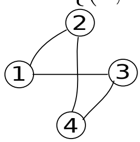
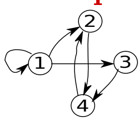
Ressources : Cormen-Leiserson-Rivest-Stein : Algorithmique (Partie Algorithmes pour les graphes)
www.sagemath.org

Graphe : Un graphe G est un couple (S, A) où S est un ensemble fini (les sommets) et $A \subset S \times S$ (les arêtes)
Dans un graphe non orienté, les couples (a, b) et (b, a) désignent la même arête.
Dans un graphe orienté les arcs (a, b) et (b, a) sont distincts. On dit que l'arc (a, b) part du sommet a et arrive au sommet b.

Exemple : $G = (S, A)$ non orienté avec $S = \{1, 2, 3, 4\}$ et $A = \{(1, 2), (1, 3), (2, 4), (3, 4)\}$



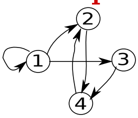
Exemple : $G' = (S', A')$ un graphe orienté avec $S' = \{(1, 2), (1, 3), (2, 4), ($



Adjacence : Soit (u, v) une arête de G. On dit que v est adjacent à u
Rq : Si G est non orienté alors l'adjacence est symétrique.

Degré : Le degré sortant d'un sommet u de G est le nombre de sommets de G adjacents à u.
Le degré entrant d'un sommet v de G est le nombre de sommets de G auxquels v est adjacent.
Pour un graphe orienté, on appelle degré la somme du degré sortant et du degré entrant.
Pour un graphe non orienté, dégré sortant = degré entrant et on note simplement degré. C'est le nombre d'arêtes **incidentes** au sommet.

Exemple :



$D_s(2) = 3 D_e(2) = 2 D(2) = 5$

Lemme des poignées de main : Soit $G = (S, A)$ un graphe. Alors $\sum_{s \in S} D(s) = 2|A|$

Application : Si il y a 7 équipes il est impossible de faire que chacune joue contre 5 autres équipes (car alors $2|A| = 35$).

Propriété : Dans un graphe non orienté, il y a toujours au moins deux sommets avec le même degré.

Preuve : Soit n le nombre de sommets de G. Les degrés possibles sont $\{0, \ldots, n-1\}$. Il ne peut pas y avoir à la fois un sommet de degré 0 et un sommet de degré $n-1$, donc il y a en réalité $n-1$ degrés possibles, pour n sommets, donc il y a obligatoirement une répétition.

Conséquence : Dans une soirée il y a toujours au moins 2 personnes qui ont le même nombre d'amis présents.

Egalité et isomorphie : Deux graphes $G = (S, A)$ et $G' = (S', A')$ sont égaux si $S = S'$ et $A = A'$.
Deux graphes $G = (S, A)$ et $G' = (S', A')$ sont isomorphes si il existe une bijection f de S dans S' (appelé ré-étiquetage de S en S') telle que $(a, b) \in A \Leftrightarrow (f(a), f(b)) \in A'$.

Exemple : $S = \{1, 2, 3, 4\} A = \{(1, 2), (1, 3), (2, 4), (3, 4)\}$ et $S' = \{a, b, c, d\} A' = \{(b, c), (c, d), (b, a), (d, a)\}$ sont isomorphes par

$$f : a \to 1$$

$$f : b \to 2$$

$$f : c \to 4$$

$$f : d \to 3$$

Chemin : Un chemin dans un graphe $G = (S, A)$ est une séquence de sommets $C = [c_1, \ldots, c_n]$ telle que $(c_i, c_{i+1}) \in A \forall i \in \{1..n-1\}$

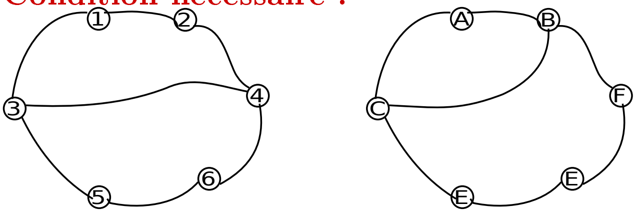
Longueur : La longueur de C est $n - 1$

Chemin simple : $i \neq j \Rightarrow c_i \neq c_j$

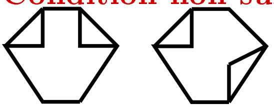
Cycle : Un cycle est un chemin "simple" mais avec $c_1 = c_n$, sauf les chemins du type $[u, v, u]$ des graphes non orientés.

Critère d'isomorphie : Deux graphes isomorphes ont le même nombre de cycles de chaque longueur. Ce critère est nécessaire mais non suffisant. (fig6)

Condition nécessaire :



Condition non suffisante :



Composante connexe : Soit $G = (S, A)$ un graphe non orienté.

Une commposante connexe de G est un sous ensemble maximal de sommets S' tel que pour toute paire de sommets (u, v) de S', il existe un chemin de u à v.

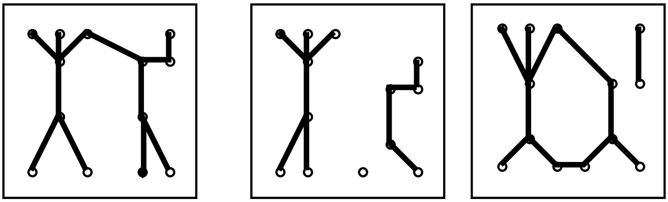
Connexité : Un graphe est dit connexe s'il ne contient qu'une seule composante connexe.

Composante fortement connexe : Soit $G = (S, A)$ un graphe orienté. Une commposante fortement connexe de G est un sous ensemble maximal de sommets S' tel que pour toute paire de sommets (u, v) de S', il existe un chemin de u à v.

Graphe fortement connexe : Ses sommets forment une composante fortement connexe.

Arbre : Une arbre est un graphe non orienté connexe et sans cycle.

Forêt : Un graphe est une forêt si chacune de ses composantes connexes est un arbre.



Arbre	oui	non	non
Forêt	oui	oui	non

Arbre couvrant : A partir d'un graphe connexe, on peut se réduire à un arbre en "coupant" des arêtes. C'est l'arbre couvrant.

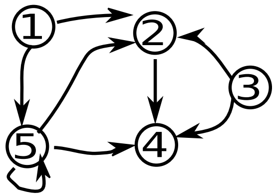
Simplification : On suppose que les sommets sont $\{1, \ldots, n\}$

Matrice d'adjacence :

$M = (m_{ij})_{1 \leq i \leq n, 1 \leq j \leq n}$ où $m_{ij} = \begin{cases} 1\text{si}(i, j) \in A \\ 0\text{sinon} \end{cases}$

Exemple :

$$\begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$



On utilisera donc un tableau bidimensionnel.

Graphes par listes d'adjacences : A chaque sommet est associée une liste qui contient les sommets qui lui sont adjacents.

On utilisera donc un tableau de listes.

Exemple :

$1 : 5, 2$

$2 : 4$

$3 : 2, 4$

$4 :$

$5 : 2, 4, 5$

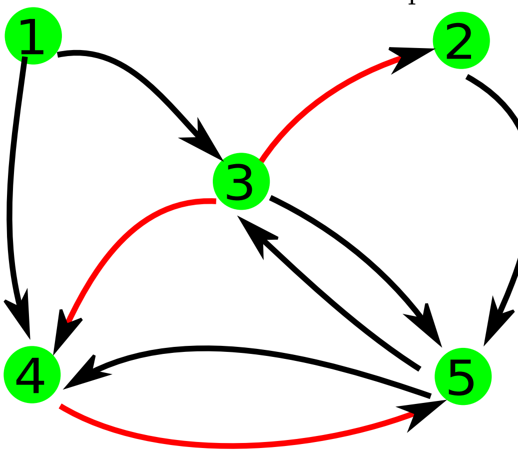
Comparaison des deux représentations :

	Matrice	Listes
Espace	$O(n^2)$	$O(n^2)$ au pire cas, $O(n)$ au meilleur cas
Initialisation	Pour i de 1 à n faire (Pour j de 1 à n faire) M[i,j]=0, donc $O(n^2)$	Pour i de 1 à n faire (T[i] = liste vide), donc $O(n)$
Test d'adjacence	$O(1)$	$O(n)$ au pire cas, $O(\text{nbVois})$ au cas moyen, $O(1)$ au meilleur cas
Afficher les voisins	$O(n)$	$O(n)$ au pire cas, $O(\text{nbVois})$ au cas moyen, $O(1)$ au meilleur cas
Ajouter l'arête (u, v)	T[u].insererEnTete(v) $O(1)$	M[u,v] = 1 $O(1)$

Conclusion :

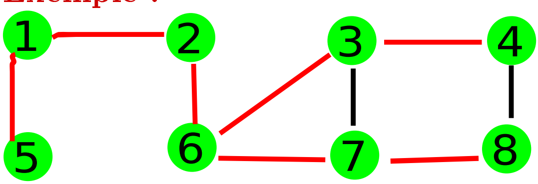
- Si on a un grand graphe avec peu d'arêtes par rapport au nombre de sommets (ou alors vraiment beaucoup, dans ce cas on peut inverser), la représentation en listes est préférable.
- Si on a un petit graphe, une matrice est plus simple à manipuler et pas plus couteuse
- Pour les autres cas, le choix dépendra du contexte.

Principe : A partir d'un sommet, on calcule un arbre couvrant sur les sommets accessibles à partir de ce sommet.



Parcours en largeur : On regarde les voisins du sommet, puis leurs voisins et ainsi de suite. Cela produit des chemins de longueur minimales du sommet de départ à tous les autres.

Exemple :



vu	V	V	V	V	V	V	V	V
pere	2	-	6	3	1	2	6	7
dist	1	0	2	3	2	1	2	3

F : $\bar{2}$ $\bar{1}$ $\bar{6}$ $\bar{5}$ $\bar{3}$ $\bar{7}$ $\bar{4}$ $\bar{8}$

u = 2 v = 1 v = 6

u = 1 v = 2 (déjà vu) v = 5

u = 6 v = 2 (déjà vu) v = 3 v = 7

u = 5 v = 1 (déjà vu)

u = 3 v = 4 v = 6 (déjà vu) v = 7 (déjà vu) u = 7 v = 3 (déjà vu) v = 6 (déjà vu) v = 8

u = 4 (tous voisins déjà vus)

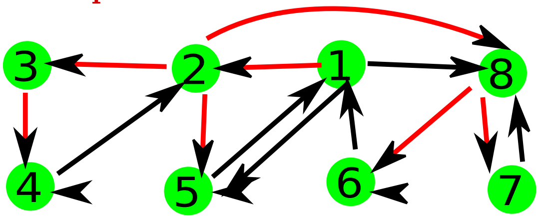
u = 8 (tous voisins déjà vus)

Complexité en temps : On simplifie en supposant que le graphe a n sommets et m arretes

lignes	Listes	Matrices
1 à 4	$O(1)$	$O(1)$
5 à 8	$O(n)$	$O(n)$
9 à 11	$O(1)$	$O(1)$
15 à 19	$O(1)$	$O(1)$
14	$O(nbVois)$ ou au pire $O(n)$	$O(n)$
14 à 19	$O(nbVois)$	$O(n)$
12	n itérations	n itérations
12 à 19	$O(m)$	$O(n^2)$

Parcours en profondeur : On part dans une direction et tant qu'on peut on avance. Quand on est coincé on recule d'un cran et on voit si il y a une autre direction.

Exemple :



vu	V	V	V	V	V	V	V	V
pere	-	1	2	3	2	8	8	2
d	1	2	3	4	7	10	12	9
f	16	15	6	5	8	11	13	14

Pile des appels récursifs :

u $= 7 \rightarrow 6$(vu), 8(vu) FINI

u $= 6 \rightarrow 1$(vu), 5(vu) FINI

u $= 8 \rightarrow \bar{6}$,$\bar{7}$ FINI

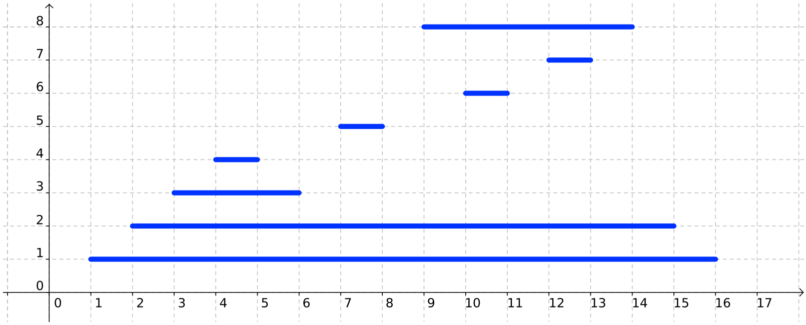
u $= 5 \rightarrow 1$(vu), 4(vu) FINI

u $= 4 \rightarrow 2$(vu) FINI

u $= 3 \rightarrow \bar{4}$ FINI

u $= 2 \rightarrow \bar{3}$,$\bar{5}$,$\bar{8}$ FINI

u $= 1 \rightarrow \bar{2}$,5(vu),8(vu) FINI temps : 16

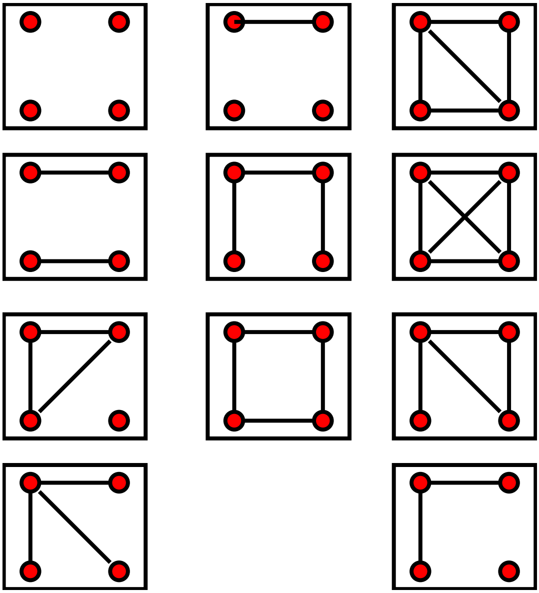


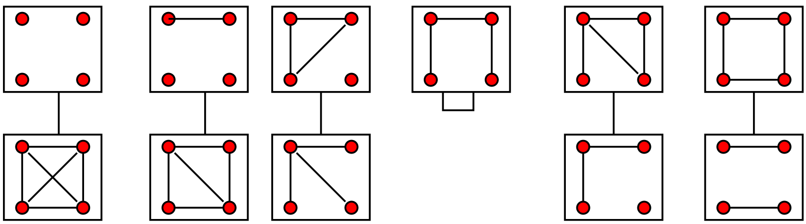
Question 1

```
Algo listes_vers_matrice

Entrée : Adj Tableau[1..n] de listes chainées
Sortie : M Matrice[1..n][1..n]

Debut

  Initialiser M à 0
  Pour i de 1 à n Faire
    Pour j dans Adj[i] Faire
      M[i][j] = 1
    FinPour
  FinPour

Fin
```

```
Algo matrice_vers_liste

Sortie : M Matrice[1..n][1..n]
Entrée : Adj Tableau[1..n] de listes chainées


Debut

  Initialiser M à 0
  Pour i de 1 à n Faire
    Pour j de 1 à n Faire
      Si M[i][j] == 1 Alors
        Ajouter(j,Adj[i])
    FinPour
  FinPour

Fin
```

Question 2





Question 3

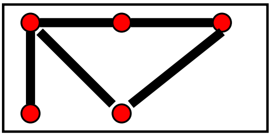
1. K_n possède $\frac{n(n-1)}{2}$ arêtes.

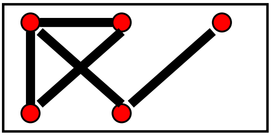
2. $\bar{K}_n$ possède 0 arête

3. $O(n^2)$

Question 4 Il y a deux sous graphes isomorphes à K_3 et aucun isomorphe à $\bar{K}_3$

Question 5





Question 6

1. .

```
Algo EstTransitif

Entrée : G = (S,A) un graphe

Debut

  Pour Chaque sommet u de S Faire
    Pour Chaque voisin v de u Faire
      Pour Chaque voisin w de v Faire
        Si non(adjacent(u,w)) Alors
          Retourne Faux
        Fin Si
      Fin Pour
    Fin Pour
  FinPour

  Retourne vrai

Fin
```

2. $O(n^2)$

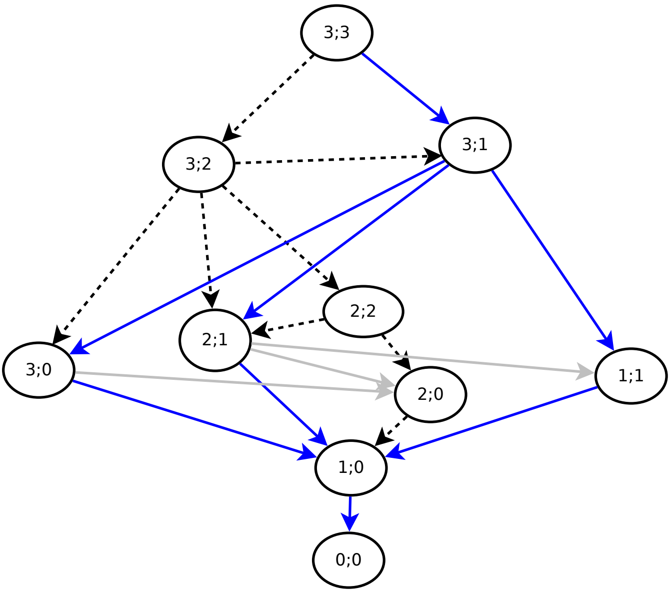
3. .

```
Algo EstTransitif

Entrée : G = (S,A) un graphe

Debut

  Pour Chaque sommet u de S Faire
    Pour Chaque voisin v de u Faire
      Pour Chaque voisin w de v Faire
        Si non(adjacent(u,w)) Alors
          CreerArete(u,w)
        Fin Si
      Fin Pour
    Fin Pour
  FinPour

  Retourne vrai

Fin
```

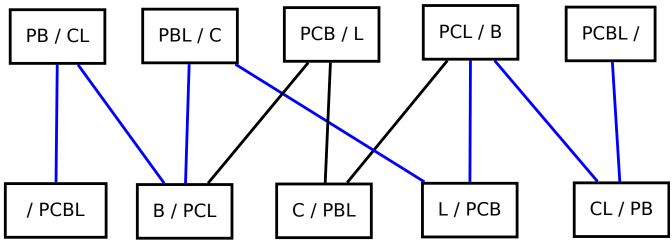
4. $O(n^2)$

5. Faire mieux ???

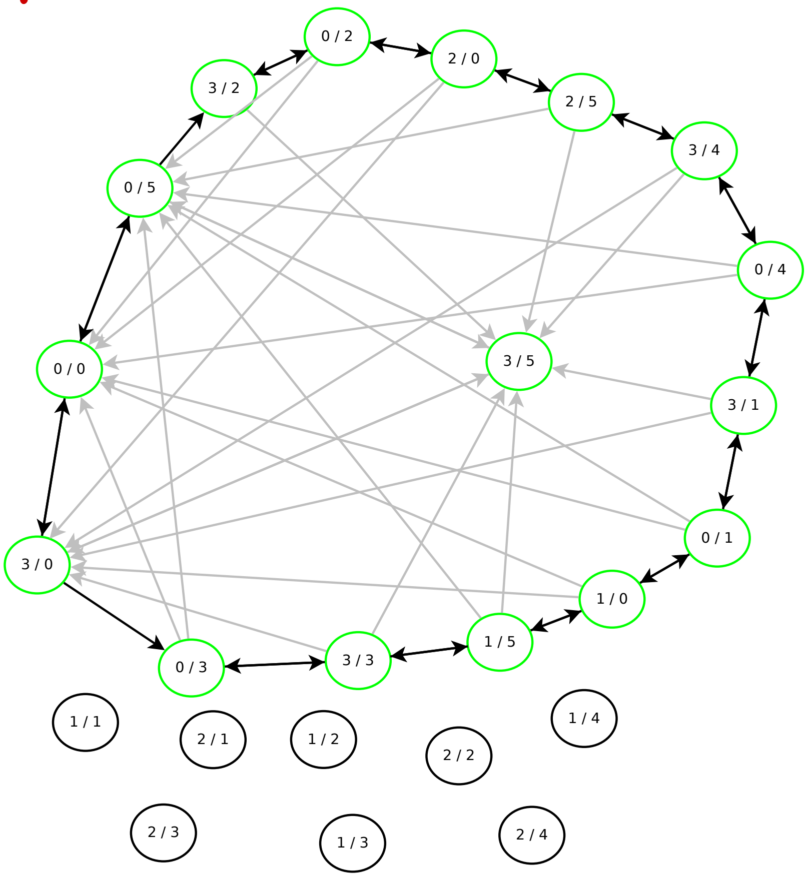
Question 7 Il faut commencer par enlever une allumette



Question 8 P = passeur, C = choux, B = chèvre (bêêê), L = loup

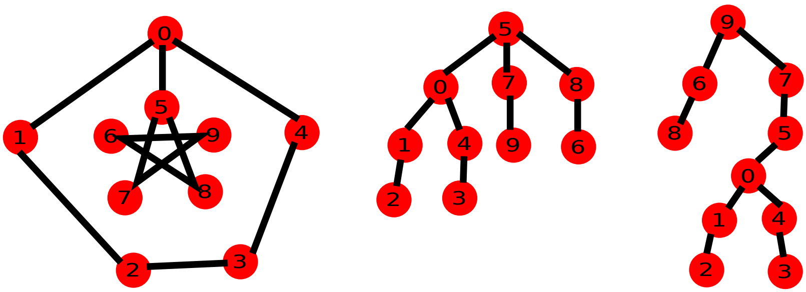


Question 9



Question 1 b

1. Oui (voir contre exemple)
2. Oui (voir contre exemple)
3. Non pour un graphe connexe (le nombre d'arêtes d'un arbre dépend uniquement du nombre de noeud car chaque fois qu'on ajoute un noeud dans un arbre on ajoute une arête (sauf la racine))



Question 2

1. .

```
Entrée : G=(S,A) un graphe non orienté à n sommets
Debut
  lignes 1 à 8 du parcours en largeur (initialisation)
  Pour chaque sommet s de G faire :
    Si vu[s] == Faux alors
      lignes 9 à 19 du parcours en largeur
    FinSi
  FinPour
Fin
```

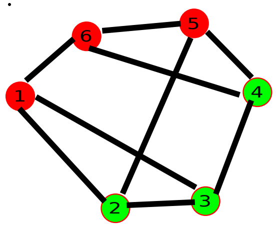
2. La forêt couvrante d'un graphe connexe est un arbre

Question 3 Il suffit de rajouter un compteur dans l'algorithme précédent :

```
Entrée : G=(S,A) un graphe non orienté à n sommets
Debut
  nb = 0
  lignes 1 à 8 du parcours en largeur (initialisation)
  Pour chaque sommet s de G faire :
    Si vu[s] == Faux alors
      nb++
      lignes 9 à 19 du parcours en largeur
    FinSi
  FinPour
  retourner nb
Fin
```

Question 4

1. .



2. Un simple triangle n'est pas biparti

3. On part du parcours en largeur ou en profondeur : on donne une couleur au premier voisin, puis l'autre à tous ces voisins qui n'ont pas de couleur, pour les autres voisins on regarde si ils sont de couleur différente, sinon on s'arrête en répondant non.

```
Algo Est-Biparti
Entrée G = (S,A) un graphe non orienté
Sortie Booléen
Debut
  vu <- tableau de booléens de longueur n
  couleur <- tableau de booléens de longueur n
  F <- File Vide
  Pour chaque sommet s de S Faire
    vu[s] <- faux
  FinPour
  Pour chaque sommet s de S Faire
    Si non(vu[s]) alors
      couleur[s] <- 0
      vu[s] <- vrai
      F.enfiler(s)
      Tant que F n'est pas vide Faire
        u <- F.defiler()
        Pour Chaque sommet v adjacent à u Faire
          Si vu[v] == faux Alors
            vu[v] <- vrai
            couleur[v] <- 1 - couleur[u]
            F.enfile(v)
          Sinon
            Si couleur[v] == couleur[u] Alors
              Retourner Faux
            FinSi
          FinSi
        FinPour
      FinTantQue
    FinSi
  FinPour
  Retourner Vrai
Fin
```

Question 5

```
Algo ContientCycle

Données :
  entraitement : tableau de booleans de taille n
  termine : tableau de booleans de taille n
Entrée : G = (S,A) un graphe
Sortie : Booléen

Debut
  Pour chaque sommet u de G faire
    entraitement[u] <- Faux
    termine[u] <- Faux
  FinPour

  Pour chaque sommet u de G faire
    Si non(termine[u]) Alors
      Tmp <- Visite-PP(G,u)
      Si Tmp alors Retourne Vrai
      FinSi
    FinSi
  FinPour
Fin
```

```
Algo Visite-PP

Entrée : G = (S,A) un graphe, u un sommet
Sortie : Booléen

Debut
  entraitement[u] <- vrai
  Pour chaque sommet v adjacent à u Faire
    Si entraitement[v] nonTermine[v] Alors
      Retourne Vrai
    FinSi
    Si non(entraitement[v]) Alors
      tmp <- Visite-PP(G,v)
      Si tmp alors
        Retourne Vrai
      FinSi
    FinSi
  FinPour
  Retourne Faux
Fin
```

Calcul des CLM : Chemins de Longueur Minimale

1. D'un sommet à un autre (on fait le 2 et on extrait le bon chemin)
2. D'un sommet à tous les autres (Dijkstra)
3. De tous les sommets à un autre (On inverse les arêtes du graphe et on fait 2)
4. De tous les sommets vers tous les sommets (Floyd-Warshall)

Définition : Soit $G = (S, A)$ un graphe et p sa fonction de pondération. Un **Chemin de Longueur Minimale** du sommet u au sommet v est un chemin de $[s_0, \ldots, s_k]$ avec $s_0 = u$, $s_k = v$ et $\sum p(s_i, s_{i+1})$ minimal.

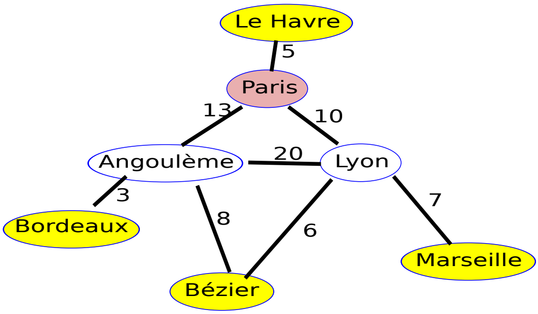
Arbre de CLM : Un arbre couvrant enraciné en un sommet s tel que l'unique chemin de s à un sommet u de cet arbre est un CLM du graphe de départ.

Remarques :

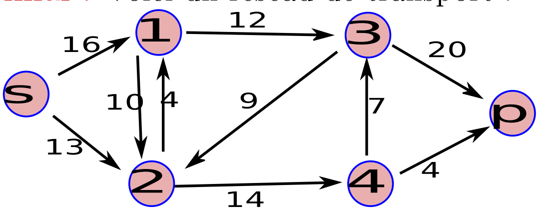
- L'arbre de CLM existe.
- Soit $[s_0, \ldots, s_k]$ un CLM de s_0 à s_k. Alors pour tout $i \in \{1, \ldots, k-1\}$ on a $[s_0, \ldots s_i]$ est un CLM de s_0 à s_i

Algorithme de Dijkstra : Meme principe que l'algo de Prim à la différence que cle[u] est la longueur du plus court chemin connu du sommet de départ au sommet u.

Algorithme de Floyd-Warshall : L'idée est de construire dynamiquement une suite de matrices $M_0, \ldots, M_n$ telles que $m_{k_{i,j}}$ est le cout du plus court chemine de i à j en utilisant comme sommets intermédiaires les sommets $1, 2, \ldots, k$



Problème du Flot Maximal : Voici un réseau de transport :



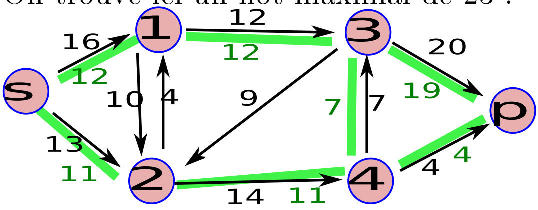
1. Un graphe orienté dont les arêtes sont pondérées positivement par une fonction de capacité
2. Une source à production illimitée
3. Un puit p à consommation illimitée

Exemple :

1. boite 1 : $[s, 2, 4, p]$
2. boite 2 : $[s, 2, 1, 3, p]$

Une contrainte : Aucune accumulation : ce qui entre est égal à ce qui sort

On trouve ici un flot maximal de 23 :

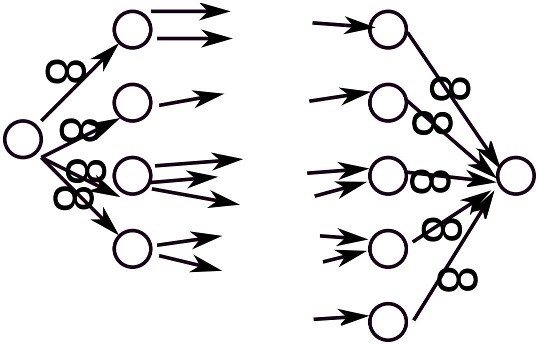


Méthode pour trouver le flot maximal : Un flot sur G est une fonction f qui a chaque arc (u, v) associe une quantité $f(u, v)$ telle que :

1. f respecte les capacités : $f(u, v) \leq c(u, v)$
2. Symétrie : $f(u, v) = -f(v, u)$
3. Conservation du flot : Pour chaque sommet u de G autre que s et p on a $\sum\limits_{v \in S} = 0$ (manière formelle de dire : ce qui entre est égal à ce qui sort).
4. **Flot du réseau :** $F = \sum\limits_{v \in S} f(s, v) = \sum\limits_{v \in S} f(v, p)$

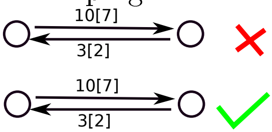
Problème du Flot maximal : Etant donné un réseau de transport G, trouver un flot f qui maximise le flot du réseau.

Remarque : Si il y a m sources et n puits, il ne peut y avoir aucune arête qui entre dans une source ou sort d'un puit :

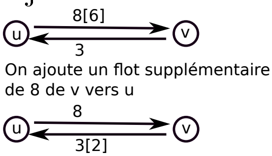


Annulation des flots

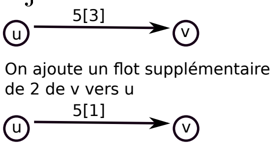
Principe général :



Ajout d'un flux dans le cas de double arête :

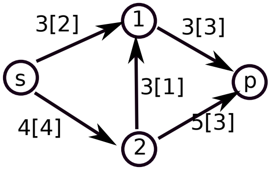


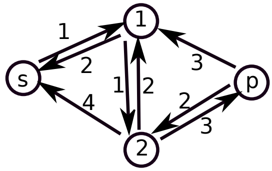
Ajout d'un flux dans le cas d'une simple arête :



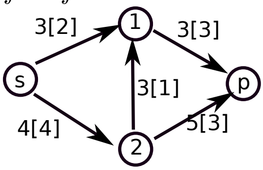
Réseau résiduel : Etant donné un réseau de transport $G = (S, A)$, sa fonction de capacité c et un flot f sur G. Le réseau résiduel est le même graphe muni de la fonction de capacité $c_f = c(u, v) - f(u, v)$

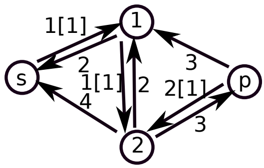
Exemple :

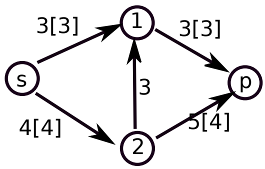




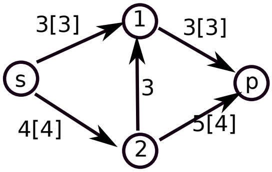
Remarque : Soit f' un flot valide sur le graphe résiduel, alors $f + f'$ est un flot valide sur G.

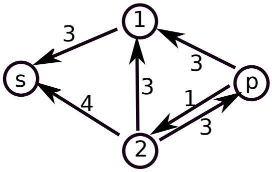






On trace encore le résiduel avec ce nouveau flot :





Propriété : S'il n'existe aucun chemin de s à p dans le graphe résiduel, alors f est maximal.

Algo de Ford-Fulkerson : (1954)

```
Algo Ford-Fulkerson

Entrée: (G,c) un réseau de transport
Sortie: f un flot maximal sur (G,c)

Debut

  //Initialisation d'un flot
  Pour chaque arc (u,v) de G faire
    f(u,v) <- 0
    f(v,u) <- 0
  Fin Pour
  cf = c

  //Ajout de flots valides jusqu'à ce qu'il n'y ait plus de chemin de
  Tant qu'il existe un chemin A de s à p faire
    ajout <- min(cf(u,v) pour tous les arcs de A)
    Pour chaque (u,v) dans A faire
      f(u,v) <- f(u,v) + ajout
      f(v,u) <- -f(u,v)
    Fin Pour
    Mettre à jour cf
  Fin TantQue

  Retourne f
Fin
```