on redescend X.



L'opération est répétée jusqu'à ce que X soit bien placé.

Exemple : retrait du minimum et le réarrangement de l'arbre

N.B. : noter présence de doublon 19 (pas de problème) !



I.6.1- L'algorithme DeleteMin

La méthode ci-dessus est appelée également "*PercolateDown*".

Elle évite les permutations (au profit d'affectations).

```
Fonction Heap::DeleteMin() :
        Si heap_plein() Alors exception "plein"
        Min=tableau[1];
        tableau[1]=tableau[taille_actuelle];     taille_actuelle -= 1;
        PercolateDown(1);
        renvyer Min
Fin  DeleteMin

Action  Heap::PercolateDown( hole)
        temp=tableau[hole];                                          // on évite les permutations
        TQ (hole*2 ≤ taille_actuelle)
           child=hole * 2;                                           // child  forcément ≤  taille_actuelle
           Si (child != taille_actuelle && tableau[child+1] < tableau[child])
                 child +=1                                    // le fils le plus petit est en tableau[child+1]
           Si (tableau[child] < temp) tableau[hole]=tableau[child]   // on remonte
           Sinon  break                                              // terminé : hole est la bonne place (non feuille)
           hole =child;
        tableau[hole]=temp;                                          // cas où hole en un nœud ou en feuille
Fin  PercolateDown
```

I.6.2- Complexité de l'algorithme DeleteMin

- La complexité de la fonction *PercolateDown* est **O(log N)**.

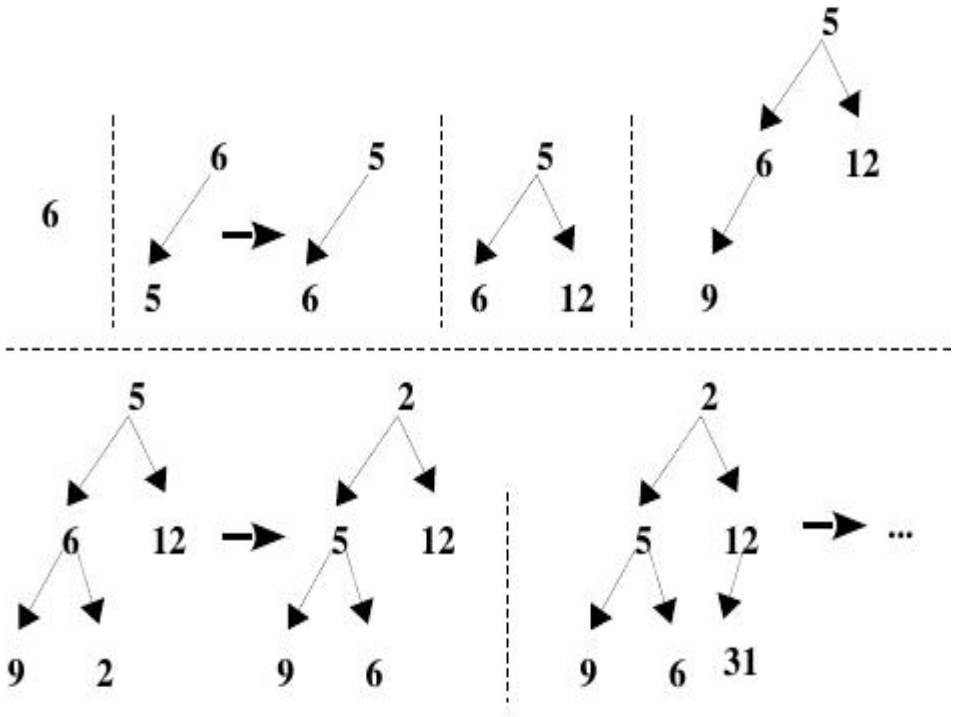
Au pire, l'élément remonté à la racine doit descendre (tout en bas).

I.7- Opération de construction de Tas

- Pour construire un TAS, on peut procéder à N insertions.

Chaque insertion étant O(log N), la construction est **O(N log N)** dans le cas pire (e.g. données sont dans l'ordre inverse avec *PercolateDown* pour tous).

Un exemple : construction pour la séquence <6, 5, 12, 9, 2, 31, 3>



- N.B. : Si on fait N insertions (sans aucune autre opération), on aura une complexité (moyenne) proche de Θ(log N) pour chaque élément (cf. complexité de l'insertion).

La solution suivante améliore cette complexité (de O(**N.LogN**) en O(**N**)).

Solution : laisser le niveau le plus bas s'organiser tout seul !

- Créer un arbre binaire avec les N éléments NON ordonnés (= remplir simplement le tableau associé) puis établir la relation d'ordre du Tas en appliquant l'algorithme suivant :

  → tableau est une donnée de la classe **Heap** (d'où l'absence de paramètres)

```
Action Heap::BuildHeap()          // application à l'arbre binaire créé
    i=taille_actuelle /2          // taille_actuelle = N
    TQ  i > 0 :                   // l'attribut tableau (qui était brut) devient ordonnée = devient un TAS
       PercolateDown(i);
       i -= 1
Fin  BuildHeap
```

- On utilise donc N/2 éléments (on délaisse le dernier niveau de l'arbre) et on applique *PercolateDown* à chaque élément de la moitié supérieure.
- Rappel : les valeurs de **i** représentent le schéma **2i** et **2i+1** (les indices)
- Cet algorithme est O(N) : voir page suivante.

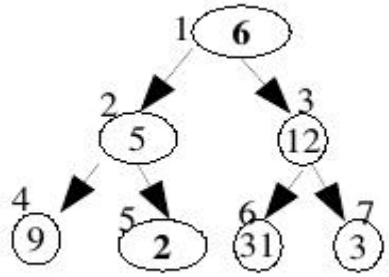
Exemple : pour la séquence <6, 5, 12, 9, 2, 31, 3> de taille 7

on a : N/2=3 et pour la moitié concernée : i=3,2,1

L'arbre initial (TAS pas encore ordonné) contenant la séquence :

indice	0	1	2	3	4	5	6	7	8
valeur	*	6	5	12	9	2	31	3	

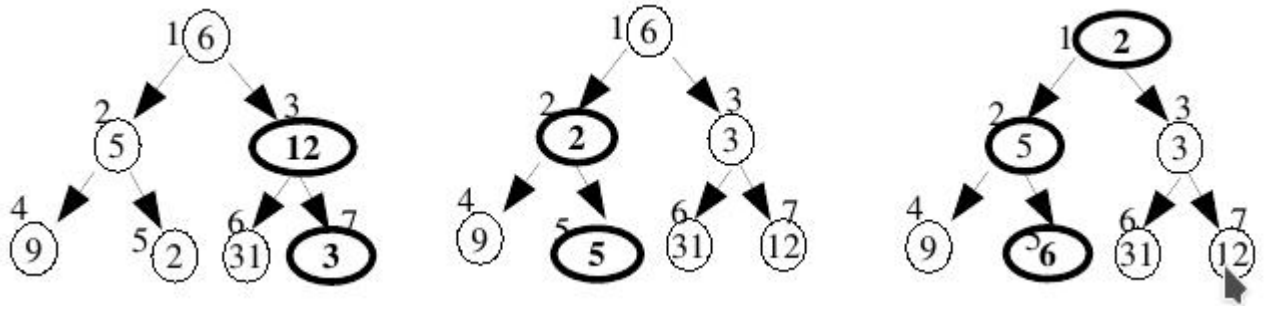
- Pour i=3, Heap[3]=12, l'appel de *PercolateDown(*3) fait descendre la valeur **12** et fait remonter son plus petit fils **3** (Heap[7]=3). Voir fig ci-dessous.



- Puis, pour i=2, on s'intéresse au couple Heap[2]=5 et Heap[5]=2

- Puis, pour i=1, on s'intéresse au couple Heap[1]=6, Heap[2]=2 et Heap[5]=5.

Ce qui fera remonter 2 à la racine et 5 au 2e niveau, 6 devient feuille.



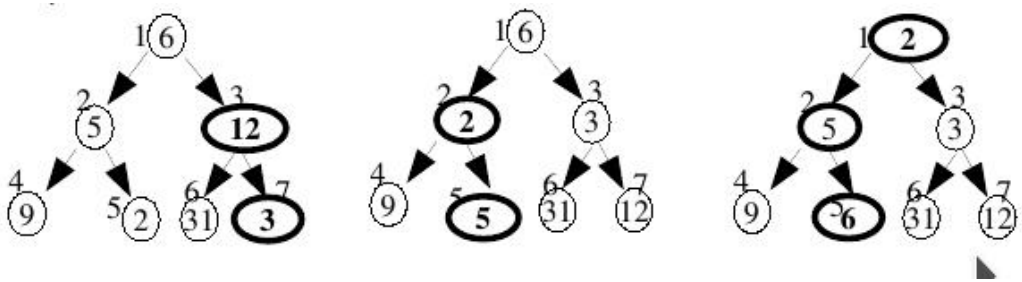
I.7.1- Complexité de BuildHeap (dernière version)

- Pour le niveau 0, on fait **h** comparaisons (h arêtes)
- Pour le niveau 1, on a 2^1=2 éléments et pour chacun, on fait *h-1* comparaisons
- Pour le niveau i, on a 2^i éléments et pour chacun, on a h-i arêtes (= h. de chaque élément).

Donc, pour un arbre de hauteur *h*, on a la somme des hauteurs des nœuds

= nombre total des comparaisons par *PercolateDown* = $\sum 2^i(h-i)$

- Dans l'exemple, h=2 (en partant de 0) et



- pour la racine : 2 comparaisons
- pour les 2 éléments du niveau 1 :

une comparaison pour chaque le total des comparaisons=4

- La somme des comparaisons sera : S = $\sum 2^i(h-i)$= h + 2(h-1) + 4(h-2) + ... + 2^{h-1}
- Multiplier S par 2 et faire les soustractions puis simplifier :

$$\mathbf{S=-h + 2 + 4 + 8 + \ldots + 2^{h-1} + 2^h =(2^{h+1} - 1) - (h +1) \simeq N\text{-}\log N}$$

Complément : Détails du calcul précédent :

Poser $A = 2 + 4 + 8 + \ldots + 2^{h-1}$, calculer $2A$ puis développer en utilisant

$2h - 2(h-1) = 2$

$4(h-1) - 4(h-2) = 4$ .....

N.B. : avec $\mathbf{S = (2^{h+1} - 1) - (h+1)}$, si $N=2^h$, pour **N grand**, on aura

$$S = 2N - (h+2) \simeq 2N \quad \mathbf{O(N)}.$$

On constate que $S \approx N$.

Rappel : avec la convention alternative pour désigner *h* au début de ce chapitre, la complexité sera du même ordre

Précisions sur la complexité de BuildHeap :

NB : on démontre qu'au cas pire, on fera entre ***1.5N*** et ***1.625N*** tests

Remarques :

- Pour un arbre contenant N éléments, on peut démontrer (par induction) que la somme des hauteurs **S=N – b(N)** où **b(N)** est le nombre de 1 dans la représentation binaire de N. Pour *h* commençant à 0, **b(N)=h+1**.
- Il a été démontré la même complexité pour la version non optimisée de BuildHeap.

  → la version non optimisée : traiter tous les niveau au lieu de la moitié sup.
- Pourquoi ne pas utiliser un ABOH pour un Tas ?

  L'insertion dans un ABOH est O(log N) et *DeleteMin* (qui enlève le minimum) est O(l).

Cependant, **ces complexités sont uniquement pour un ABOH équilibré.**

⇨ On pourra utiliser un AVL mais un AVL est quelque peu "lourd" pour un Tas.

II- Exemples d'application du Tas

I)- Soit une séquence de N éléments.

Trouver le k^e plus petit / plus grand élément.

Solution classique : trier les valeurs puis prendre le k^e élément.

Complexité = celle du tri (au mieux **O(N log N)** e.g. par *merge_sort*)

2e solution :

Voir chapitre 1, calcul de la médiane : O(N).

3e solution : si on décide d'utiliser un TAS :

- Utiliser BuildHeap pour construire le Tas : O(N)
- Appliquer *k* DeleteMin : O(k . log N)
- Complexité= O(k . log N)
- Si $k = \left\lceil \frac{N}{2} \right\rceil$, on aura θ**(N log N)**

II)- Heap Sort : pour trier N éléments, construire un Tas puis extraire un par un les minima (par DeleteMin) et les ajouter dans un tableau qui sera trié.

→ Les solutions O(N) restent meilleures.

III- Heap-Sort : utilisation d'un TAS

– Étant donné un ensemble de N données, on crée un Heap.

– Si les données arrivent une par une, la construction du Heap est O(N. log N)

Si les données sont disponibles, on aura O(N) (par buildHeap)

– Pour trier ces données, on extrait les minima successifs du Tas

```
Action Tri_Tas_ascendant(Tab) :        // Tab : Tableau de taille N à trier
  TAS Tas(Tab);                        // Par le Ctor de TAS, Tab devient le tableau brut du TAS (le Ctor appelle
Build_Heap())
  Tab = vide                           // Préparation pour recevoir la séquence ordéonnée
  Pour i = 1.. N
     X = Tas.DeleteMin()                    //  extraction du minimum
     <Ajouter simplement X  dans Tab>       // Tab contiendra à la fin une séquence ordonnée
Fin  Tri_Tas_ascendant
```

- La complexité de cet algorithme (appelé Heap-Sort) est **O(N. Log N)**.
- **Heap-sort** est un algorithme utilisé lorsque Quick-sort risque **$O(N^2)$**.
- Parmi les applications utilisant un TAS, on étudie Dijkstra, MST, Flux, ...

IV- TAS en Python

TAS (ou Heap Queue) en Python : une variante d'une File de priorité.

Python permet d'aborder un TAS comme une liste Python standard :

heap[0] est le plus petit élément et **heap.sort()** conserve l'invariant du TAS.

Son implantation utilise un tableau/une liste (soit heap) où (rappel) :

forall k>=0 : heap[k] <= heap[2k+1] et heap[k] <= heap[2k+2].

⇨Les TAS en Python adapte la relation 2i+1, 2i+2 et utilise l'indice 0.

Les éléments absents sont considérés comme infinis.

Dans un tel arbre (min_heap), le minimum est toujours au sommet.

Exemple : ../..

IV.1- Un exemple complet (max_heap)

Dans cet exemple, on crée un TAS à partir d'une liste puis on examine son contenu avant d'afficher ses éléments un par un dans l'ordre.

```
import heapq , random
def test_tas(listForHeap):    # On reçoit une liste
    print("la liste initiale avant hepify :", listForHeap)
    # heapq.heapify(listForHeap)              # pour  min_heap   (par défaut)
    heapq._heapify_max(listForHeap)         # pour max_heap
    print("Le contenu après heapify", listForHeap)
```

On obtient :

```
la liste initiale avant hepify : [18, 5, 1, 16, 14, 7, 2, 18, 8, 15]
Le contenu après heapify [18, 18, 7, 16, 15, 1, 2, 5, 8, 14]
```

Il reste à utiliser les indices i, 2i+1 et 2i+2 ../..

La suite du même code :

```
print("Affichage du tas avec root=0 puis 2i+1 et 2i+2 : ")
i=0
print(f"i : {i}, ele à cet indice = {listForHeap[i]}")
while(True) :
    if 2*i+1 < len(listForHeap) : print(f"i : {2*i+1}, ele à cet indice = {listForHeap[2*i+1]}")
    else:  break
    if 2*i+2 < len(listForHeap) : print(f"i : {2*i+2}, ele à cet indice = {listForHeap[2*i+2]}")
    else:  break
    i+=1
print('-'*50)
```

La trace :

```
Affichage du tas avec root=0 puis 2i+1 et 2i+2 :
  i : 0, ele à cet indice = 18
  i : 1, ele à cet indice = 18
  i : 2, ele à cet indice = 7
  i : 3, ele à cet indice = 16   <<– le plus grand suivant est bien en heap[3]
  i : 4, ele à cet indice = 15
  i : 5, ele à cet indice = 1
  i : 6, ele à cet indice = 2
  i : 7, ele à cet indice = 5
  i : 8, ele à cet indice = 8
  i : 9, ele à cet indice = 14
```

Retrait des éléments dans l'ordre :

```
    # Suite  de la fonction test_tas(listForHeap)
    print("Heap sort : Affichage avec retrait un par un ")
    for i in range(len(listForHeap)) :
        pop_max = heapq._heappop_max(listForHeap)
        print(pop_max, end=", ")
    print()
```

Trace :

```
Heap sort : Affichage avec retrait un par un
18, 18, 16, 15, 14, 8, 7, 5, 2, 1,
```

La partie Main :

```
if __name__ == "__main__" :
    # Test avec le heap
    liste=[random.randint(1,20) for i in range(10)]
    # Test général du tas (Attention, la liste changera)
    print('-'*50)
    test_tas(liste)
```

IV.2- Exemples : heap-sort avec heappush et heappop

```
from heapq import *

def heapsort_by_heapq(liste):
    h = []
    for value in liste:
        heappush(h, value)
    return [heappop(h) for i in range(len(h))]

#--------------------------------------
if __name__ == "__main__" :
    print(heapsort_by_heapq([1,5,2,1,8,6,9]))

# Trace :
[1, 1, 2, 5, 6, 8, 9]
```

Une 2e manière : avec priorityQueue

```
from queue import PriorityQueue

# MM chose avec priorityQueue (put et get)
def heapsort_by_priority_queue(liste):
    h = PriorityQueue()
    for value in liste:
        h.put(value)
    return [h.get() for i in range(h.qsize())]
#---------------------------------------
if __name__ == "__main__" :
    print(heapsort_by_priority_queue([1,5,2,1,8,6,9]))

# Trace :
[1, 1, 2, 5, 6, 8, 9]
```

ET avec heapify (construction du TAS)

```
from heapq import *

def heapSort_avec_heapyfy(liste) :
    heapify(liste)
    return [liste.pop(0) for i in range(len(liste))]
# Si on ne veut pas pop(0) mais seulement pop(), la liste sera trié ordre descendant
#---------------------------------------
if __name__ == "__main__" :
    print(heapSort_avec_heapyfy([1,5,2,1,8,6,9])) # Liste sera triée descendant si on fait pop() dans la fonction.

# Trace :
[1, 1, 2, 5, 6, 8, 9]
```

IV.3- classe PriorityQueue

La différence d'un **heapq** par rapport à une File de priorité (**PriorityQueue**):

heapq est (par défaut) un min-heap

→ Toute intervention (sauvage) sur le tableau sous-jacent risque de détruire le TAS

Mais on peut réparer (cf. l'exemple ci-dessus).

Exemple pour Max_heap

```
import heapq
listForTree = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
heapq.heapify(listForTree)              # pour  min_heap (par défaut)
heapq._heapify_max(listForTree)         # pour max_heap

listForTree      # Est devenu un max_tas : 15 est en tête.
#  [15, 11, 14, 9, 10, 13, 7, 8, 4, 2, 5, 12, 6, 3, 1]
```

IV.3.1- Un exemple de PriorityQueue

```
import queue as Q
def test_priorityQueue() :
    q = Q.PriorityQueue()
    q.put(10)
    q.put(1)
    q.put(5)
    while not q.empty():
        print(q.get(),end=', ')
    print()
# Trace :   1, 5, 10,
```

PriorityQueue est plutôt utilisé avec les threads et Processus où une Queue est partagée entre les processus.

V- Addendum : autres opérations sur les Tas *(section optionnelle)*

- **Get_max** : trouver le maximum dans un *min_heap* :

il faudra tout examiner car le maximum doit se trouver au niveau des feuilles.

Il y a environ N/2 (2^{h-1}) éléments au niveau des feuilles.

- D'une manière générale, le Tas possède peu d'information sur l'ordre des éléments : **trouver un élément quelconque nécessite O(N).**

- **TAS indirect** : pour représenter des informations complexe sur les éléments, utiliser une table de hachage.

→ Le TAS contient des clefs (de Hachage) sur les éléments stockés ailleurs.

../..

- En principe, les éléments d'un TAS ne sont pas modifiables (comme dans un ABOH).

  Mais dans certains cas, on dispose d'opérations supplémentaires de modification comme : **Decreasekey , Increasekey, Remove...**

- **decreaseKey(Position, delta)** :

  réduire la valeur de l'élément à la position *Position* de *Delta*

  il faut ensuite éventuellement procéder à des *move_up*.

  Exemple : utilisé dans les syst. d'exploitation où l'on modifie les priorités des tâches

  Complexité : **O(log N)**

- **increaseKey(Position, delta)** : idem.

  Dans les systèmes d'exploitation, les priorités des tâches sont modifiées après un certain temps passé dans le CPU

  Complexité : **O(log N)**

- **remove(Position)** : supprimer le nœud de position *Position*.

  Cette opération peut être réalisée par *decreaseKey(Position, infty)* :

  → L'élément de cette position devient le minimum possible (négative possible), puis on peut appliquer *DeleteMin* pour supprimer ce dernier.

  Complexité : **O(log N)**

N.B. : on peut évidemment supprimer l'élément à une position puis "remplir" le trou par les fonctions précédentes.

Exemple : dans les systèmes d'exploitation, un processus arrêté par un utilisateur (terminaison anormale) est supprimé de cette manière de la file des priorités.

Dans des cas similaires, le **TAS ne contient pas directement les données.**

- Comme le cas de hachage
- Un identifiant comme dans les systèmes d'exploitation.

- Si une application a besoin de modifier les valeurs stockes dans un TAS, on utilise une technique spécifique (voir exemple suivant)../..

V.1- Exemple de modification des éléments d'un TAS

Rappel : il existe une relation d'ordre entre les éléments d'un TAS.

- Par défaut, les éléments d'un TAS ne sont pas directement modifiables sauf à procéder à un post-contrôle.
- Une alternative est d'utiliser un TAS contenant des indices (entiers i, j) sur les données stockées ailleurs (soit dans un tableau Data).
- On ne compare pas les indices i avec j (dans le TAS) mais Data[i] avec Data[j].
- Cette technique est préférée pour pouvoir intervenir sur les données d'un TAS
  → (voir plus loin *Dijkstra* et la mise à jour des Distances).
- On dira que le TAS (Heap) est déporté (ou externe ou indirect)

Rappel : en interne, le Tas est géré par les indices 2i et 2i+1

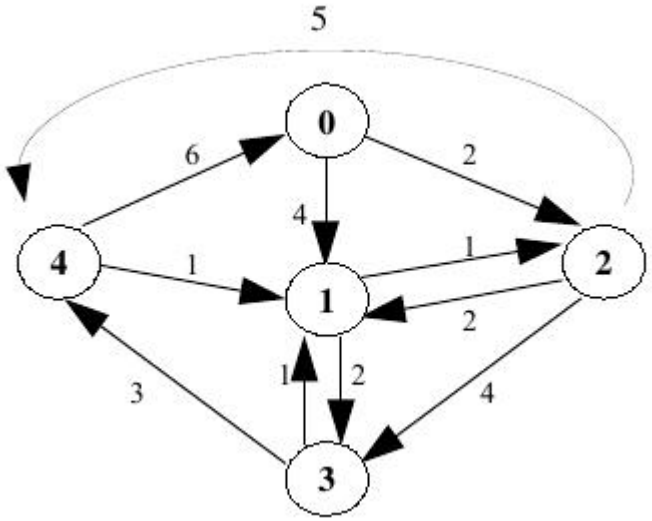
VI- Dijsktra : les plus courts chemins d'un nœud aux autres

- Une version de l'algorithme de Floyd (voir cours 5) où l'on calcule les plus courts chemins depuis un nœud source.

VI.1- Exemple 1

Soit un graphe valué connexe et un nœud D.

Trouver les plus courts chemins depuis D --> tous les autres nœuds du graphe.



Départ = noeud 1
1 → 3 → 4 → 0 distance = 11
1 → 2distance = 1
1 → 3distance = 2
1 → 3 → 4distance = 5

Départ = noeud 0
0 → 1distance = 4
0 → 2distance = 2
0 → 2 → 3distance = 6
0 → 2 → 4distance = 7

- **Méthode** (on utilisera un graphe et éventuellement un TAS) :

– ***Distance****[1..Nb_nœuds]* est un tableau qui contiendra les distances entre le nœud de départ et tous les autres nœuds. Initialisation :

DISTANCE[1..Nb_nœuds]=infini

– ***Trajet****[1..Nb_nœuds]* est un tableau contenant le nœud qui précède chaque nœud dont la distance minimum est calculée.

Du fait de cette fonctionnalité, on appelle ce tableau **coming_from.**

A l'aide de ce tableau, de proche en proche, on pourra reconstituer les trajets (minima) depuis le départ jusqu'à chaque nœud du graphe (v. plus loin).

VI.2- Principe de l'algorithme pour le graphe G=(V,E)

- On part d'un nœud de départ D ∈ V, on note les distances entre ce nœud et ses adjacents dans *Distance*;
- On choisit ensuite le nœud X le plus proche de D (min(Distance[i]), le nœud i non encore traité).
- Pour chaque nœud **Y** adjacent de X de distance **d** de X (c-à-d. $(X,Y)=d \in E$), on recalcule Distance[Y] = min(Distance[Y], Distance[X]+d),

  → C-à-d : sachant qu'on sait déjà aller de **D** à **Y** à une distance actuelle *Distance[Y]* , a-t-on intérêt à aller de **D** à **Y** en passant par **X** ou non ?
- Réitérer jusqu'à la fin (tous les nœuds traités).

☞ La recherche du nœud de distance min peut se faire avec un **TAS** (Heap).

VI.3- Détails de la méthode Dijkstra

```
Pour tout nœud N dans G, DISTANCE[N]=infinie
Pour tout nœud N dans G, Coming_From[N]=N;          // initialisation du  Trajet
Choix d'un nœud de départ N du graphe G;
départ=D
marquer(D);
DISTANCE[D]=0;                                      // de D à D coûte 0 , on a Coming_From[D]=D  ci-dessus  !
Pour tout nœud X adjacents de D dans G
      DISTANCE[X]=poids(D    X);
      Coming_From[X]=D
Fin pour;
Tant que Non tous les nœuds traités
      Trouver le nœud X non-marqué* tel que DISTANCE[X] soit minimale          // voir la remarque*
      marquer(X);
      Pour tout nœud  Y non marqué adjacent de X dans le graphe G
         DISTANCE[Y]=min(DISTANCE[Y], DISTANCE[X] + poids(X->Y));
         Si DISTANCE[Y] est modifiée alors Trajet[Y]=X
      Fin pour;
Fin Tq;
```

(*) Remarque sur **non marqué** : les nœuds déjà traités stockés dans un ensemble S qui grandit depuis {} en y ajoutant chaque nœud (de distance minimum) considéré → Voir le code C/C++ en **Annexe.**

VI.4- Complexité de l'algorithme Dijkstra

- **Sans Heap,** la recherche du nœud de distance min=$O(|V|)$.

- Cette recherche est faite pour chaque nœud complexité=$O(|V|^2)$.

N.B. : le temps de mise à jour des nœuds = constant pour chaque arc : $O(|E|)$

La complexité (sans Heap) de Dijkstra = $O(|E|+|V|^2)$

- Si le graphe est dense, on a $|E|=\Theta(|V|^2)$ (e.g. représentation matricielle)

$O(|E|+|V|^2)=O(|V|^2+|V|^2)=O(2.\ |V|^2)=O(|V|^2)$. ../..

- Si le graphe est peu dense avec $|E|=\Theta(|V|)$, la version sans Heap sera assez lente.

Si on utilise un TAS : la recherche du nœud de distance min = $O(\log |V|)$.

Cette recherche est faite pour chaque nœud complexité=$O(|V|. \log |V|)$.

→ le temps de mise à jour des nœuds = constant pour chaque arc :

$O(|E|) = O(|V|)$.

La complexité (avec Heap) = $O(|E| \log |V|)$

versus $O(|V|^2)$ sans utiliser un Heap.

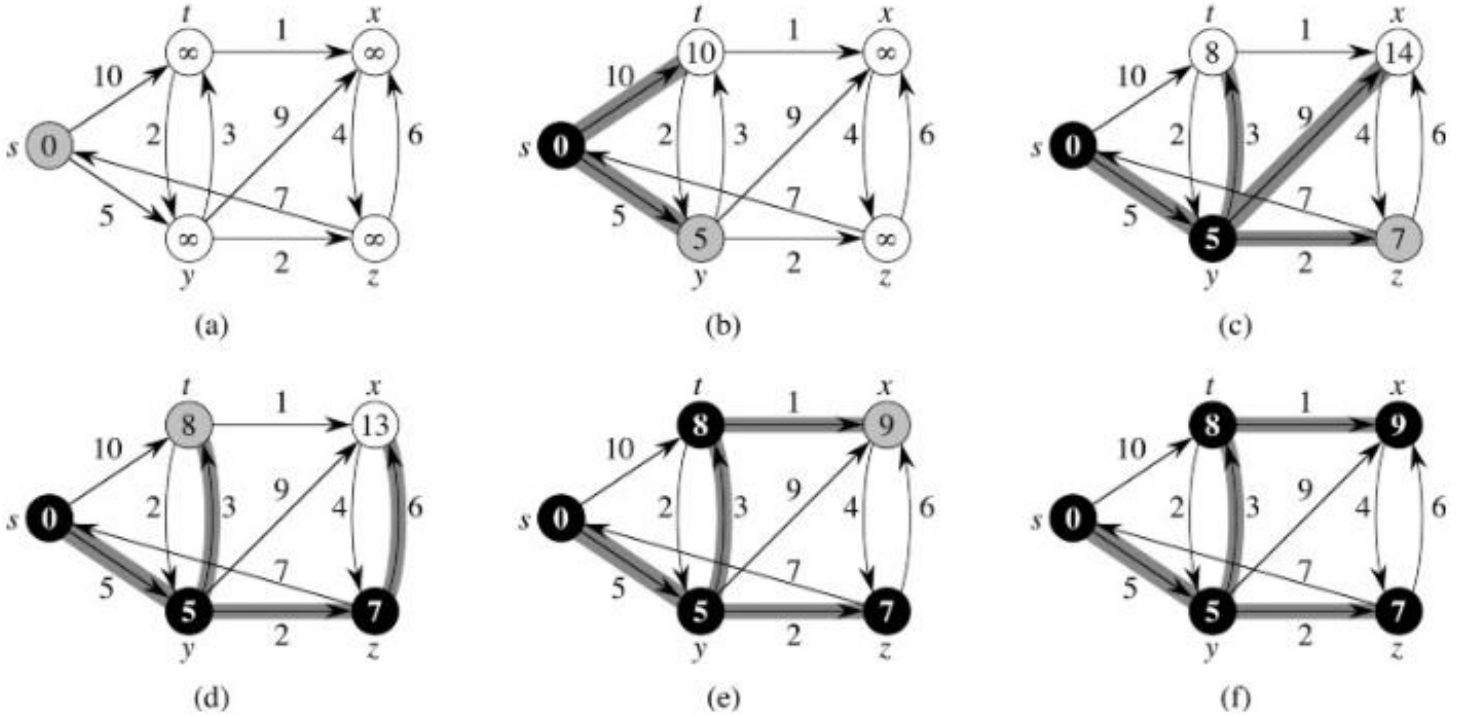
VI.5- Exemple 2

Cet exemple montre les premières étapes (par une méthode différente)

fig. b : on doit choisir entre les voisins de 0 : <0,5> et <0,10> → <0,5> est mieux,

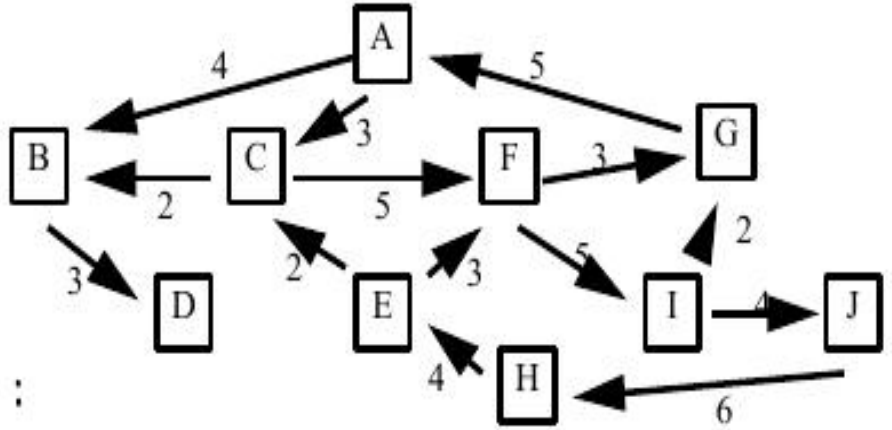
→ 5 sera le prochain nœud dont on examine ses voisins : voisin(5) = <7,8, 14>

fig c : entre <5,7> et <5,8> et <5,14> : <5,8> est mieux ; prochain voisin(5) =7 …



VI.6- Exemple 3 (Dijkstra)

- En partant du nœud A du graphe ci-dessus, on obtient les résultats suivants :



- Le tableau Distance contiendra

A	B	C	D	E	F	G	H	I	J
0	4	3	7	27	8	11	23	13	17

:

- Le tableau des trajets (appelé Coming_From) contiendra :

A A A B H C F J F I

- Par exemple, pour atteindre le nœud F, on a un coût de 8 unités et on passe par C.
- Pour aller à C, on passe par A etc.

VI.7- A propos du tableau Coming_From

On a :

Coming_From[i]= prédécesseur du nœud i (dans le parcours).

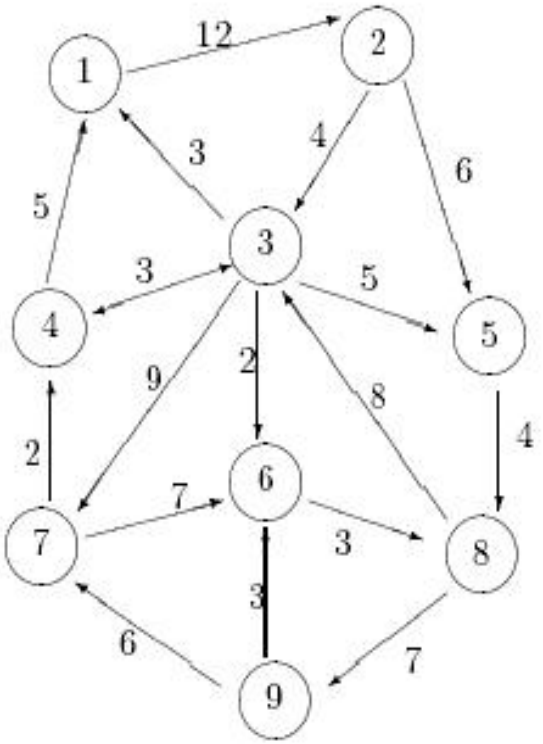
Pour afficher le trajet :

```
Pour tout nœud X
  Répéter
    afficher(X)
    afficher("<-- ")
    X=Coming_From[X]
  Jusqu'à (X=Départ)
  afficher(X);
```

VI.8- Exercice : Dijkstra

Soit le graphe valué orienté ci-dessus.

Appliquer le calcul de Dijkstra en partant du nœud 1.



VII- Algorithme ARM (MST) et graphes non orientés

- Il s'agit de la construction de l'*Arbre de Recouvrement du Poids Minimum* (ARM)

  MST (Minimum Spanning Tree).

- On a un graphe **non orienté**

  --> plus difficile pour les graphes orientés

- L'arbre ARM d'un graphe non orienté connexe G=(V,E) est **un sous graphe G'=(V,E') avec E'⊆ E** (G' est formé de V et de certaines arêtes G) qui connecte tous les nœuds de G avec un **coût total** minimal.

- Un ARM existe si G est **connexe**.

  L'algorithme pourra dire si G n'est pas connexe.

- **Exemples d'application des MST** (minimisation des coûts):

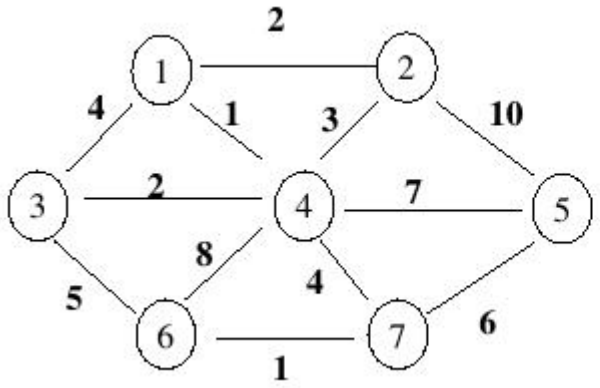
  → Câblage d'un bâtiment,

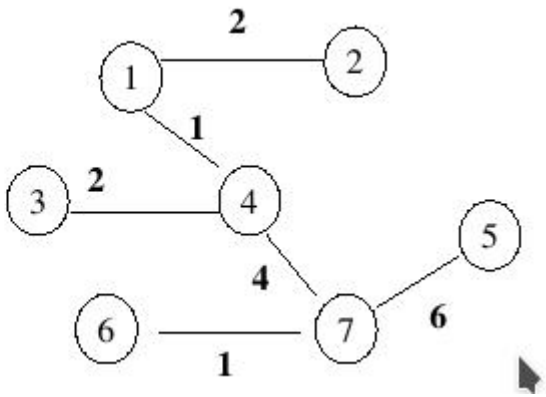
  → Accès aux nœuds d'un réseau,

  → etc.

VII.1- Exemple d'un graphe et du MST

Départ=1





- Pour G=(V,E), le nombre d'arêtes du MST final = |V|-1.

- Le résultat est un arbre car il est acyclique (et ses arêtes supposées orientées depuis le Départ). C'est un arbre de recouvrement car il couvre tous les nœuds; il est minimum car le total des coûts est minimum.

../..

- Le but d'un MST est de minimiser le total des coûts.

- Le MST n'indique rien sur les chemins les plus courts (cf. Dijkstra) entre les nœuds.

Dans l'exemple ci-dessus, entre les nœuds 4 et 5, on peut trouver un meilleur chemin.

VII.2- Propriété d'un MST (ARM)

Un ARM est tel que

- Si une arête **e** qui n'est pas dans *arm* final y est ajoutée, alors on crée un cycle (car l'*arm* doit être minimal).

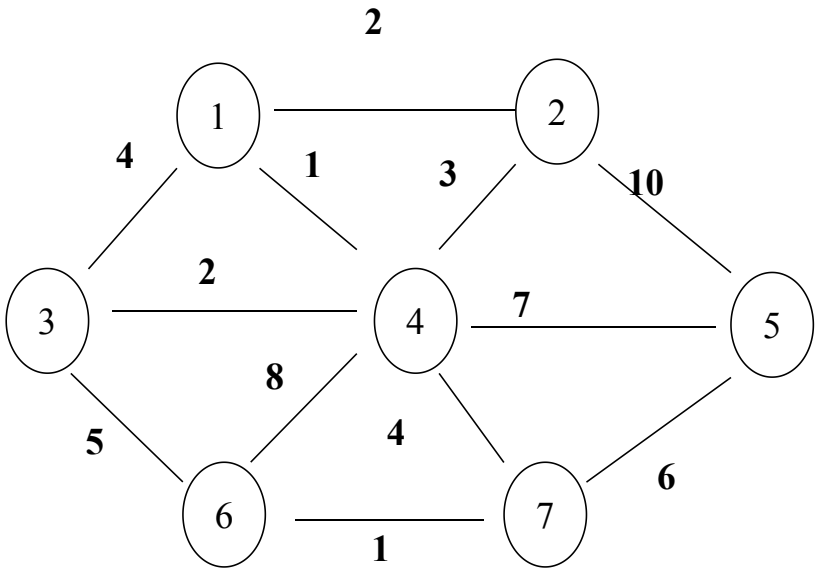
  Le retrait d'une (autre ?) arête du cycle rétablit la propriété du MST.

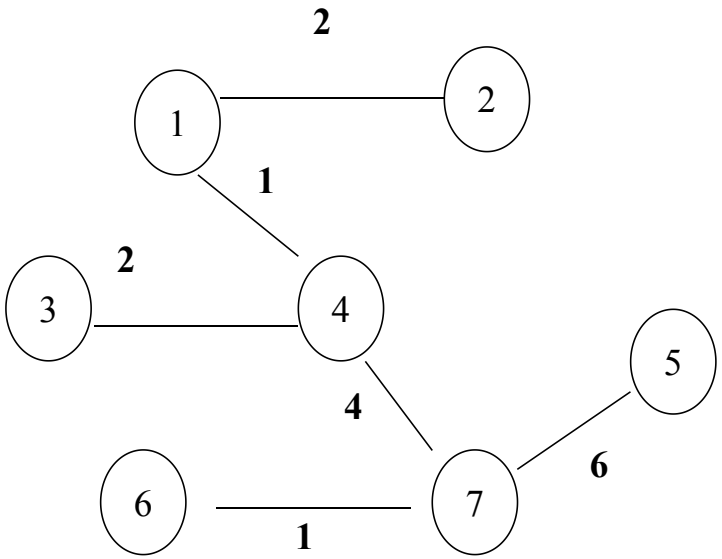
- Le coût du MST sera réduit si **e** est d'un coût moindre que l'arête supprimée.

- Lors de la création d'un MST, si une arête de plus ajoutée est de coût min (et elle évite la création d'un cycle), alors le coût du MST ne peut pas être amélioré car tout remplacement d'arête donnera un coût au moins égal à celui de l'arête min ajoutée.

- Ces propriétés permettent d'utiliser une méthode *greedy* (pas à pas) pour calculer le MST d'un graphe.

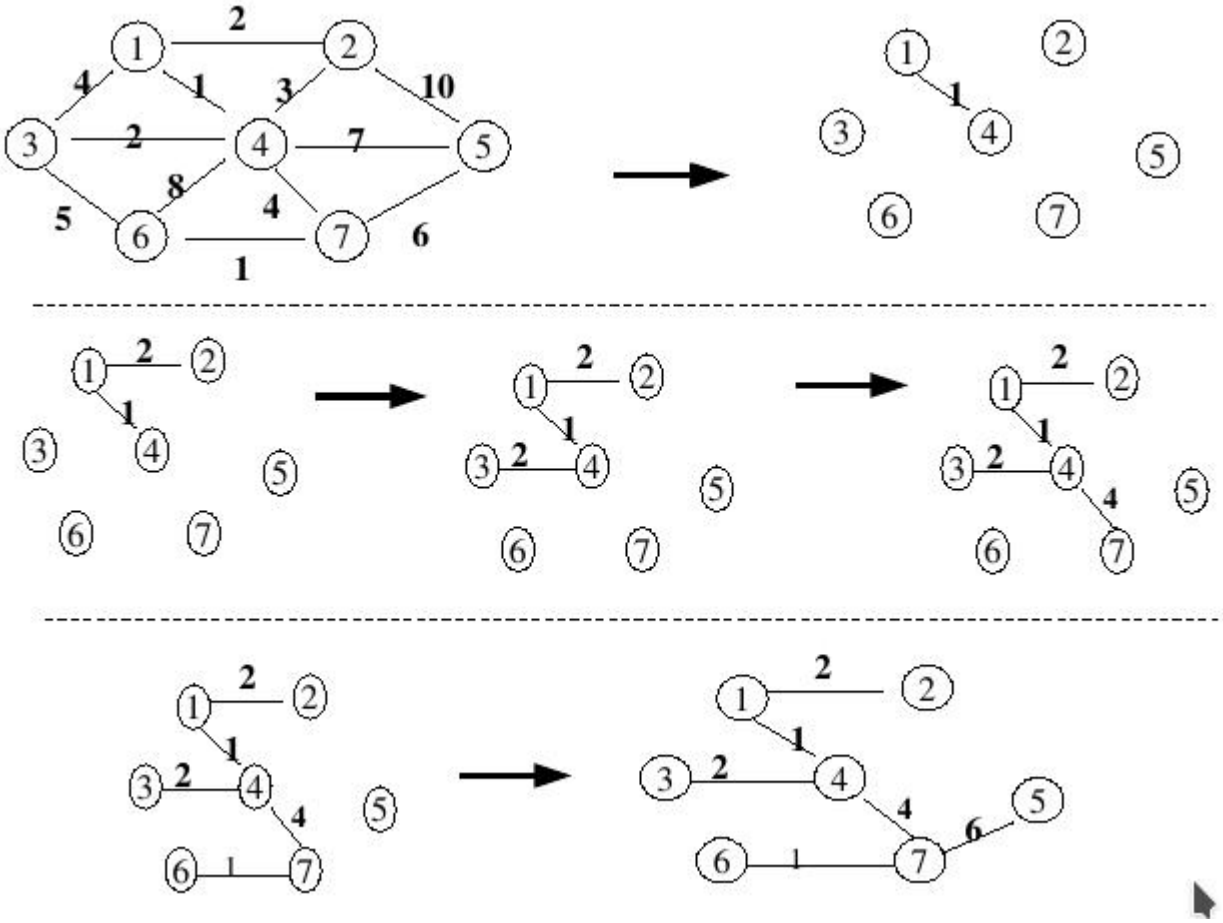
Un exemple





VII.3- Exemple : MST par une stratégie greedy (Algorithme PRIM)

Départ=1, on fait grossir **un même** ensemble de nœuds qui deviendra l'ARM.



- Dans les deux algorithmes suivants, seul le critère de choix des arêtes diffère.

VIII- Algorithme de PRIM

- On fait grossir l'arbre MST petit à petit.
- A chaque étape, un nœud sera associé au MST par l'ajout d'une arête.
- A chaque étape, on a un ensemble de nœuds déjà dans l'arbre.
- L'algorithme trouve alors un nouveau nœud en choisissant l'arête (u, v) de coût minimum (parmi les arêtes du graphe) telle que **u** soit dans le MST et **v** ne le soit pas.

Les étapes (algorithme PRIM) :

- Au début, on connaît le nœud Départ. Chaque étape ajoute un nœud (via une arête).
- Les structures de données peuvent être les mêmes que pour Dijkstra :

  - Un tableau pour représenter les éléments traités (les marqués, appelé ici *Connu*),

  - Un tableau pour les *Distances* et un tableau *Coming-From* (*CF*) pour les trajets.

Les 3 tableaux (état initial) :

	V1	V2	V3	...	V7
Connu	FAUX	FAUX	FAUX	FAUX	FAUX
Distance	infinie	infinie	infinie	infinie	infinie
CF	0	0	0	0	0

La **Distance** contiendra à toute étape et pour chaque nœud Vi le poids de la plus courte arête qui connecte **Vi** à l'ensemble **Connu** (par sa distance à un nœud dans **Connu**).

L'algorithme ressemble à celui de **Dijkstra** sauf pour le traitement de *Distance* où la règle de mise à jour sera :

après le choix d'un nœud *v*, pour tout nœud inconnu **w** adjacent de *v*.

Distance[w]=min(Distance[w] , Poids(w,v))

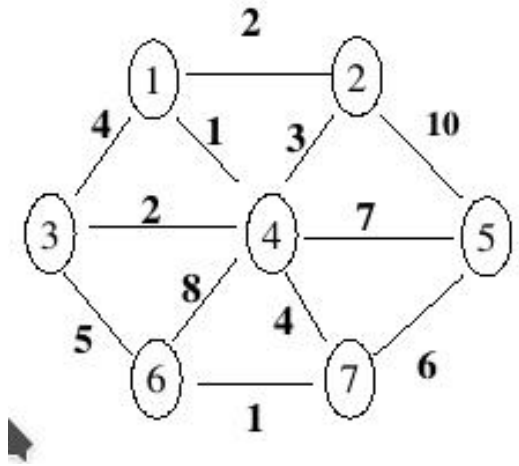
On signale les nœuds dans l'ensemble **Connu** en couleur verte, les choix en jaune.

Initialisation :

Dans l'exemple, V1 est donné (ou choisi random); on met à jour V2, V3 et V4 :

	V1	V2	V3	V4	...	V7
Connu	1	0	0	0	0	0
Distance	0	2	4	1	0	0
CF	0	0	0	0	0	0

Étape 1 : Le nœud suivant sera V4 (distance *minimum*).



- Dans G, tout nœud est adjacent de V4.
- V1 n'est pas touché car il est dans *Connu*.
- V2 sera inchangé car Distance[V2]=2 alors que Poids(V4,V2)=3 (+ MAJ des autres).
- On met l'accent sur Distance.

	V1	V2	V3	V4	V5	V6	V7
Connu	1	0	0	1	0	0	0
Distance	0	2	2	1	7	8	4
CF	0	0	0	0	0	0	0

Étape 2 : Le nœud suivant choisi est V2 (V3 est aussi un candidat).

Le choix de V2 n'affecte pas le tableau.

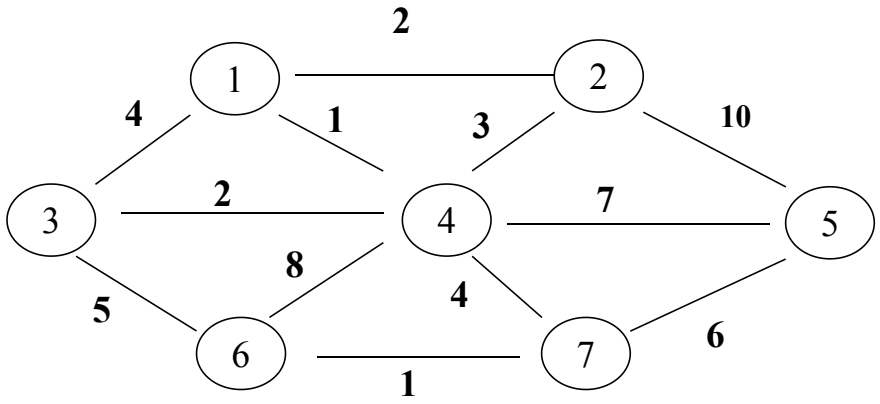
Étape 3 : Le nœud suivant est V3 qui modifie Distance[V6] (passe de 8 à 5).

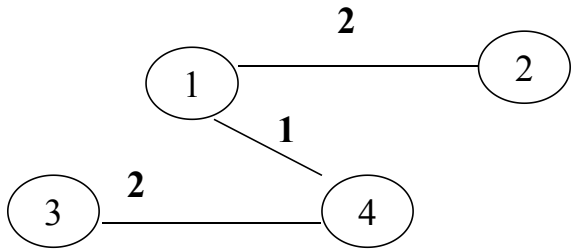
	V1	V2	V3	V4	V5	V6	V7
Connu	1	0	0	1	0	0	0
Distance	0	2	2	1	7	8	4
CF	0	0	0	0	0	0	0

Le tableau devient :

	V1	V2	V3	V4	V5	V6	V7
Connu	1	0	0	1	0	0	0
Distance	0	2	2	1	7	5	4
CF	0	0	0	0	0	0	0

Étape 4 : On choisit ensuite V7 qui modifie la Distance pour V6 et V5.





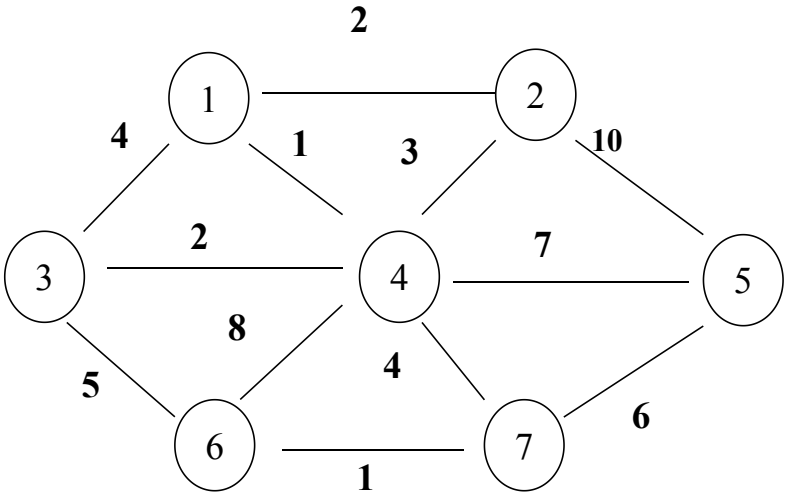
	V1	V2	V3	V4	V5	V6	V7
Connu	1	0	0	1	0	0	0
Distance	0	2	2	1	6	1	4
CF	0	0	0	0	0	0	0

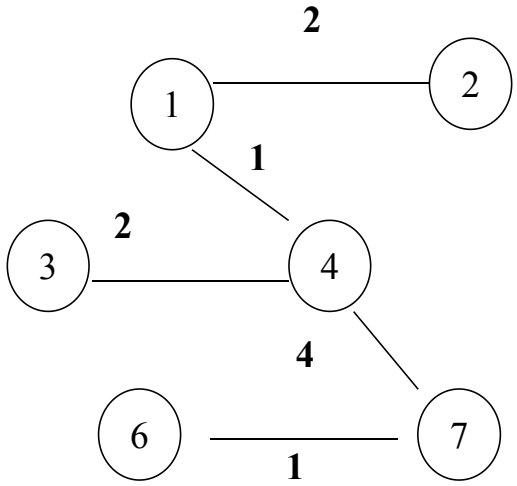
Étape 5 : Puis V6 est choisi ...qui ne modifie pas la distance pour V5

	V1	V2	V3	V4	V5	V6	V7
Connu	1	0	0	1	0	0	0
Distance	0	2	2	1	6	1	4
CF	0	0	0	0	0	0	0

Étape 6 : choix du dernier nœud V5

	V1	V2	V3	V4	V5	V6	V7
Connu	1	0	0	1	0	0	0
Distance	0	2	2	1	6	1	4
CF	0	0	0	0	0	0	0





Le **tableau final sera :**

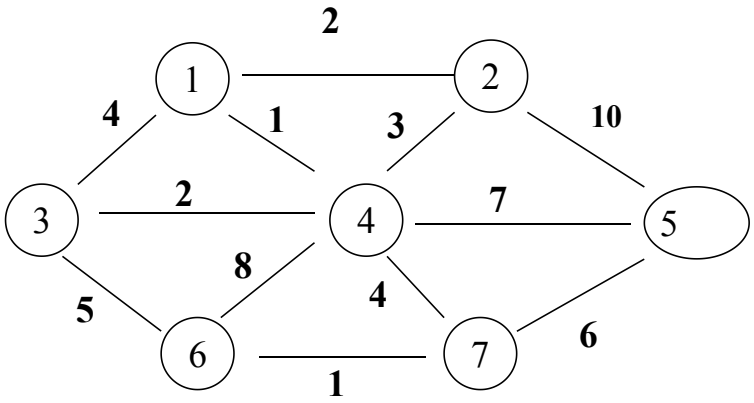
	V1	V2	V3	V4	V5	V6	V7
Connu	1	1	1	1	1	1	1
Distance	0	2	2	1	6	1	4
CF	V1	V1	V4	V1	V7	V7	V4

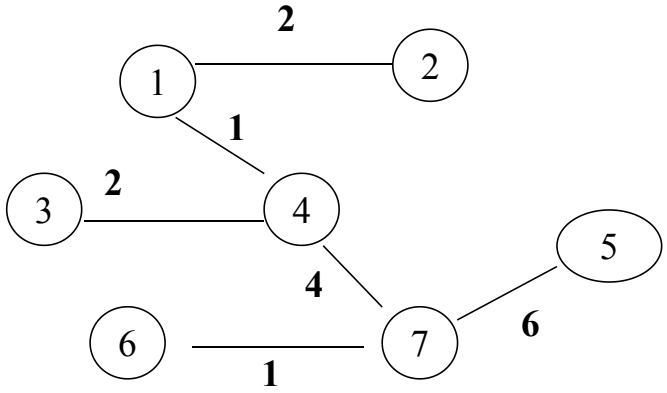
→ Le coût total = 16 (la somme des distances).

N.B. : Au départ, il y a un choix du nœud initial arbitraire (random).

→ Si l'on change de choix initial, le MST peut être différent.

- La complexité est la même que celle de *Dijkstra*.





VIII.1- Algorithme de principe PRIM pour un graphe G=(V, E)

```
Action PRIM( G) :                    // le graphe G = (E, V)
    arm  : ensemble d'arêtes  = {}
    Connu : ensemble de nœuds  = {Déaprt}
    TQ  Connu <> l'ensemble des nœuds V
       Choisir une arête (p, q) de poids minimal dont p  dans Connu mais pas q
       Ajouter  q à Connu
       Ajouter (p, q) à arm                  // et mettre à jour les autres noeuds(dépend de la structure de données)
    Fin TQ
Fin
```

- L'algorithme de PRIM s'applique ici aux graphes **non orientés** :

  chaque nœud va figurer dans la liste des adjacents de ses voisins.

- La complexité de l'exécution=$O(|V^2|)$ sans *Heap*.

- Cette complexité est optimale pour les graphes denses (voir l'algorithme de *Dijkstra*). ../..

- Avec un TAS, la complexité sera $O(|E| \log |V|)$, en part. pour les graphes peu denses.

- **Rappel** : dans un graphe peu dense, $|E|=\Theta(|V|)$.

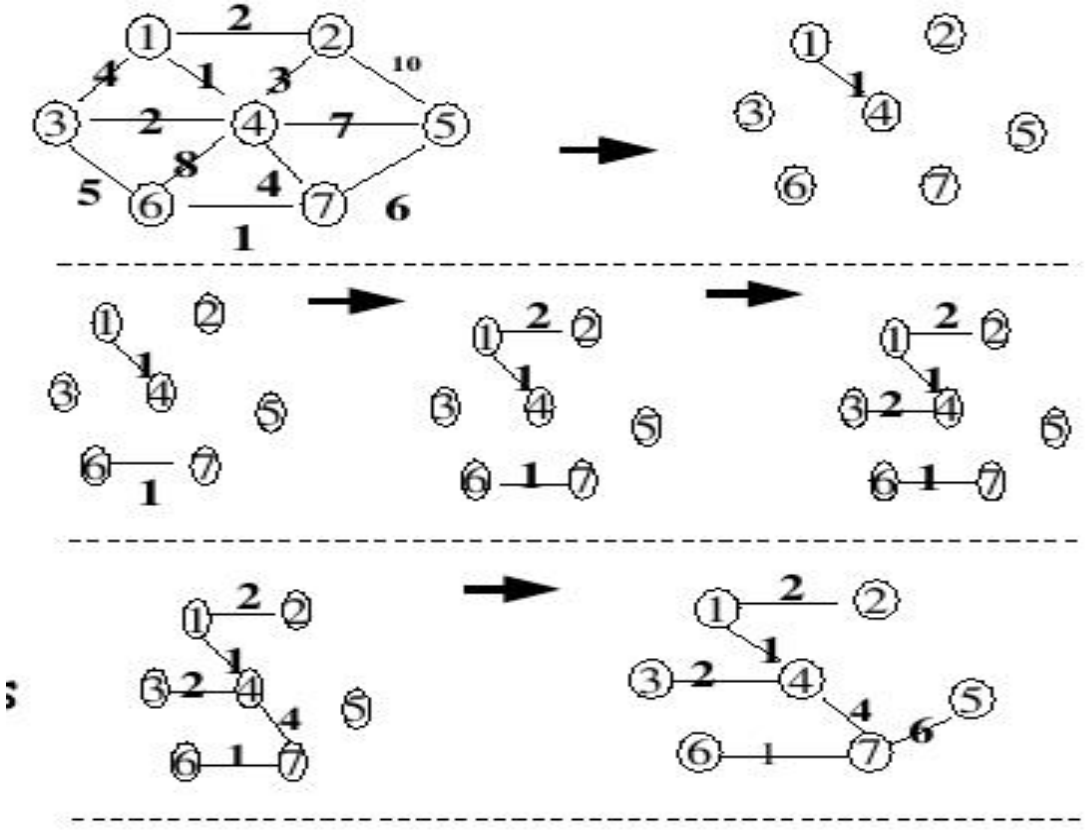
  Avec un Heap, il y aura **|E|** DeleteMin (suppression du minimum) à effectuer ($\simeq|V|$).

IX- Algorithme de Kruskal

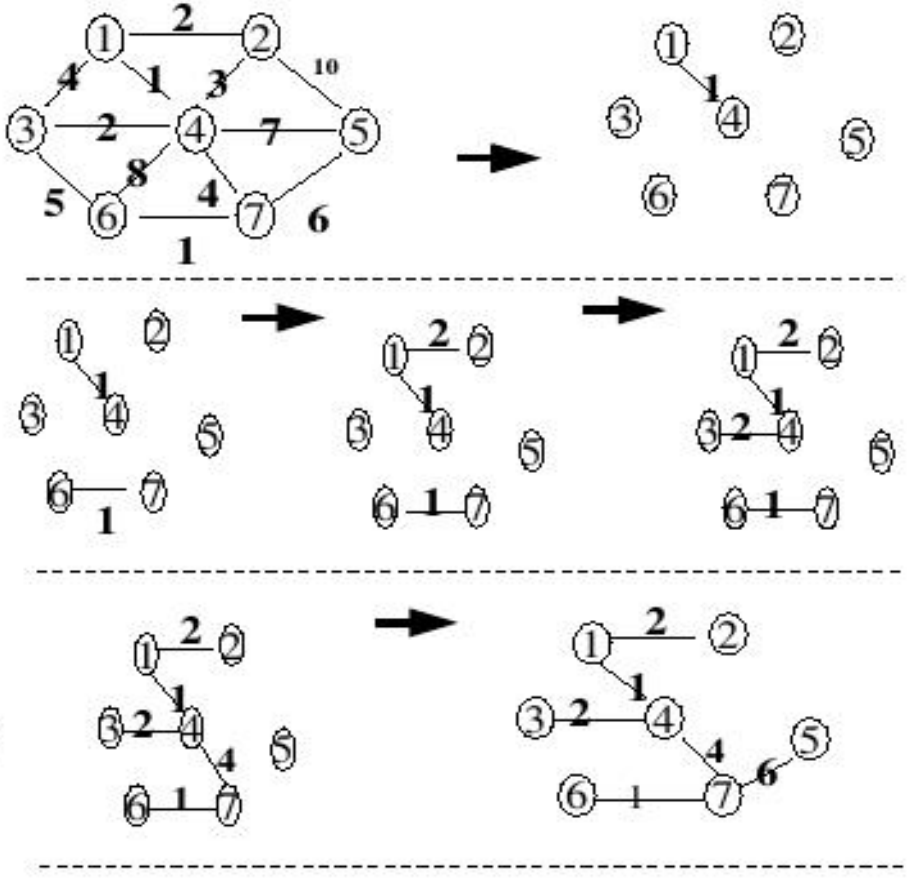
- L'algorithme de PRIM fait *grossir* **un même ensemble** de nœuds (*CONNUS*).
- L'algorithme (pas à pas) de *Kruskal* peut créer **plusieurs ensembles** de nœuds.
- On choisit les arêtes de la plus petite à la plus grande (coût) et on accepte une arête si elle ne crée pas un circuit.

Exemple :

- Init : ∀ nœud est dans son propre ensemble (arbre).



- **Si** *u* et *v* sont dans un même ensemble, on rejette l'arête *(u,v)* car ces nœuds sont déjà (indirectement) connectés et l'ajout de *(u,v)* produira un cycle.



Sinon, l'arête (*u, v)* est acceptée et on fait l'union des 2 ensembles auxquels *u* et *v* appartiennent.

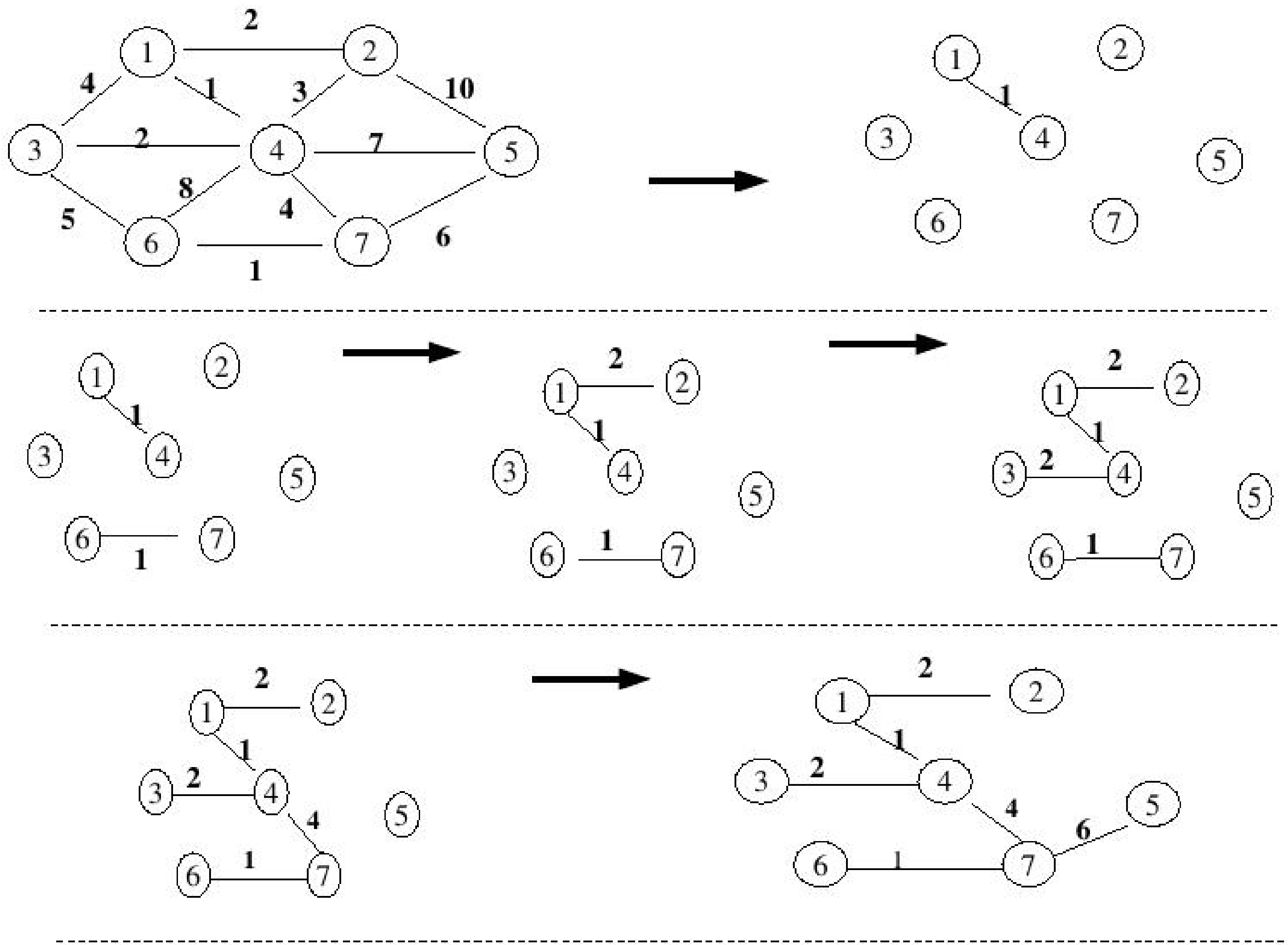
;

- **L'invariante de cet algorithme :**

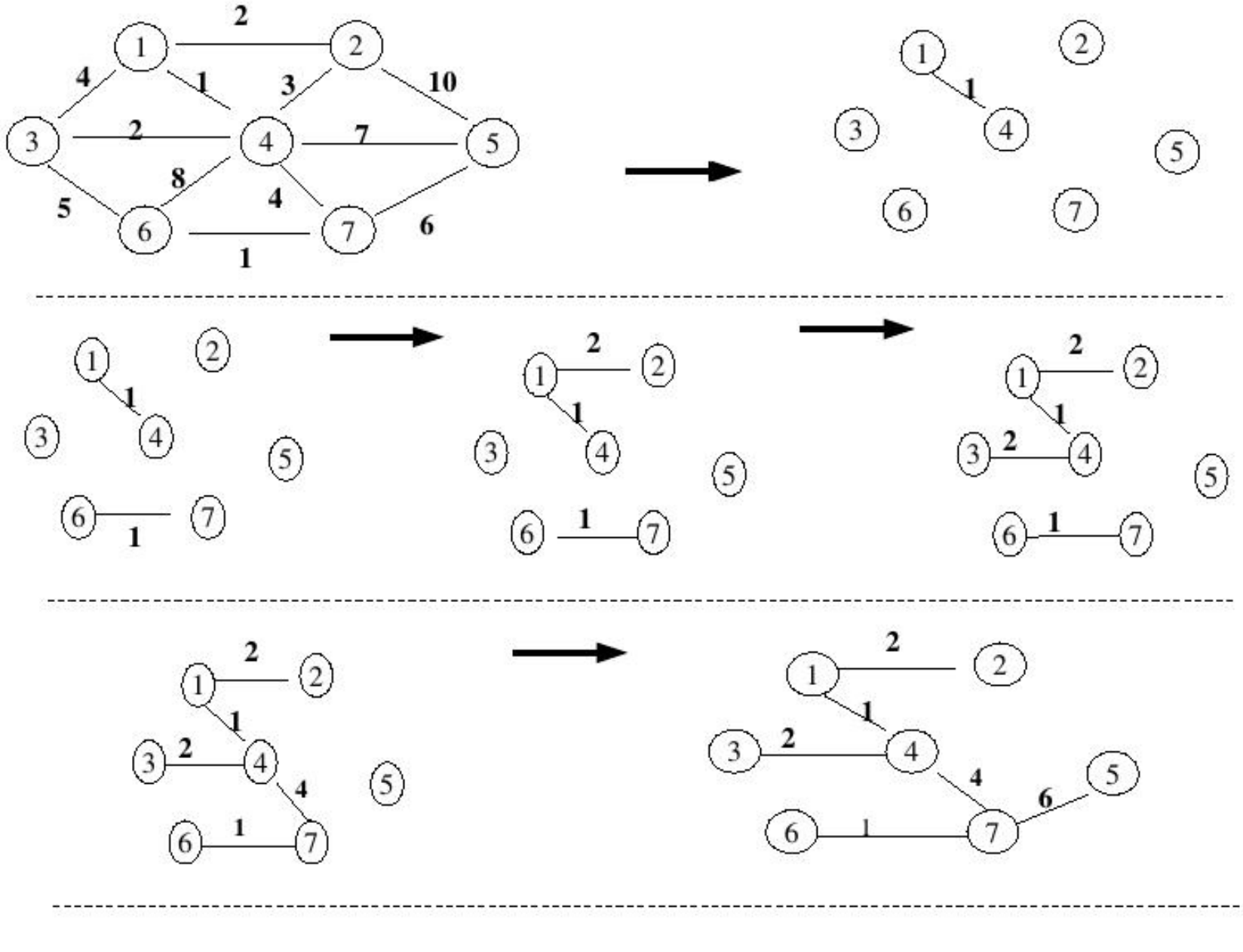
à tout instant, deux nœuds appartiennent à un même ensemble s'ils sont connectés dans l'arbre de recouvrement final.

- Le principe décrit maintient l'invariante de l'ensemble car quand une arête *(u,v)* est ajoutée à un ensemble, si un nœud **p** était connecté à **u** et **q** à **v**, après l'ajout de (u,v), **q** et **p** sont connectés et appartiendront au même ensemble.

- L'algorithme de **Kruskal** maintient une forêt (collection d'arbres).
  - Au départ, il y a |V| arbres d'un seul nœud chacun.
  - L'ajout d'une arête fusionne deux arbres en un seul.
  - A la fin de l'algorithme, il y aura un seul arbre qui est le MST.
  - Pour savoir si l'on accepte une arête *(u,v)* à une étape, on utilise l'algorithme **Union-Find** qui permet de fusionner deux arbres (si *(u,v)* acceptée).



Détails :


1
2
3
4
5
6
7
2
4
1
3
10
2
7
8
4
5
6
1
1
2
3
4
5
6
7
1
2
3
4
5
6
7
1
1
2
3
4
5
6
7
1
2
1
1
2
3
2
4
5
6
7
1
4
2
1
2
3
2
4
5
6
7
1
4
6

IX.1- L'algorithme de principe de Kruskal

```
Fonction kruskal(G)  :                  // Graphe G = (V, E)
     AretesAcceptees=0;                 // nbr d'arêtes traitées
  DisjSet S(nb_nœuds);                  // un tableau d'ensembles disjoints
  priority_queue h                      // Sera de taille nb_aretes de G (|E|)
  nœud U, V; Arete e;
  Set Uset, Vset;

     // Créer  le tableau des données puis de transformer ce tableau en un Heap
  Tas h = Créer un TAS  à partir de l'ensemble des arêtes selon leur poids

 TQ  (AretesAcceptees <  |V| – 1) :
        e= h.DeleteMin();               // retrait d'une arête e= (u,v) de coût min
        Uset=find(S, u);                // L'ensemble auquel appartient u
        Vset=find(S, v);                // L'ensemble auquel appartient v
        Si (Uset != Vset)   :           // On accepte (u,v)
           AretesAcceptees++;
           unionSets(S, Uset, Vset); // Faire l'union de User et de Vset dans S
     Fin TQ
     renvoyer S
Fin  kruskal
```

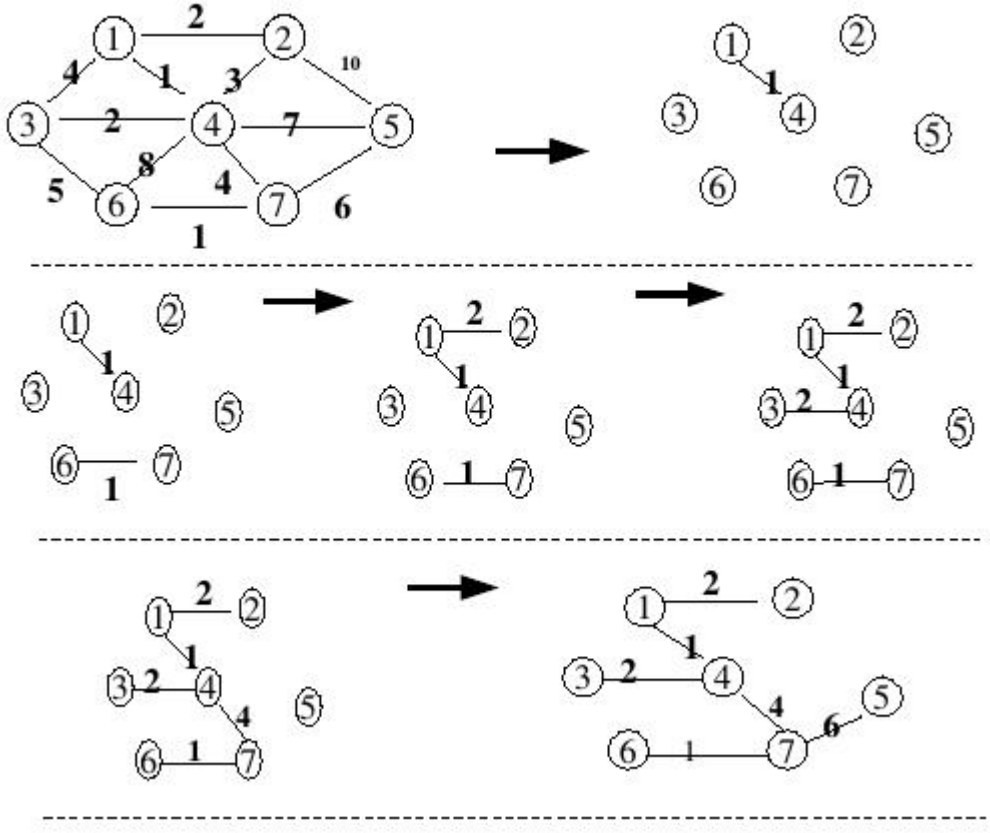
NB : avec *priority_queue* , la création du Heap est faite plus simplement.

IX.2- Application à un exemple

Appliqué à l'exemple précédent, on traite les arêtes dans l'ordre donné ci-dessous :

Rappel : au départ, autant de singletons que de nœuds.

Arête	Poids	Action
(V1,V4)	1	Fusion, {1,4}
(V6, V7)	1	Fusion, {6,7}
(V1,V2)	2	Fusion, {1,2,4}
(V3,V4)	2	Fusion, {1,2,3,4}
(V2,V4)	3	Non :
		V2 et V4 déjà dans le même arbre
(V1,V3)	4	Non : idem
(V4,V7)	4	Fusion, {1,2,3,4,6,7}
		V4 et V7 sont dans deux ensembles ≠
(V3,V6)	5	Non
(V5,V7)	6	OK : {1,2,3,4,5,6,7}



Remarques (et TAS) :

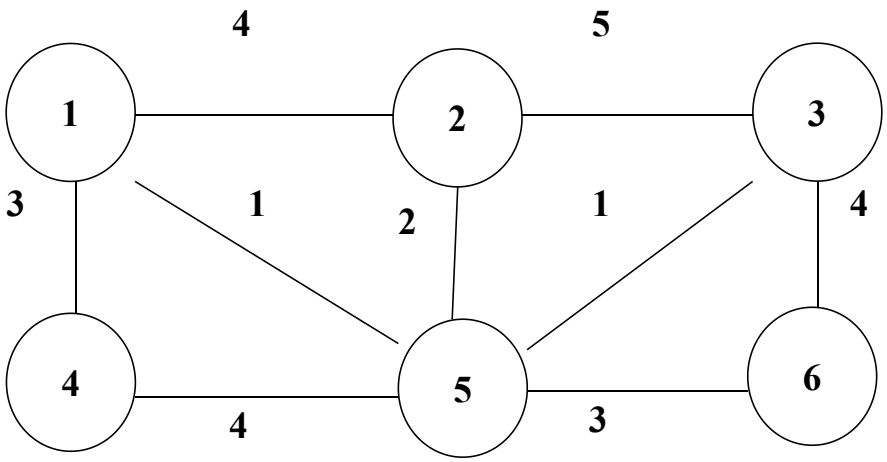
Dans l'algorithme de **Kruskal**, les arêtes peuvent être ordonnées (efficacité).

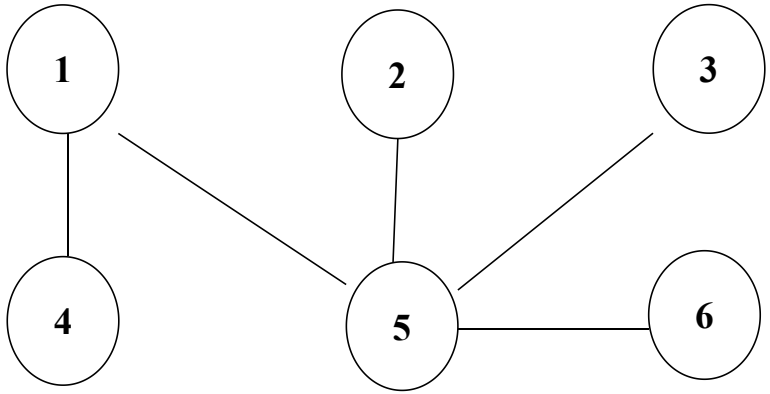
→ Si l'ensemble des arêtes est non ordonné, il est possible que toutes les arêtes doivent être examinées et essayées.

→ Par exemple, si l'on ajoute à l'exemple précédent une arête supplémentaire (V5, V8) de poids 100, toutes les arêtes devront être essayées.

→ Un **Heap** sera utile (même si en général, cet algorithme examine peu d'arêtes).

IX.3- Un autre exemple(avec TAS)





IX.3.1- La complexité de Kruskal avec un Heap

- La cas pire = O(|E| log |E|) dominée par les opérations du *Heap*.

  Rappel : *DeleteMin* est **O(log |E|)**.

- On note que si $|E| = O(|V|^2)$ dans un graphe dense, la complexité sera

  O(|E| log |V|)

  car $\log |E| = \log |V|^2 = 2 \log |V|$

- Mais en réalité et dans la plupart des cas, l'algorithme est plus rapide que cette borne.

N.B. : la représentation des ensembles, Union-Find, ... disponibles sous Python

IX.4- Exercice (pour Prim et Kruskal)

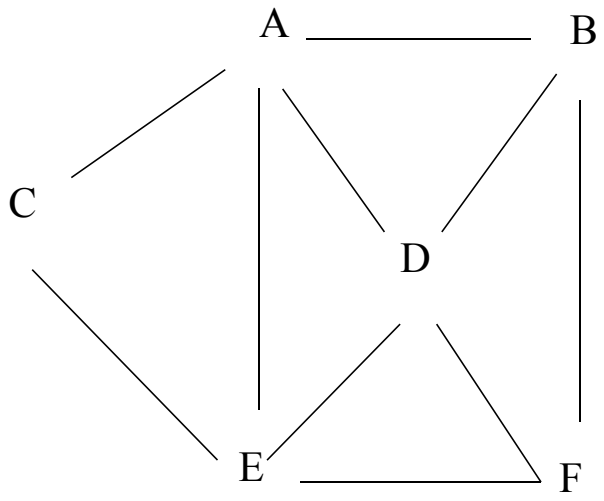
Le graphe et sa solution pour les deux méthodes.

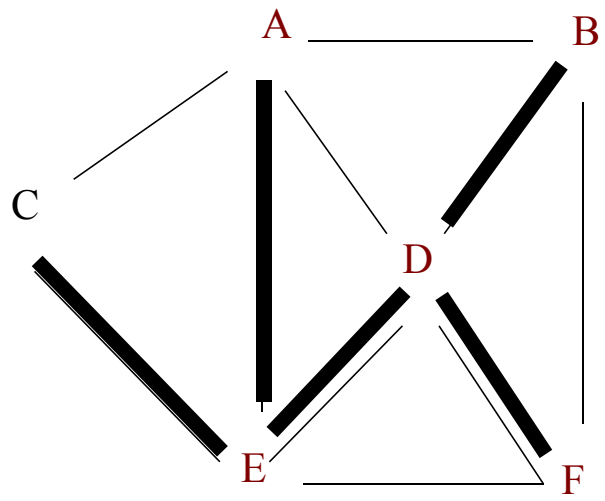
Préciser l'ordre d'ajout des arêtes pour chacune des deux méthodes.

Rappels :

La complexité (avec un TAS) : **O(|E| log |V|)**

Création du TAS : O(|V|), retrait des minima : **O(|V| log |V|)**





X- Algorithme de Baruvka

- Les algorithmes vus précédemment sont *greedy* et peuvent utiliser un Heap pour trouver le minimum de poids.
- Il y un algorithme ancien mais assez simple et plus efficace pouvant utiliser un Heap

Principe de l'algorithme de Baruvka :

Graphe *G=(E,V)* et F un sous graphe de G (forêt) contenant initialement les nœuds (singletons) de G.

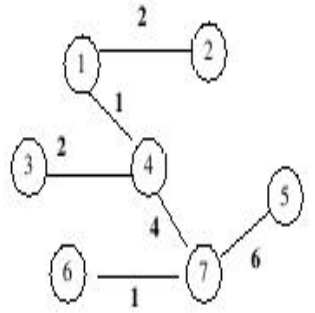
Tant que F a moins de |V| arêtes :

Pour toute composante connexe Ci de F : // connexe dans G, Ci est un ensemble de noeuds

Trouver **e=(u, v)**$\in$ **E** de poids minimum telle que v $\in$Ci, u $\notin$Ci

// l'arête **e** est liée à **Ci** par le nœud **v**

Ajouter e à F (sauf si elle y est déjà)

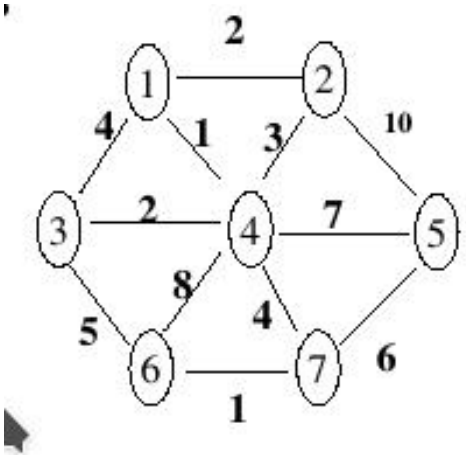


Remarques (algorithme rappelé) :

Tant que F a moins de |V| arêtes :
  Pour toute composante connexe Ci de F : // connexe dans G, Ci est un ensemble de noeuds
    Trouver **e=(u, v)**∈ **E** de poids minimum telle que v ∈Ci, u ∉Ci // l'arête **e** est liée à **Ci** par le nœud **v**
    Ajouter e à F (sauf si elle y est déjà)

- L'ajout de **e** à **F** est O(1); la recherche des C_i composantes connexes est O(N) (recherche en largeur d'abord).
- Marquer un nœud qui va appartenir à une nouvelle composante
- Identifier **e** min adjacente de C_i = recherche dans les adjacents pour les nœuds de C_i.
- Dans la boucle "toute composante connexe C_i de F " :

  Si on choisit toujours la même composante connexe (e.g. C_1)



L'algorithme devient l'algorithme PRIM (en un peu plus compliqué !).

Exécution en parallèle possible de l'algorithme :

- Initialiser chaque nœud par son propre numéro de composante connexe.
- Lancer un parcours en profondeur et changer le numéro de composante connexe du 1er nœud atteint par le numéro de la racine (départ) en prenant l'arête la plus courte, etc...
- L'algorithme ci-dessus devient :

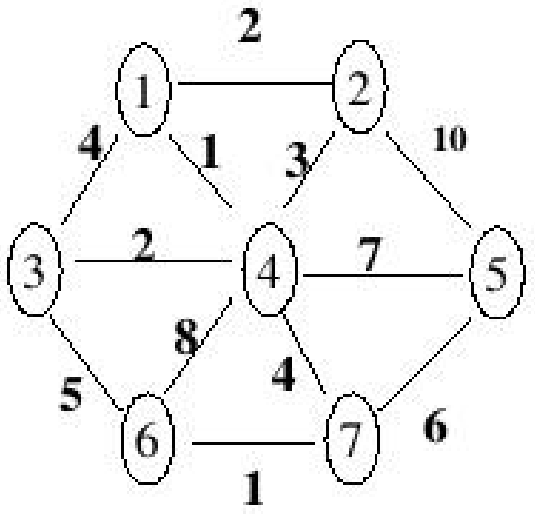
```
Initialiser la Forêt F à chaque nœud de V
TQ nb_arêtes != |V|-1 :
    Pour toute C composante connexe de F        // connexe dans G
        Trouver e=(u,v)∈ E de poids minimum telle que v ∈C, u ∉C
        Ajouter e à C    // modification (parallèle)
    Fin Pour
Fin TQ
```

N.B. : Dans la boucle "Pour toute C composante connexe de F", la forêt F se modifie à chaque pas.

Déroulement pour le graphe :

Composantes pendant les itérations (une exécution séquentielle):

- C1={1}, … , C7={7}



Quelques itérations : choix d'un ensemble Ci

- En **noir** : **e1**={1,4}, **e2**={6,7},

  e3={1, 2}, e4={3,4},

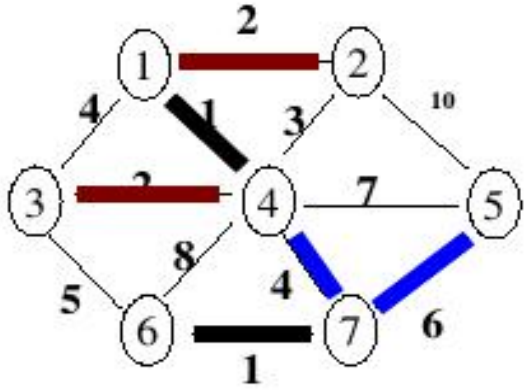
  e5={2,4}, e6={4,7}, e7={1,3}, ...

- En rouge (ignorer les bleus) :

  e11={1,4,2}, e21={6, 7, 4}, e31={2,1,4},

  e41={3, 4, 1}, e51={4,1,3},

  e61 = {5, 7, 6}, e71={7, 6, 4}

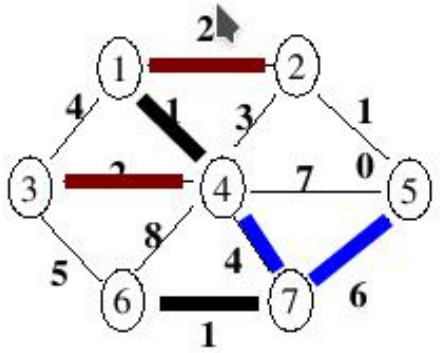


On supprime les doublons (la forêt est un ensemble sans doublon)

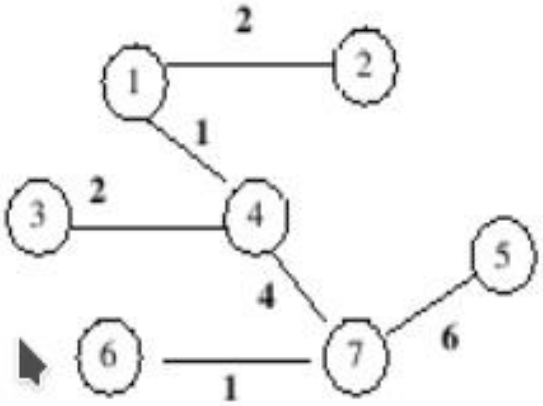
- **en bleu**

  e21={1, 4, 2, 3}, e22=e12,

  e23={5,7,6, 4}, e24={4,6,7,1},.....



A chaque itération, on a choisit l'arête minimale partant de la composante connexe et on supprime les doublons et cycles.

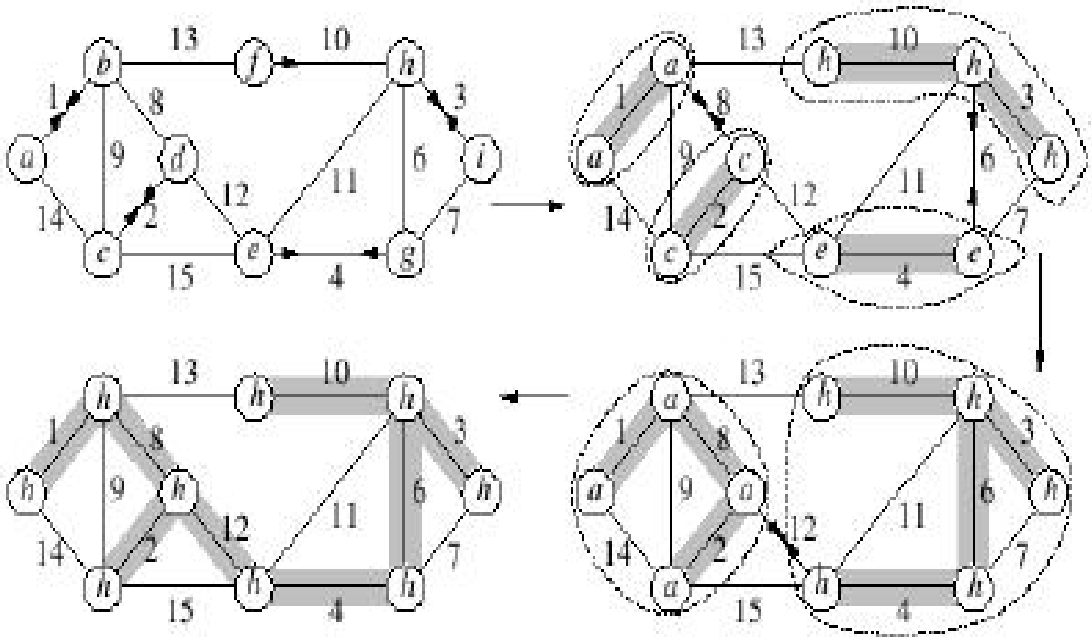


- Le résultat final est le même MST que les autres méthodes.
- L'exécution **parallèle** possible,
- L'utilisation de **Heap** possible (conseillée) pour le choix de l'arête minimale reliant une composante aux autres.

X.1- Un Exemple (Baruvka)

N.B. : les nœuds signalés par la même lettre désignent le même *cluster*.

- L'exécution est en parallèle.
- Ici, il n'y a pas de **Heap** et la méthode est parallèle.



X.2- La complexité de l'algorithme Baruvka

- Soit le graphe G = (E, V)

  A chaque itération (recherche de fusion des nœuds par une arête minimale) :

  Le nombre des arêtes est divisé par deux

  Il y aura donc (log |E|) itérations.

  Complexité de Baruvka = O (|V] log |E|)

XI- Comparaison des 3 méthodes

- Les 3 algorithmes ont la même complexité au cas pire.
- La structure de données est différente pour chaque algorithme
- **Il n'y a pas de gagnant**, sauf pour la possibilité d'exécution **parallèle** (*Baruvka*).

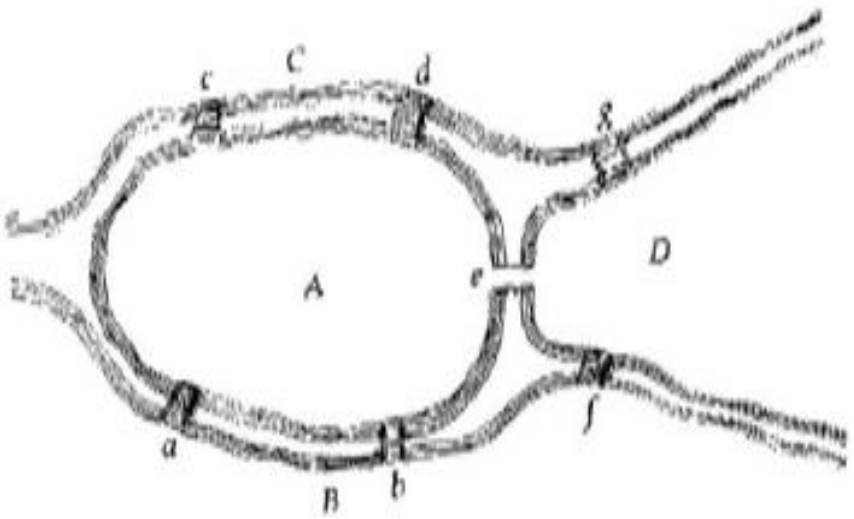
XII- Complément : Problème des réseaux de flux

Les algorithmes de ce chapitre relèvent d'un cadre général sur la théorie des graphes :

→ *network flow problem*.

XII.1- Historique

- Ville de Koningsberg (Allemagne) avec une île (au milieu) et des ponts.
- Les habitants s'amusaient à essayer de partir de n'importe quel point, franchir une seule fois chaque pont et revenir au point de départ.

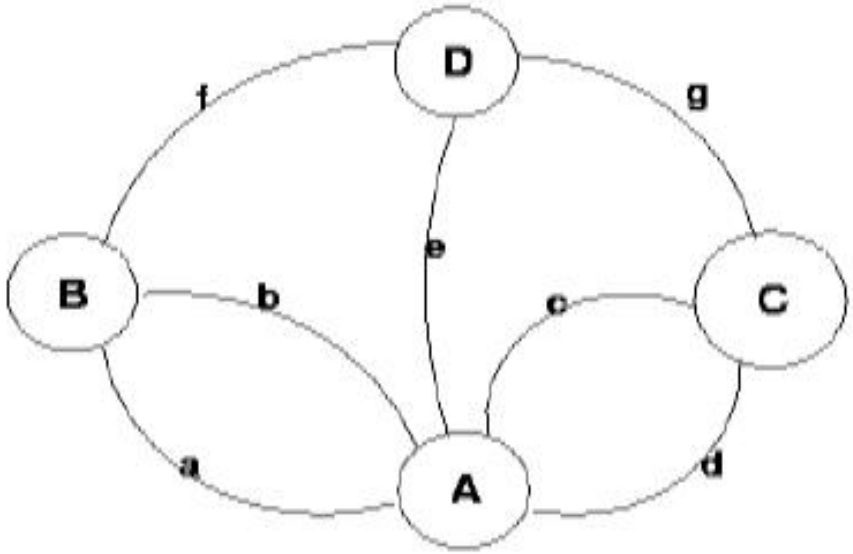


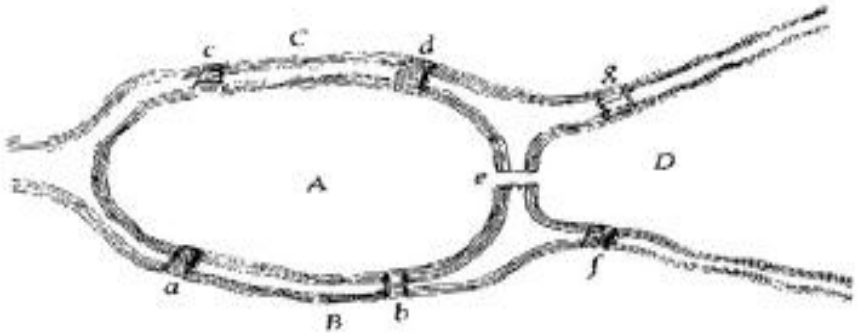
- Personne n'a réussi, jusqu'en 1736 où le Mathématicien Suisse *Leonhard Euler* a prouvé qu'il n'y avait pas de solution à ce problème.
- *Euler* a présenté le problème sous forme d'un graphe où les régions (quartiers) de l'île sont *A,B,C,D* et les ponts (appelés *arêtes*) *a,b,c,d,e,f,g* qui connectent ces quartiers. → C'est la naissance du problème de **réseaux de flux**.

- Le problème est présenté sous forme de graphe :

  les arcs = les ponts et

  les nœuds les quartiers de la ville (de l'île).





XII.2- Classe des problèmes de réseaux de flux

- Le problème du *Postier Chinois*.
- Il s'agit de trouver la tournée avec un minimum de répétition (éviter repasser par la même rue / point).

XII.3- Variantes du Postier

- Beaucoup d'autres problèmes sont des variantes du problème du *Postier* :

  Ramassage des ordures, Recherche du chemin le moins encombré, Trafic sur les auto-routes (ou autres réseaux), Ramassage scolaire, Inspection des lignes de transmission, Livraison de marchandises, etc.

  → TSP (voyageur de commerce)

Ces problèmes sont des variantes du problème du Postier.

XII.4- Variantes du problème de Flux

- Outre le problème du *Postier* (qui n'en est qu'un) et ses variantes, on a :

- Problème de transmission d'information, problème de transport de personnes,

- Distribution de marchandise (entrepôts, ...), Distribution de marchandise et d'énergie, etc.

- Les utilitaires relevant du Téléphone, Internet, Câble et services Télévision, nos déplacements (sur les routes, train, avion, etc).

Époque moderne : outils de résolution de problèmes logistiques.

XII.5- Caractéristiques du problème de Flux

Le problème du réseau de flux est caractérisé par :

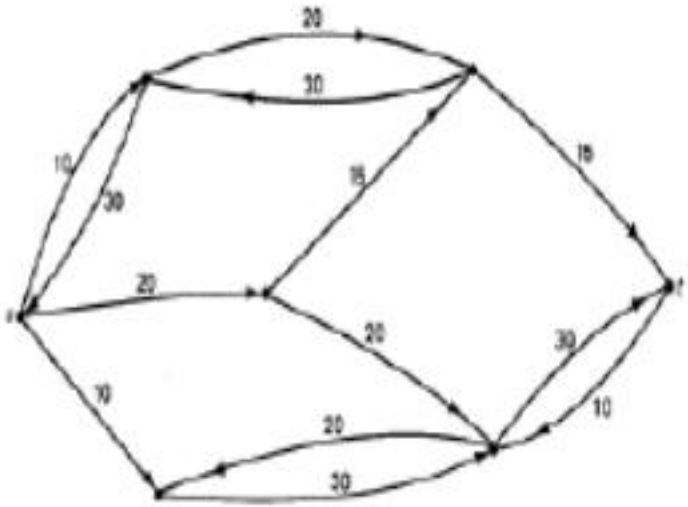
- **Visuel** : un contenu visuel (diagramme, graphe) facilement compréhensible
- **Flexibilité** : les réseaux similaires doivent pouvoir être utilisés en :

  sociologie (communication), fiscalité, politique, ...
- **Calculabilité** (ré-solvabilité) : ils existent des algorithmes efficaces.

Parmi les problèmes les plus importants (en termes de liens entre 2 nœuds) :

- Dijkstra (les chemins les plus court).
- Minimisation du coût de distribution (MST et ses variantes)
- Maximisation de flux (et ses variantes – voir ci-dessous).

Un exemple de réseau de flux (s : source, t=terme) :



→ différentes solutions (et méthodes)

→ différentes complexités.

Exemples d'algorithmes :

Ford et Fulkerson, Edmonds et Karp (**$O(E^2V)$**),

Dinic (**$O(E.V^2)$**), *Karazanov* (**$O(V^3)$**), etc.

L'un des plus récents est *Goldberg et Tarjan* (complexité **$O(E.V.\ \log(V^2/E))$**

→ une des meilleures complexités.

N.B. : l'algorithme de *Ford et Fulkerson* légèrement modifié (**$(O(V^3))$**) est l'un des plus populaires (connu sous le nom MPM).

XII.6- Définition de réseau de flux

Le flux dans un réseau est une fonction **f** qui attribue un nombre réel à chaque arc telle que :

- pour tout arc **e**, $0 \leq f(e) \leq capacité(e)$
- pour tout nœud (sauf *source* et *terme*) **V**

$$\sum_{Init(e)=V} f(e) = \sum_{Terme(e)=V} f(e)$$ *Tout ce qui arrive à V sort de V*

La 2e condition = **le principe de préservation** du flux d'un nœud V

→ comme dans la loi de *Kirchhoff* en électricité.

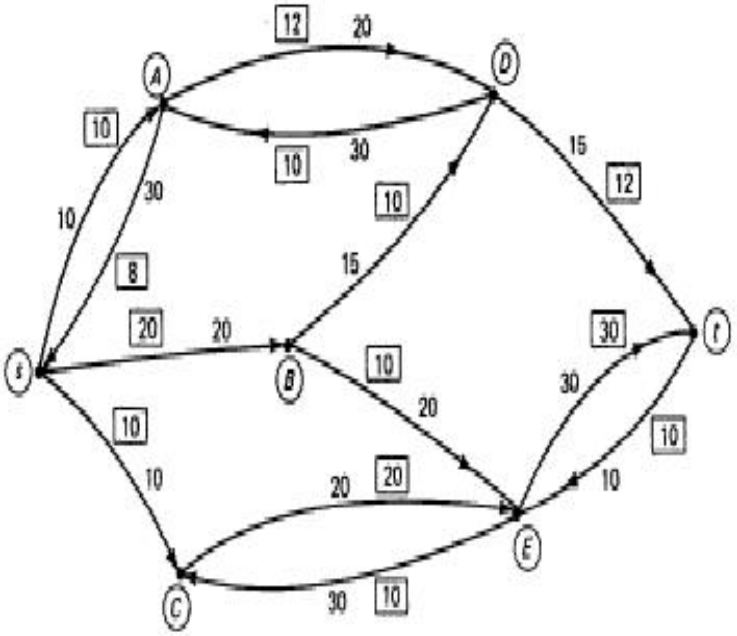
XII.7- Une instance d'optimisation de Flux

- Circulation (P. Ex.) du **gaz** circulant dans les tuyaux (représentés par les arcs).
- Le but : connaître **le maximum** que ce réseau peut transporter (entre S et T).

→ Dans la solution ci contre, maximum = 32.

→ Les valeurs dans les **carrés** :

les quantités qui circulent sur les arcs.



Important :

Le principe de préservation précédent s'applique à tous les nœuds sauf la source (S) et le terme (T).

→ Le résidu (Entrées – Sorties) de chaque nœud (sauf s et t) doit être nul.

→ Pour **s** et **t**, ce résidu est en général non nul est appelée la valeur du flux.

→ Elle est = 32 dans la solution ci-dessus

XII.8- Un autre exemple

- La solution précédente n'est pas le résultat d'un algorithme totalement efficace.
- Pour le même réseau, un algorithme à base de la notion *d'augmentation* tel que l'algorithme de *Ford* et *Fulkerson* donne de meilleurs résultats.

L'idée de l'algorithme de *Ford et Fulkerson* :

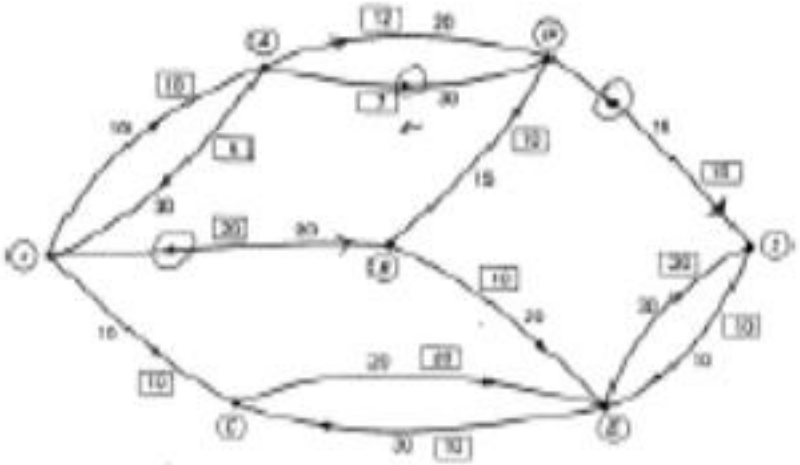
- On commence par une *valeur quelconque* de flux (0 possible).
- On cherche ensuite un chemin d'*augmentation de flux* :

  un chemin de S → T qui alimente une quantité supérieure à la valeur précédente.
- Réitérer jusqu'à plus d'amélioration.

- Dans la solution suivante (algorithme *Ford* et *Fulkerson*), la valeur du flux est **35**.
- *Les valeurs dans les carrés* : la quantité qui circule effectivement sur cet arc (quantité forcément inférieure à la capacité de l'arc).

XII.9- Addendum : quelques explications

- La figure montre un chemin d'augmentation de l'exemple précédent.



- Un arc peut être sélectionné dans un chemin d'augmentation pour **2 raisons** :

1- La direction de l'arc est **cohérente** avec la direction S

→ T et la valeur du flux sur cet arc est <u>en dessous</u> de la capacité de l'arc.

2- La direction de l'arc est opposée à S

→ T et la valeur du flux sur cet arc est <u>strictement positive</u>.

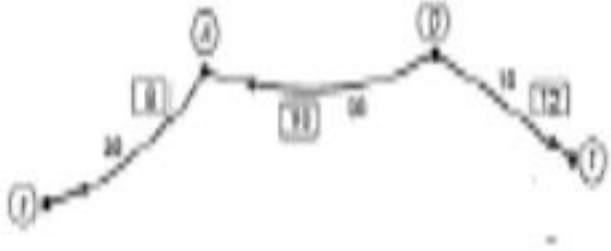
- Donc, pour un chemin d'augmentation, tous les arcs orientés de manière cohérente peuvent être augmentés, et ceux incohérents diminués :

  → le résultat sera une augmentation générale du flux S → T.

N.B. : la notion de **cohérence** est fortement liée à un chemin donné de S à T.

Elle dépend donc non pas de la direction de l'arc mais de celle du chemin choisi.

Dans la figure ci-contre, le premier arc est incohérent :



→ on peut le diminuer (à 8).

L'arc suivant ramène 10 du flux vers la source :

→ il peut diminuer.

Ces deux diminutions augmenteront le flux S --> T.

L'arc suivant (bien orienté) porte 12 unités vers T, il peut être augmenté (par au plus 3 qui est la meilleure valeur d'augmentation sur les 3 arcs).

Cette augmentation est répercutée sur les 3 arcs dans la figure ci-contre.

XIII- Détails et Calcul du maximum de capacité sur un réseau

Problème (graphe valué orienté) :

On veut calculer les chemins de **capacité maximale** depuis un nœud donné (*source*) jusqu'à tous les autres nœuds d'un graphe connexe valué.

- Les poids des arcs (arêtes) représentent les capacités des liens entre les nœuds.

Application-1 : un réseau d'irrigation et l'on veut connaître la **capacité maximum d'irrigation** possible de chaque point (nœud) depuis un nœud **source** donné.

Application-2 : un réseau de routes où l'on désire connaître les chemins de capacité maximale depuis un nœud (une localité) jusqu'aux autres.

- Dans ce problème, la longueur d'un chemin n'a pas d'importance, on veut seulement maximiser les capacités.

- **Par exemple**, dans le cas de régulation de **trafic**, on préfère faire circuler les véhicules sur les voies dégagées (ou de capacité plus grandes) au lieu de créer des bouchons sur des trajets a priori plus courts mais où à cause des bouchons, on ne peut pas circuler !

Un trajet court avec une vitesse quasi nulle est beaucoup moins intéressant qu'un trajet plus long mais avec une circulation plus fluide.

Remarque : habituellement, dans le problème de capacité dans un graphe, chaque arête porte deux valeurs :

une capacité maximum et un flot (ou flux) actuel (inférieur à la capacité maximum).

- Ici, nous considérons un cas simplifié de ce problème et ne traitons que la capacité.
- **La capacité maximum** du chemin $\langle X_1, X_2, \ldots, X_n \rangle$ est :

$$\textit{min}(\textit{capacité}(X_i, X_{i+1})) \text{ , } \textit{tout } i = 1 \ldots n-1$$

Exemple et Principe :

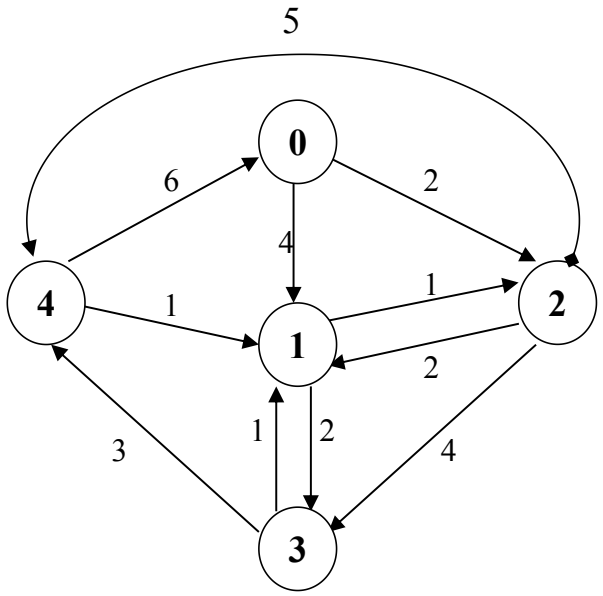
- On part d'un nœud *Start*

- On traite les

X=successeurs(start) et on

calcule les capacités

- Puis on prend les succs. de X ....



Dans ce graphe, en partant du noeud 0, les capacités maximum ainsi que les trajets sont :

Départ = 0
0 → 1capacité = 4
0 → 2capacité = 2
0 → 1 → 3capacité = 2
0 → 2 → 4capacité = 2

Départ = 1
1 → 3 → 4 → 0capacité = 2
1 → 3 → 4 → 0 → 2 capacité = 2
1 → 3capacité = 2
1 → 3 → 4capacité = 2

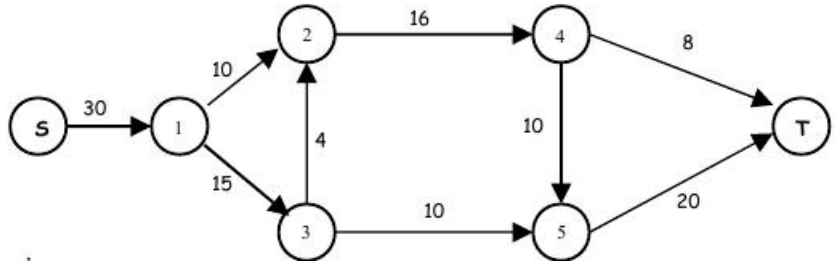
On est dans le cadre de la **Programmation Dynamique** par niveau du graphe (voir figure, et cours 4).

Ce principe est réalisé en conservant pour un nœud le maximum des capacité calculé par les différents chemins partiels.

Rappel : *capa_max(chemin X1_X2 _.. Xn)=min(capa(Xi, Xi+1))* i.e. **le goulet s'impose** !

XIII.1- Principe de la méthode

Soit le graphe suivant où les valeurs portées par les arcs représentent une capacité quelconque :



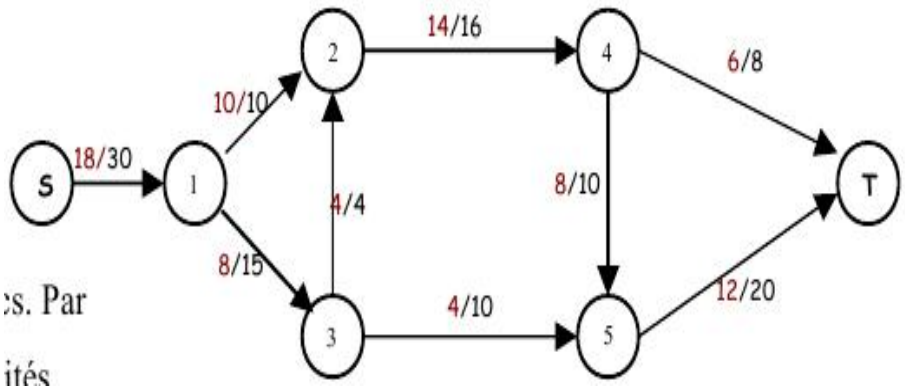
- volume en gaz, eau, pétrole pouvant être véhiculé par ces arcs;
- un nombre de véhicule pouvant circuler sur les voies (un arc = une route),
- etc …

Le **but** est de conduire une quantité maximale depuis le nœud S (source) vers le nœud T (terme).

On note qu'en amont de la source S, la capacité disponible est considérée **infinie** mais au maximum 30 unités pourraient passer de S au nœud 1.

XIII.2- Principe du traitement (Ford-Fulkerson)

Etape k : supposons qu'à l'étape k du traitement, nous ayons le graphe suivant :



- Les valeurs en couleur rouge représentent les quantités *consommées* (déjà utilisées) par les arcs.

  → Par exemple, sur l'arc <S,1>, on a déjà utilisé 18 unités.

- On constate que la capacité (arrivée à T) de cette état est C_k = **6+12=18**.

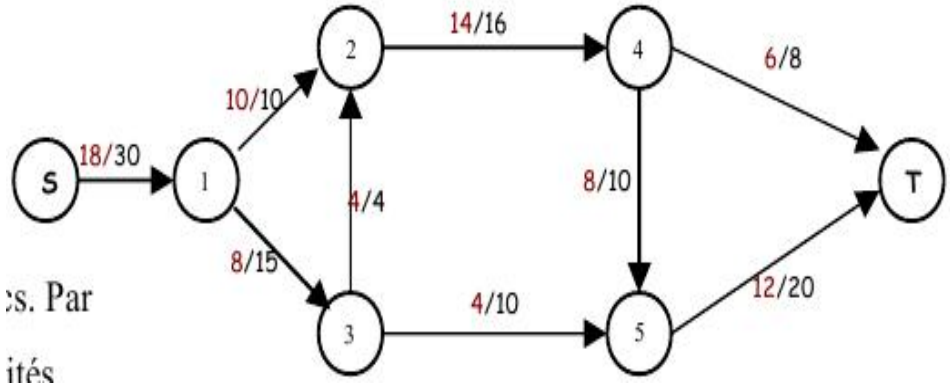
  → on est pour le moment capable d'acheminer 18 unités depuis S vers T.

  → On peut suivre les valeurs arrivant sur T et vérifier comment ces 18 unités traversent le réseau.

On peut attacher à chaque arc la **capacité résiduelle** de l'arc qui est représentée par la différence entre la capacité originelle de l'arc et la quantité déjà utilisée.

→ Par exemple, résiduelle(<S,1>)= 30-18=12.

Etape k+1 : A cette étape, on cherchera un **chemin simple** (sans circuit / boucle) d'une capacité C_{k+1} de S vers T dans le graphe résiduel (où les arcs portent les valeurs résiduelles).



→ Ce chemin apportera une **amélioration** de la capacité déjà calculée.

- On ajoute la capacité apportée par ce chemin (d'amélioration) au résultat C_k ,

→ Et on met à jour les capacités résiduelles et on répète l'étape k+2 comme pour k+1.

N.B. : une simplification dans la recherche d'un chemin d'amélioration (d'*augmentation*) est de supprimer, du graphe résiduel, les arcs dont la valeur résiduelle est devenue nulle. → Ceci permettra de trouver les chemins plus rapidement.

Comment arriver à l'étape k : il suffit de commencer avec k=0.

Conditions d'arrêt : lorsque plus aucun chemin améliorant n'existe.

Calcul des chemins améliorants :

- Il existe une heuristique qui préfère le parcours en **largeur**.

  → Dans ce cas, l'algorithme est appelé **Edmond-Karp**.

- On peut démontrer que le parcours en largeur trouvera le plus "rapidement" (plus tôt dans le graphe) un chemin simple de capacité locale maximale.

- On constate que ce problème peut se décliner sous la forme d'un problème de **chemin le plus long** entre S et T.

Question de Complexité :

- Le flux max. sera atteint lorsque plus aucun chemin améliorant ne peut être trouvé.
- Mais ce problème est dans le cas général indécidable (en particulier avec des capacités dynamiques et changeantes).
- Par contre, si on peut garantir la **terminaison**, la solution trouvée est **correcte**.
- Le calcul d'un chemin dans un graphe est $O(|E|.|V|)$.
- L'utilisation d'un TAS permet d'atteindre $O(|E|. \log(V))$ si le graphe n'est pas dense. Dans le cas contraire (graphe dense), le TAS n'apporte rien (pour $|E|=V^2$).

- Si les valeurs sur les arcs sont des entiers, la complexité de Ford-Fulkerson est $O(|E|. m)$ avec m=maximum de capacité.

  → Au pire, chaque chemin *d'augmentation* apporte au moins une unité.

- La complexité de Edmond-Karp : $O(|E|^2.|V|)$.

XIII.3- Algorithme Edmonds-Karp

Action principale : augmenter le Flux tant qu'il y a un chemin améliorant.

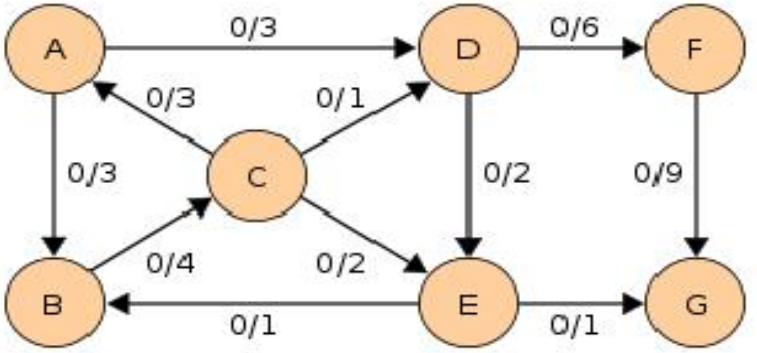
```
Flux_Max(Graphe G, Source S, terme T) renvoie entier      % le flux maximum
  Flux_ameliorant=0;  Flux_max=0;
  existe_chemin_ameliorant =faux;
  répéter :
    Trajet=vide;
    existe_chemin_ameliorant=
          un_chemin_residuel_en_largeur(G, S, T, Trajet, Flux_ameliorant);
    Si (existe_chemin_ameliorant == faux)  Alors quitter l'itération        // break;
    Flux_max = Flux_max  + Flux_ameliorant;
    Mise à Jour des Residuels(G, S, Trajet, Flux_ameliorant);
  jusqu'à  existe_chemin_ameliorant=faux
  renvoyer  Flux_max;
```

Etape k : recherche d'un chemin en largeur d'une capacité Ck :

```
Fonction un_chemin_residuel_en_largeur(Graph G,  Source S, terme T, Trajet,  Flux_ameliorant) renvoie bool
        Coming_From : vecteur[taille de G] d'entiers          % permet de calculer le trajet (chemin)
        Marquage : vecteur[taille de G] de bool init faux
        pour tout Noeud dans  G  Coming_From [i]=i;
        File F=vide
        enfiler(F, S);
        Tant que F <> vide
           Noeud = premier(F);            % opération en 2 temps : consulter sommet + défiler.
            Si nœud = T                   % on calcule min des capa sur ce trajet
                  <Flux_ameliorant , Trajet> =  extraire à l'aide de Coming_From  Trajet Flux_ameliorant via G
           renvoyer Vrai;
      FinSi
           Pour tout X successeur de Nœud dans G
           Si X non marqué et Capacité résiduelle de <Noeud ,X>
                 Alors marquer X;
                       Coming_from[X]=Noeud;
                       enfiler(X);
                 FinSi
            Fin Pour
      Fin  Tan que
      renvoyer Faux;
Fin  un_chemin_residuel_en_largeur
```

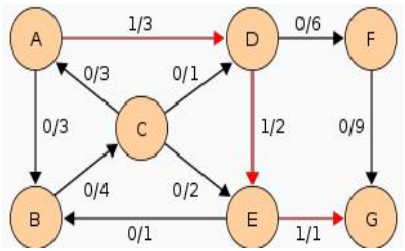
Exemple : soit le graphe pour lequel on veut calculer le flux maximum.

Les étapes de l'algorithme ci-dessus sont décrites ci-dessous.

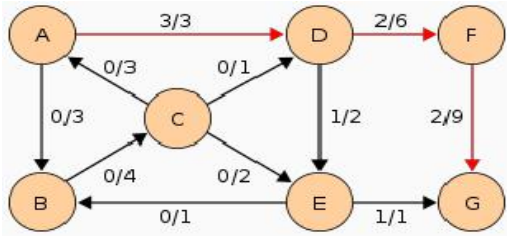


Flux Maximum calculé = 5.

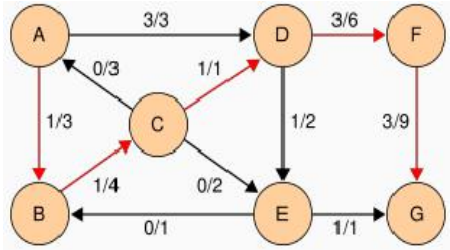
Les trajets sont donnés en couleur rouge.



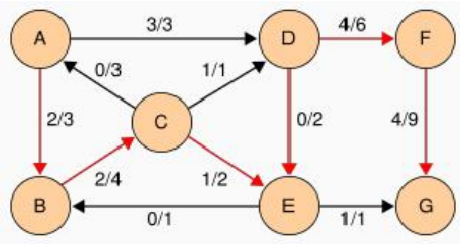
(k=1) C_1=1



(k=2) C_2=2



(k=3) C_3=1

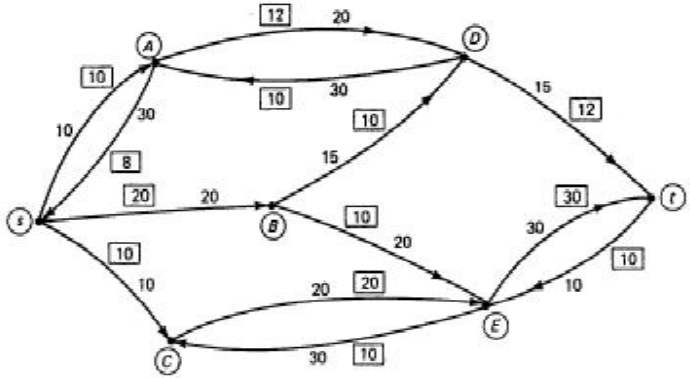


(k=4)

C_4= 1

Exercice : vérifier que la capacité maximum du graphe suivant = 40.

Remarque : on peut prouver que la capacité calculée pour un nœud en choisissant le maximum des capacités partielles produit la bonne solution.



Soit le chemin allant de Start à Y :

Start X1 X2 ... Y de capacité C

Soit un autre chemin de Start à Y :

Start X'1 X'2 ... Y de capacité C'

Pour que C'>C, il faut : capacité (Start, X'1) ≤ capacité (Start, X1)

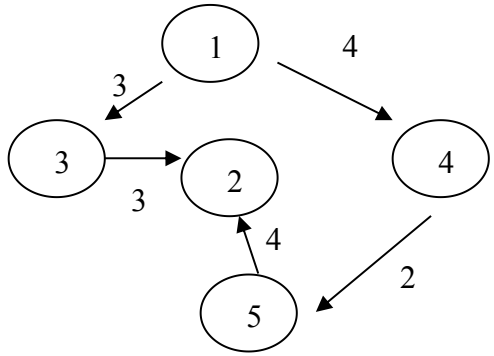
(pour que X'1 soit traité après X1) et min(capacité(X'i, X'j)) > C.

Or, on peut prouver que cela n'arrive pas si l'on choisit toujours le nœud de capacité maximum. ../..

XIII.4- Exemple 1

Y a-t-il un risque de calculer pour le nœud 2 la capacité 2 (1-4-5-2) et de s'en servir pour calculer d'autres capacités avant de trouver qu'en passant par 3,

on pourrait calculer la capacité 3 pour le nœud 2 ?



Non, car (1-3) est de capacité 3 > capa(4,5) et 3 sera choisi avant 5.

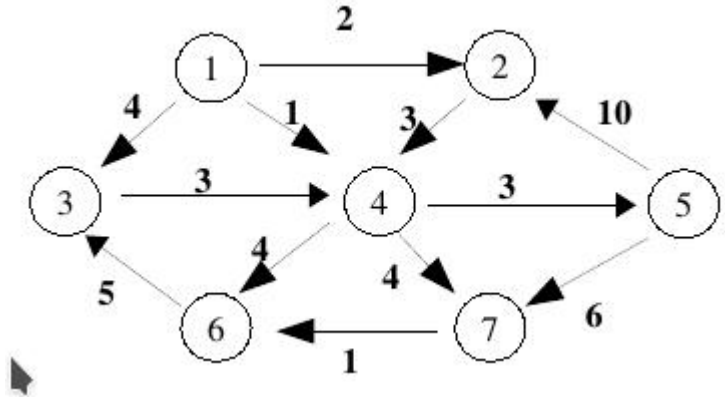
Rappel : la méthode de **programmation dynamique**.

XIII.5- Exemple 2

Cet exemple montre également qu'il faut prendre les candidats max ($Capa_i$)

Départ=1

Si l'on traite dans l'ordre 1,2,4,3 ... on aura des mauvais résultats (puisqu'il n'y a pas d'ordre).



Solution : quand on choisit le prochain nœud, il faut que sa capa soit maximale (utiliser un Heap).

Preuve : soit un chemin de capacité C : x1,x2...,xn.

Peut-il y avoir un autre chemin de meilleure capacité C' x1,x'2, ..., x'm

non car x1 -> x2 > x1 -> x'2 C>C'

Et si x1 -> x2=x1 alors x'2, C=C' ...

Mise en garde importante : *comme on peut le constater dans ces exemples, nous abordons le problème du flux (flot) et de la capacité dans une approche non optimisée pour en donner une idée plus simple.*

Les algorithmes suivants sont indicatifs et ne donnent pas les mêmes résultats que celui de Ford – Fulkerson.

Voir la bibliographie (au chapitre 1) pour plus de détails.

XIV- Capacité maximum entre deux nœuds (B & B)

- Dans la section précédente, on a calculé la capacité maximum **entre un nœuds et tous les nœuds** d'un graphe. Même si nous étions intéressés par le flux maximum entre la sources S et l'arrivée T, les flux calculés sont maximaux sur l'ensemble des nœuds.

- Dans certains cas, l'on peut être intéressé par la capacité maximum entre deux nœuds N et M, autre que S et T.

- Hypothèse : utiliser la méthode précédente pour calculer les capacités entre N et tous les autres nœuds puis utiliser celle entre N et M.

  C'est le principe de *Dijkstra* (pour les chemins)

- Mais si le graphe contient un nombre important de nœuds, ce calcul peut être très coûteux. Il y aura beaucoup de calculs inutiles.
- La méthode : on calcule la capacité maximum entre deux nœuds (Départ et Arrivée).
- La méthode de parcours de graphe utilisée est dite "**Branch & Bound**" (développer – évaluer/borner).

B & B : on développe un arbre de recherche et une **fonction d'évaluation** permet d'attribuer une valeur (**coût**) à chaque nœud développé.

Le but de la recherche B&B est de minimiser (resp. maximiser, selon le problème) la valeur de cette fonction.

Remarque : dans un parcours pour trouver le chemin le plus court, on tente de minimiser le coût d'un chemin (la longueur du trajet) et d'arriver à une destination en ayant minimiser la longueur du chemin.

Par contre, dans le cas du problème présent, on veut maximiser la valeur de cette fonction : la capacité d'un chemin entre deux nœuds.

XIV.1- Principe de la méthode B&B

Rappel : la capacité d'un chemin <X1, X2, ..., Xn> est le **minimum** des capacités de chaque arc <Xi, Xi+1> et on veut trouver un chemin entre deux nœuds qui **maximise** cette capacité.

Dans cette méthode, les successeurs possibles du *Départ* sont développés et évalués selon la fonction d'évaluation de la capacité.

- Parmi ces successeurs, on choisira celui qui maximise la capacité du chemin partiel depuis le *Départ* jusqu'à ce nœud.

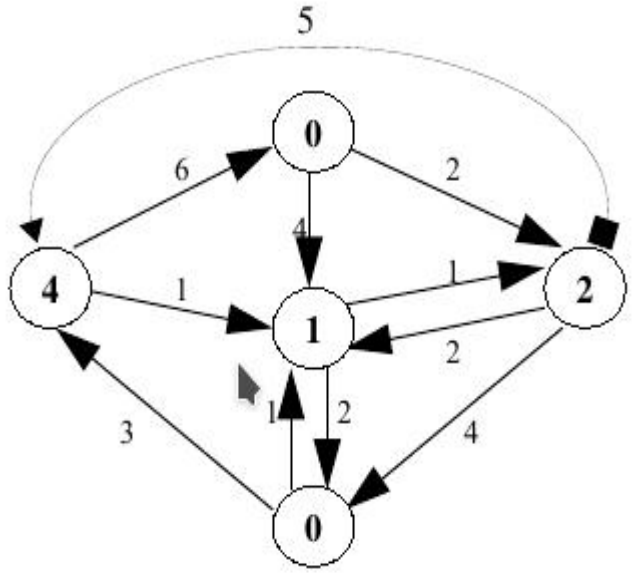
Pour choisir le nœud prochain à développer, on sélectionne le nœud le plus prometteur :

→ celui dont la capacité du chemin partiel est maximum.

XIV.2- Exemple 1

N.B. : Ici, **le maximum local ne donne un max global** (Hill Climbing)

Voir aussi l'exemple du graphe suivant.



Problème : Départ=3 et Arrivée=1.

Trouver un chemin entre ces 2 nœuds de capacité max.

Les successeurs de 3 sont : {1 , 4},

Les chemins partiels <3, 1> et <3, 4>.

capacité(<3, 1>) = 1, capacité(<3, 4>) = 3.

On choisit de continuer le développement du chemin <3, 4>.

On notera un chemin partiel <X , Y> de capacité C par : **<X , Y>**$_C$.

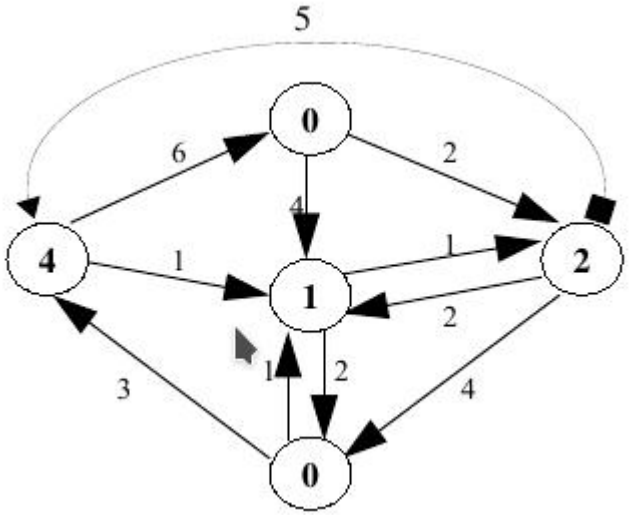
Les chemins partiels sont stockés dans une liste ordonnée selon la valeur de la capacité.

La tête de cette liste contient le chemin partiel de capacité maximum.

Dans ce chemin partiel, si le dernier nœud = Arrivée, ce chemin partiel donne une solution au problème (pas forcément le meilleur car choix du meilleur local).

Les contenus de la liste des chemins partiels

pour Départ=3, Arrivée=1 sont :



- [<3>]
- [<3 , 4>$_3$, <3 , 1>$_1$] Liste ordonnée selon les capa
- [<3 , 4, 0>$_3$, <3 , 4, 1>$_1$,<3 , 1>$_1$] Liste ordonnée
- [**<3 , 4, 0, 1>$_3$** , <3 , 4, 0, 2>$_2$, <3 , 4, 1>$_1$,<3 , 1>$_1$]

Liste ordonnée

La tête de la liste donne ici le chemin 3 4 0 1 de capacité 3.

XIV.3- Exemple 2

Départ=3, Arrivée=2

[<3>]

[<3 , 4>$_3$, <3 , 1>$_1$] Liste ordonnée

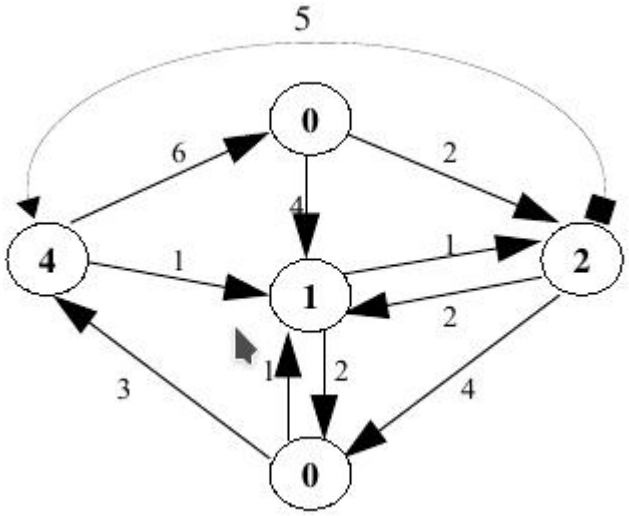
[<3 , 4, 0>$_3$, <3 , 4, 1>$_1$,<3 , 1>$_1$] Liste ordonnée

[<3 , 4, 0, 1>$_3$, <3 , 4, 0, 2>$_2$, <3 , 4, 1>$_1$,<3 , 1>$_1$]

Liste ordonnée

[<**3** , 4, 0, **2**>$_2$, <3 , 4, 0, 1, 2>$_1$, <3 , 4, 1>$_1$,<3 , 1>$_1$]

Liste ordonnée



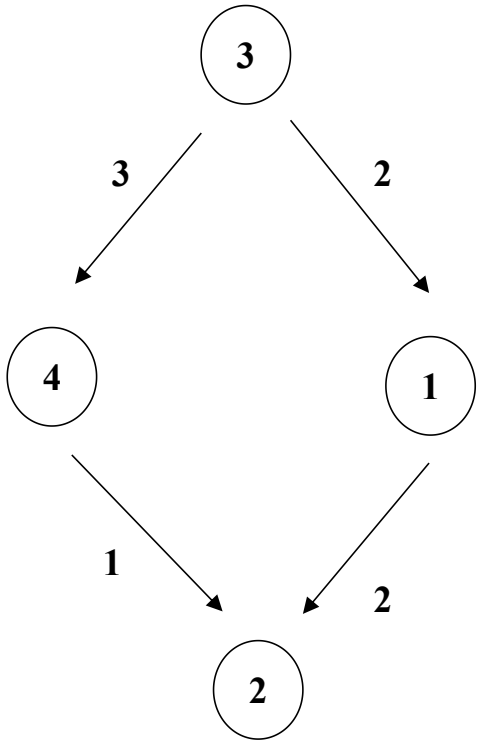
Remarque : le chemin partiel <3 , 4, 0, 1>$_3$ aura deux successeurs :

<3,4,0,1,2>$_1$ et <3, 4,0,1,3>$_1$.

Mais le chemin partiel <3, 4, 0, 1, 3> contient un circuit : 3 ... 3.

Ce développement est éliminé de la liste des chemins partiels.

XIV.4- Exemple 3



La liste traitée :

[<3>]

[$<3,4>_3$, $<3, 1>_2$]

[$<3,1>_2$, $<3, 4, 2>_1$]

Remarque : même si l'on arrive ici, cet élément n'est pas en tête

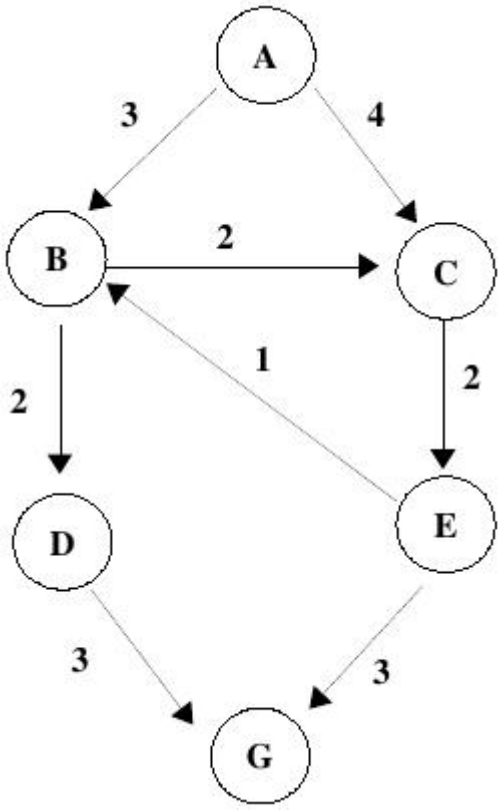
--> l'algorithme continue.

N.B. : Dans une variante de cet algorithme, on peut examiner la présence de la solution (arrivée présente) avant de développer le chemin partiel en tête de la liste.

XIV.5- Exemple 4

N.B. : Voir dans le cahier Fit les étapes sur le graphe.

(feuille manuscrite).



Trace :

A –(2)--> B –(2)--> D –(2) --> G (2)

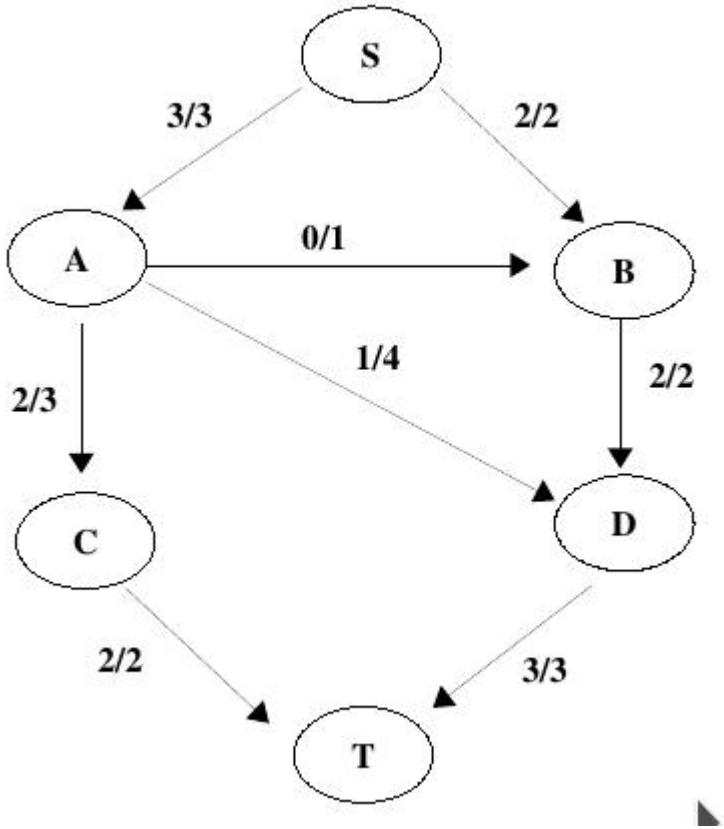
La capacité du goulet l'emporte.

A –(2)--> C –(2)--> E –(2) --> G (2)

…

XIV.6- Exemple 5

Avec arc **inversé** (n/m : capacité utilisée sur la capacité de l'arc).



Trace :

S –(3)--> A –(3)--> D –(3) --> T (3)

Remarque sur les structures de données :

Pour représenter le graphe G, prévoir une table des valeurs

Prévoir une table des prédécesseurs avec des valeurs 0

Trouver un chemin de capacité ≠ 0 et mettre à jour les succ/préd jusqu'à plus de successeur depuis S.

XIV.7- Remarques sur la méthode B & B

La méthode précédente privilégie un maximum local (Hill Climbing).

XIV.7.1- Méthode B & B globale

Dans une variante de B&B qui aboutit à un maximum global, on procède de la manière suivante :

1. On calcule (parcours en profondeur) un premier chemin avec un coût de référence.
2. On rentre dans une itération pour calculer tous les chemins mais à chaque mise à jour d'un coût partiel (par exemple, lors de l'ajout d'un arc au chemin partiel en cours de construction), on compare la coût partiel avec le coût de référence. Le chemin partiel est abandonné si le coût de référence est meilleur.

3. Si un nouveau chemin aboutit avec un coût inférieur à la référence, ce dernier de vient la référence.

4. Les itérations continuent jusqu'à l'épuisement de tous les chemins possibles.

L'inconvénient de B&B global est de développer tout le graphe de recherche. Cet inconvénient est atténué par le fait que certaines branches sont abandonnées dès que leur coût est supérieur au coût de référence. Ce qui nécessite des tests réguliers.
Il existe des environnements de programmation (avec gestion de contraintes) qui facilitent cette gestion.

(Voir Annexes pour le code).

XIV.8- Addendum : détails de l'implantation

Le graphe orienté valué est ici représentée par une matrice d'adjacences.

Cette matrice carrée contient les capacités des arcs. On note par -1 l'absence d'arc entre deux nœuds.

Pour le stockage des chemins partiels, on utilise une liste des chemins partiels développés.

Chaque boîte de cette liste contient :

- le nœud de départ
- le nœud d'arrivée
- le nœud courant (du chemin partiel depuis Départ)
- le trajet entre Départ et le nœud courant
- la capacité du chemin partiel

../..

- La capacité maximum d'un chemin est le minimum des capacités de chaque arc de ce chemin.
- Le maximum local ne donne **pas forcément** le maximum final.
- On ne choisit par le meilleur arc (meilleure capacité) entre deux nœuds mais le meilleur chemin partiel.
- Les circuits sont évités et un chemin partiel avec circuit est éliminé de la liste.
- Le trajet est un vecteur où pour chaque nœud (0..N), on note dans T[i] le successeur de i (le tableau *going_to*)
- L'algorithme s'arrête quand le nœud courant du chemin partiel en tête de la liste des états développés est égal à l'Arrivée. Ce nœud contient son propre trajet.

XV- Annexes

NDLR : Voir éventuellement (aussi) le fichier Annexes de codes de 2008-09

XVI- Autres représentations de graphes

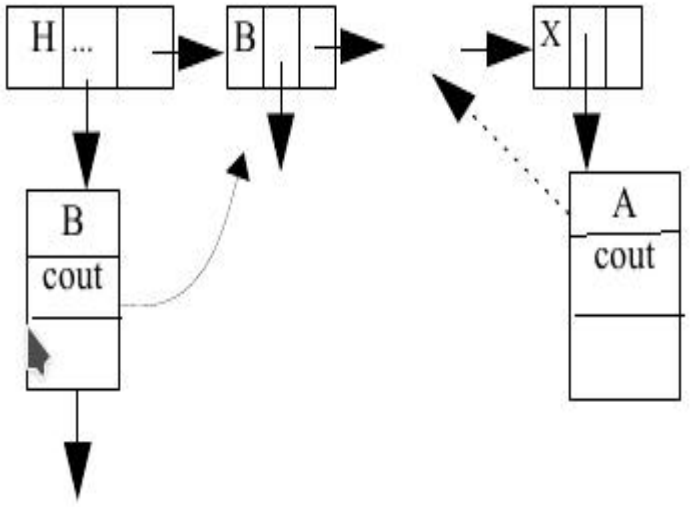
XVI.1- Représentation dynamique : structures de données(à la 'C')

Dans la représentation par listes d'adjacents accrochées à un tableau, si le tableau à son tour est une liste chaînée, on peut utiliser la structure suivante pour représenter dynamiquement le graphe.

C'est une des structures fréquemment employée.

Représentation avec un tableau de pointeurs :

Les nœuds du graphe sont présentés dans un tableau.



Chaque ligne du tableau contient le nom du nœud, son nombre d'adjacents (éventuellement) et un pointeur sur la liste des adjacents de ce nœud.

Chaque nœud d'adjacent contient l'indice de l'adjacent dans le tableau, un coût et un champ suivant.

(Voir la solution suivante avec des pointeurs).

Déclarations (C/C++):

```
typedef struct nœud_secondaire          // un nœud d'adjacent
   {struct nœud_prinipal * info;
    int cout;                           // le coût de la transition
    struct nœud_secondaire * svt;
   } *Liste_adj;
typedef struct nœud_principal           // un nœud principal
   {int info;                                 // ou char ...
    int nb_adj;                         // éventuellement
    struct nœud_ principal * svt;
   } *Graphe;
```

L'implantation (voir annexes) :

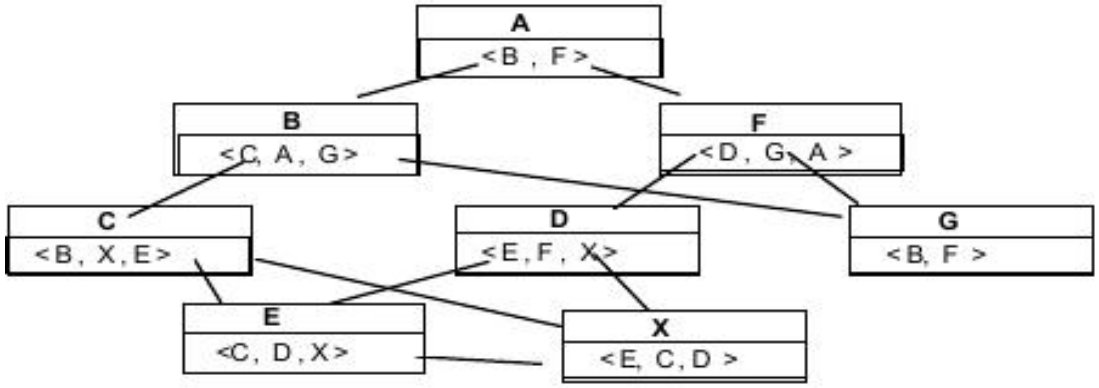
- Parcours en profondeur avec recherche de chemin :
  - avec trajet + coût + gestion des circuits.
    - Le coût est 1 entier;
    - le trajet est représenté par un tableau d'entiers (type tableau *going-to*)
- La gestion du marquage (si non faite à l'aide du trajet) par un tableau d'entiers.

XVI.2- Représentation Dynamique par listes doublement chainées

- Graphe non orienté non valué :

```
Type nœud=enregistrement
        info :  information contenue dans le nœud
        adjacents :  Liste de nœud;
   Fin nœud;
Type graphe=Liste  nœud;
```

Remarques :

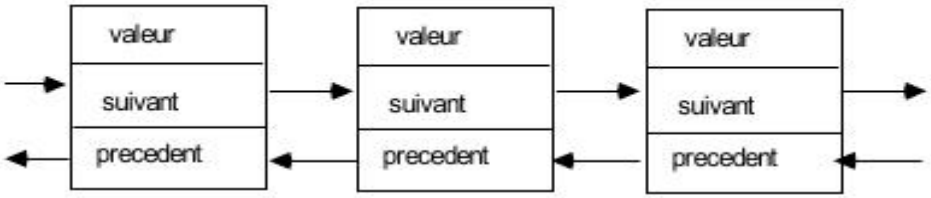


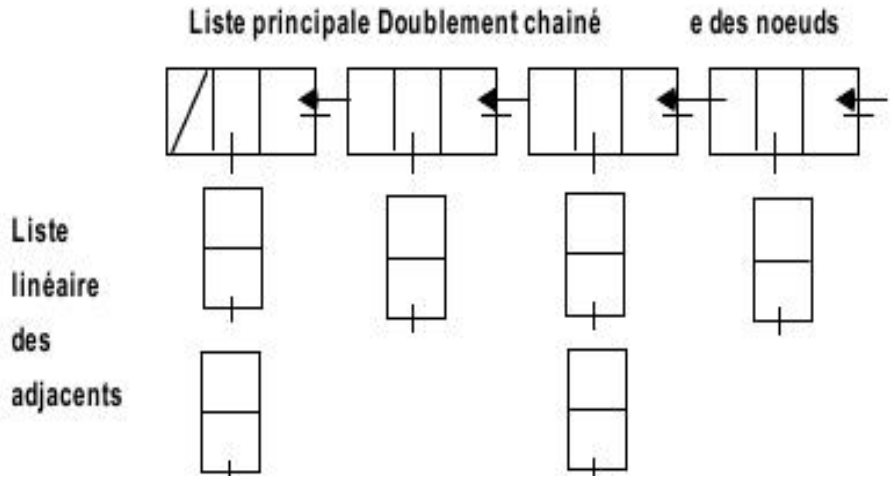
- La liste principale ainsi que la liste des *adjacents* ne représentent aucun ordre.

- La liste principale peut être doublement chaînée. Ce qui permet de traiter des parcours dans les deux sens (cf. problèmes tels que PERT).

En STL, cette réalisation pourra se faire à l'aide des itérateurs.

Les listes principale et d'adjacents :



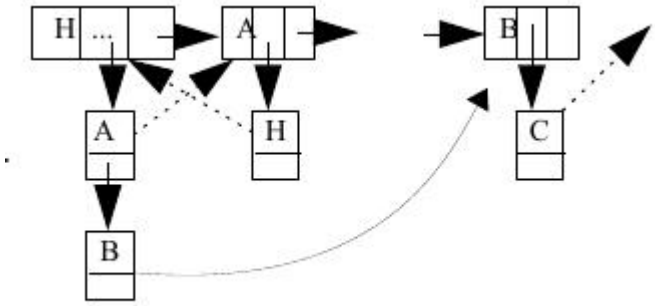


- On pourra envisager d'autres formes (liste des adjacents doublement chaînée, liste principale circulaire, ...)

Représentation compacte dans des listes secondaires :

Dans la liste des adjacents, on peut éviter de répéter les informations sur les nœuds adjacents en utilisant une référence sur le nœud en question.

Dans ce schéma, pour le nœud H, l'adjacent B est représenté par une références (pointeur, itérateur, ...) sur ce nœud dans la liste principale (plutôt que de répéter le nœud B).

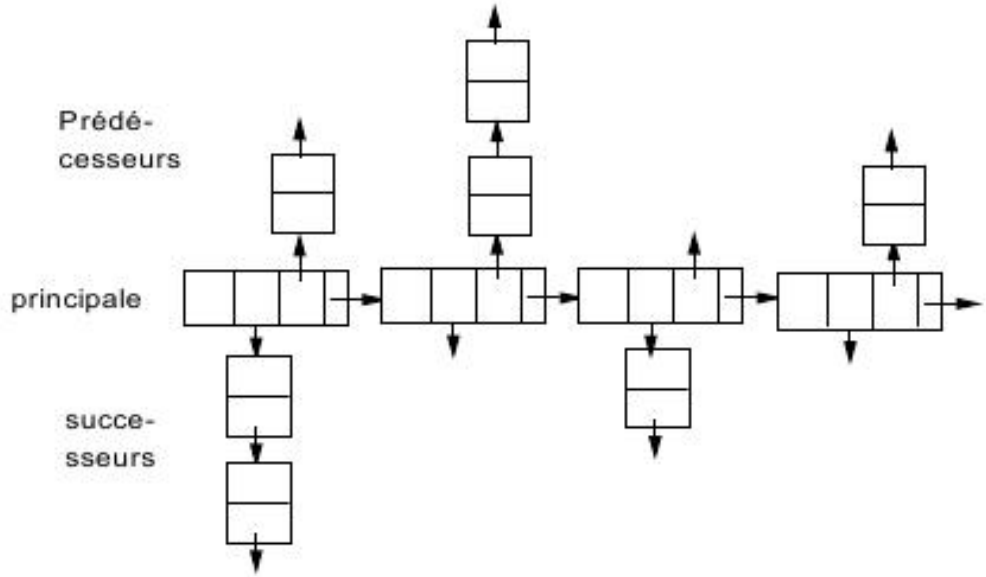


Pour la partie déclaration, voir la définition suivante.

Cas de graphe orienté :

```
Type nœud=enregistrement
        info : ......
        pred, succ : ref nœud;
   Fin nœud;
Type graphe=Liste de  nœud;        // équivalent à "ref Nœud"
```

- On met en place la possibilité pour chaque nœud d'accéder à ses successeurs et à ses prédécesseurs.
- Le graphe reste orienté et la remarque ci-dessus reste valable.



Cas de graphe valué :

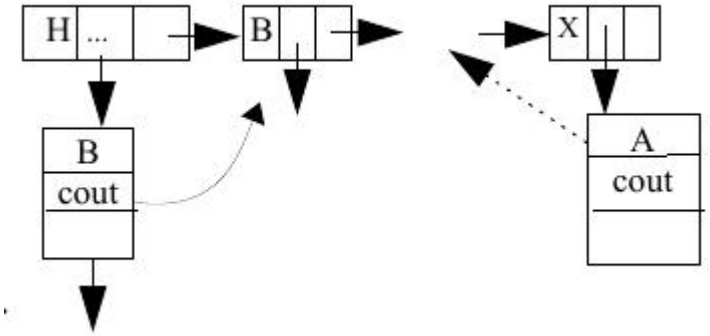
- Chaque élément de la liste des adjacents (ou succ. ou prédécesseurs) peut contenir une valeur.
- Par exemple, pour un graphe valué orienté :

```
Type nœud=enregistrement
        info : ......
        adjacents : liste de  (ref nœud x cout);      // cout : valeur de l'arc/arête
Fin nœud;
```

- La liste des adjacents est de la forme :

Ici, pour l'adjacent B du nœud H, le champ "coût" peut contenir le coût d'aller de H à B.

Ce coût peut être une distance / vitesse / durée, ...



Remarque : La liste principale et celle des adjacents ne représentent en général aucun ordre.

Le seul ordre (orientation) est exprimé par le fait qu'un nœud possède (ou non) une liste de succ.

- **Autres structure de données :**

  Par 3 tableaux (plus sophistiquée); List/vector de STL avec des itérateurs, ...

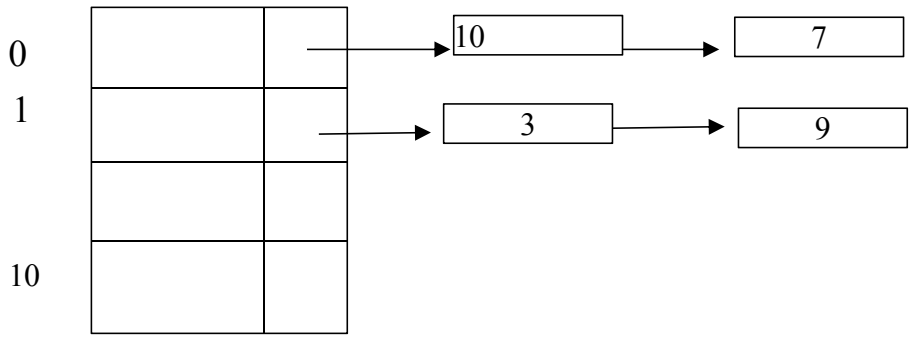
XVI.3- Exercices du calcul du chemin dans un graphe

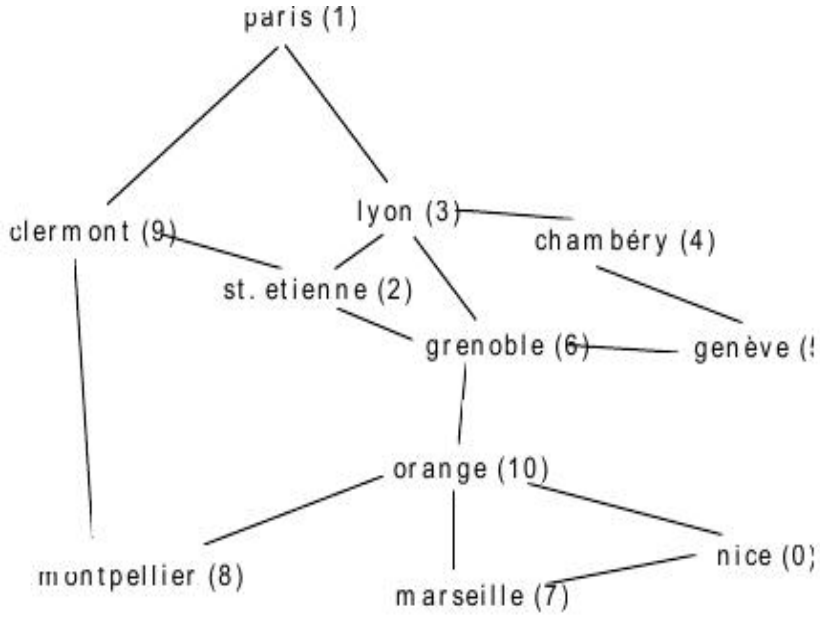
XVI.3.1- En profondeur : cas de graphe non valué

Graphe d'un **réseau routier**

Une <u>représentation</u> possible du graphe :

un tableau de *Max_ville* référençant les adjacents de chaque nœud :

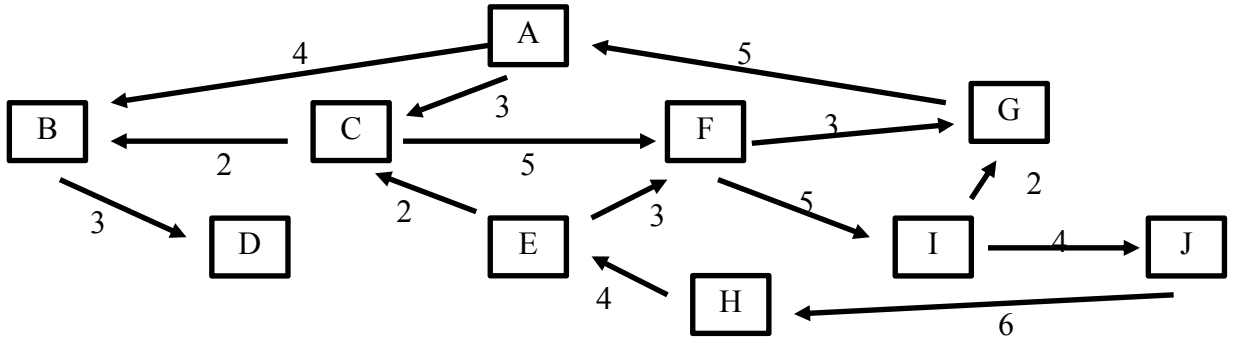




- On utilise une **file d'attente** pour le parcours en largeur d'abord.
- Le **chemin** est un tableau de 0..Max_ville d'entiers.
- La table de **marquage** est un tableau de 0..Max_ville de booléens.

XVI.3.2- En profondeur avec coût (graphe valué)

Soit le graphe orienté valué :



Les valeurs sur les arcs représentent les coûts des transitions : aller de A à B coûte 4 unités.

A- Proposer une structure de données pour implanter ce graphe en mémoire.

B- Écrire l'algorithme de recherche d'un élément (**en profondeur**) avec le calcul du coût de cette recherche. Cet algorithme doit gérer les circuits dans le graphe.

Indications : Parcours en profondeur avec recherche de chemin :

- Avec trajet + coût + gestion des circuits.

- Le coût est 1 entier;

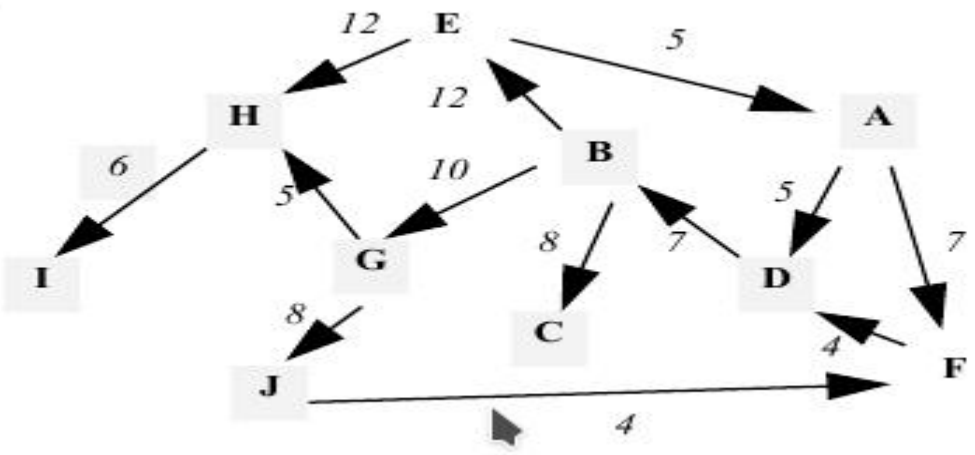
- Le trajet est représenté par un tableau d'entiers (type tableau *going-to*)

- La gestion du marquage (si non faite à l'aide du trajet) par un tableau d'entiers.

XVI.3.3- Exercice : parcours en largeur (graphe valué)

- Soit le graphe orienté valué suivant

Les nombres sur les arcs représentent les coûts des transitions : aller de A à B coûte 5 unités.



A- Proposer une structure de données pour implanter ce graphe en mémoire, puis

B- Un algorithme de recherche d'un élément dans ce graphe avec le calcul du coût.

La méthode de recherche employée doit être une stratégie de parcours en largeur (par niveau).

Cette recherche ne doit donc pas examiner les nœuds dans un ordre aléatoire (par exemple, l'ordre de la construction du graphe).

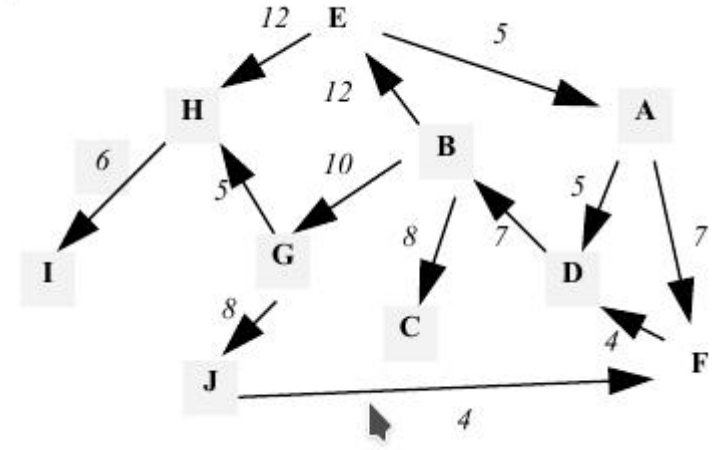
Par exemple, pour aller de A à G dans le graphe ci-dessus :la méthode de parcours en largeur examine tous les successeurs d'un nœud avant de descendre au niveau suivant../..

Dans le graphe ci-dessus, on examine dans l'ordre les nœuds A, B, H , C, I, J, G.

A titre d'indication, le parcours en profondeur donnera A, B , C, H, I , G.

XVI.3.4- En largeur (Best First)

Reprendre l'exercice précédent mais lors du choix, sélectionner toujours le successeur de meilleur coût (optimum local).



Par exemple, parmi les successeurs de **E**, on choisira ***A*** avant **H** (s'il s'agit de minimiser un coût).

H sera exploré en cas d'échec du choix précédent.

=> Conservation des trajets partiels.

=> Lien avec B & B.

=> Choix irrévocable.

Table des matières