GRAPHES ET ALGORITHMES

3-MIC

M.-J. Huguet

https://homepages.laas.fr/huguet

2020-2021

Objectifs

- **Acquérir des connaissances sur un outil de modélisation**
  - Problèmes discrets / combinatoires
- **Acquérir une connaissance de quelques problèmes classiques de Graphes**
  - Modélisation
  - Méthodes de résolution

Bibliographie

- Précis de Recherche Opérationnelle – 6ème édition – R. Faure, B. Lemaire, C. Picouleau, Dunod, 2009.
- Graphes et Algorithmes – 4ème édition – M. Gondran et M. Minou, Lavoisier, 2009.
- Network Flows : Theory, Algorithms and Applications – K. Ahuja, J. Orlin et T. Magnati – Prentice Hall, 1993
- Graph Theory and its Applications – 2nd edition – J. Gross et J. Yellen – Chapman & Hall, 2005
- MOOC : Conception et mise en oeuvre d'algorithmes, Ecole Polytechnique (2013-2014)

Déroulement

- **UF Graphes - Programmation Objet**
  - 8 Cours et 7 TD de Graphes
    - Intervenants de TD : Mohamed Siala – Valentin Antuori – MJ. Huguet
  - 1 examen écrit

  - Bureau d'Etudes (Projet) : 7 séances de TP + Travail Personnel
    - Développement d'algorithmes corrects et efficaces pour résoudre des problèmes de mobilité
    - Implémentation en Java.
    - Rapport + (Soutenance)

Plan

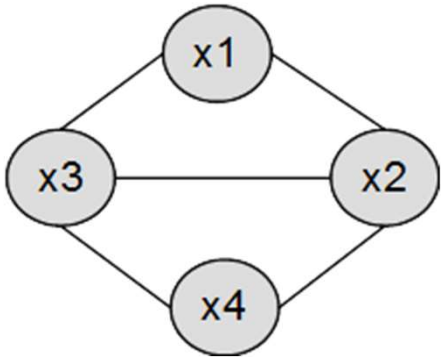
1. **Introduction** **(1 cours)**
   - Généralités
   - Quelques définitions
   - Représentations informatiques
2. **Parcours de Graphe** **(2 cours)**
   - Principe du parcours
   - Parcours en profondeur
   - Parcours en largeur
   - Premières applications d'un algorithme de parcours
     - Connexité – Forte connexité
     - Divers , ….
3. **Optimisation et Graphes**
   - Plus courts chemins (2 cours)
   - Problèmes de flots (2 cours)
   - Quelques problèmes NP-difficiles (1 cours)

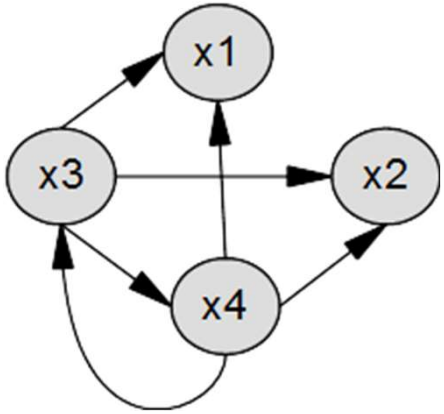
Plan

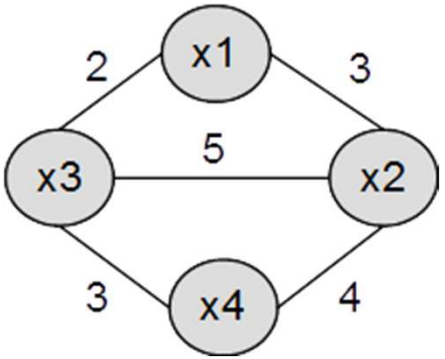
1. **Introduction**
   - Généralités
   - Quelques définitions
   - Représentations informatiques
2. **Parcours de Graphe**
   - Principe du parcours
   - Parcours en profondeur
   - Parcours en largeur
   - Premières applications d'un algorithme de parcours
     - Connexité – Forte connexité
     - Divers ….
3. **Optimisation et Graphes**
   - Plus courts chemins
   - Problèmes de flots

1.1. Généralités (1)

- **Graphes**
  - Modéliser des **relations** entre un ensemble fini **d'entités**
  - Chercher des propriétés, des interactions, ….







  - Entités = **sommets**
  - Relations **binaires** = non orientées / orientées / pondérées

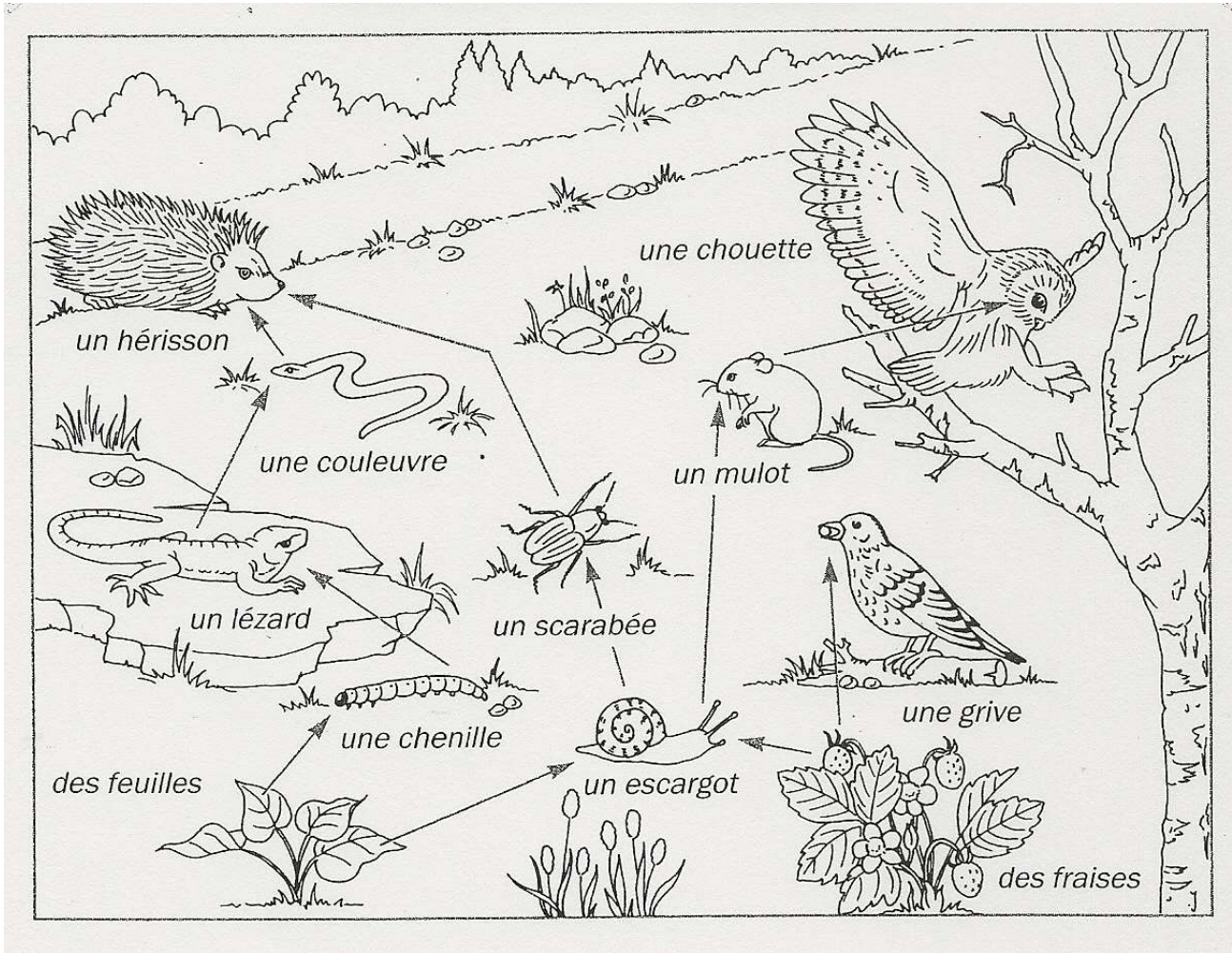
A l'origine des Graphes

- **Leonhard Euler : article 1735**
  - Problème des 7 ponts de Königsberg
    - Trouver une promenade partant d'un point donné, revenant à ce point et passant une fois et une seul par chacun des 7 ponts de la ville

  - Résultat

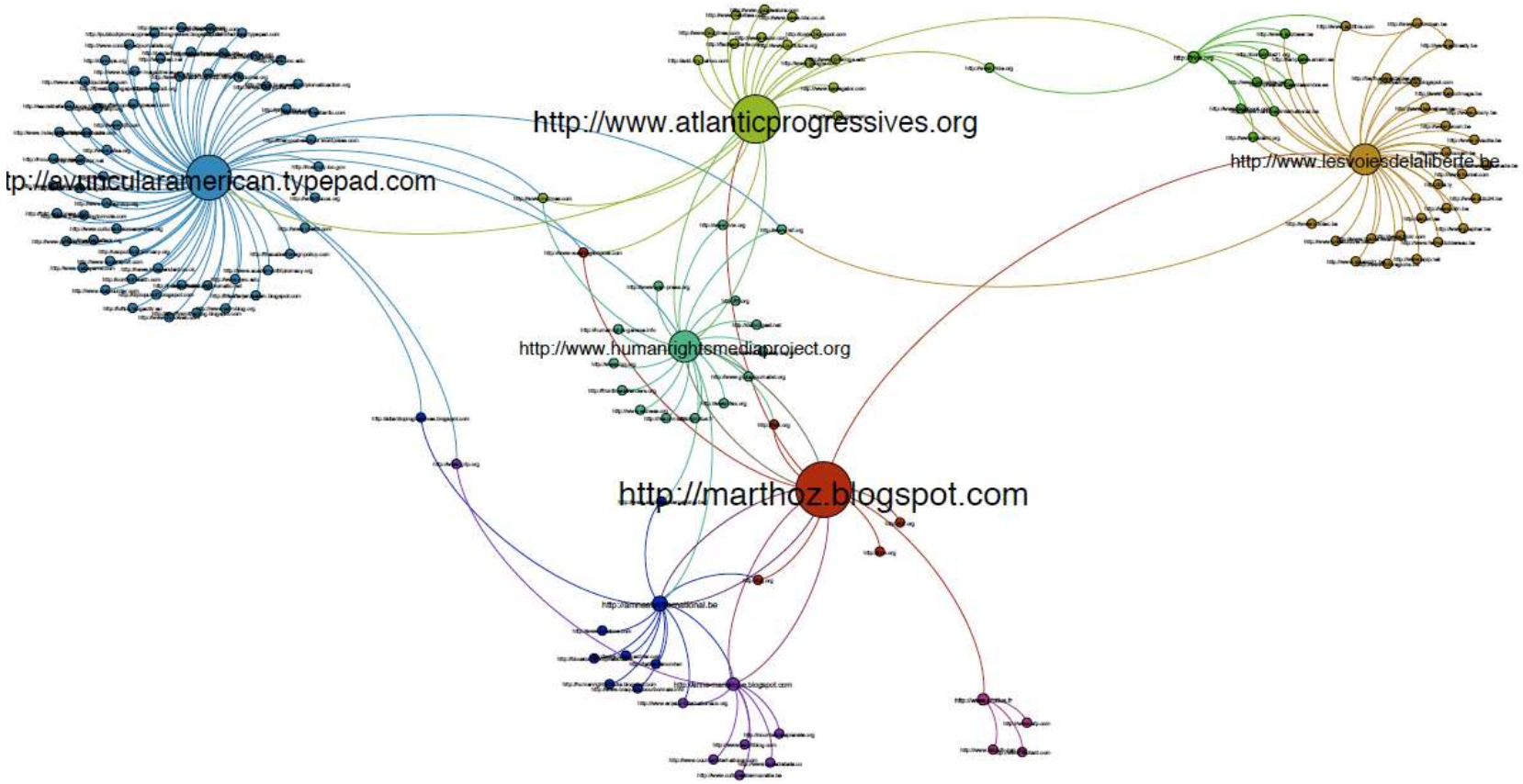
Exemples

- **Chaines alimentaires**



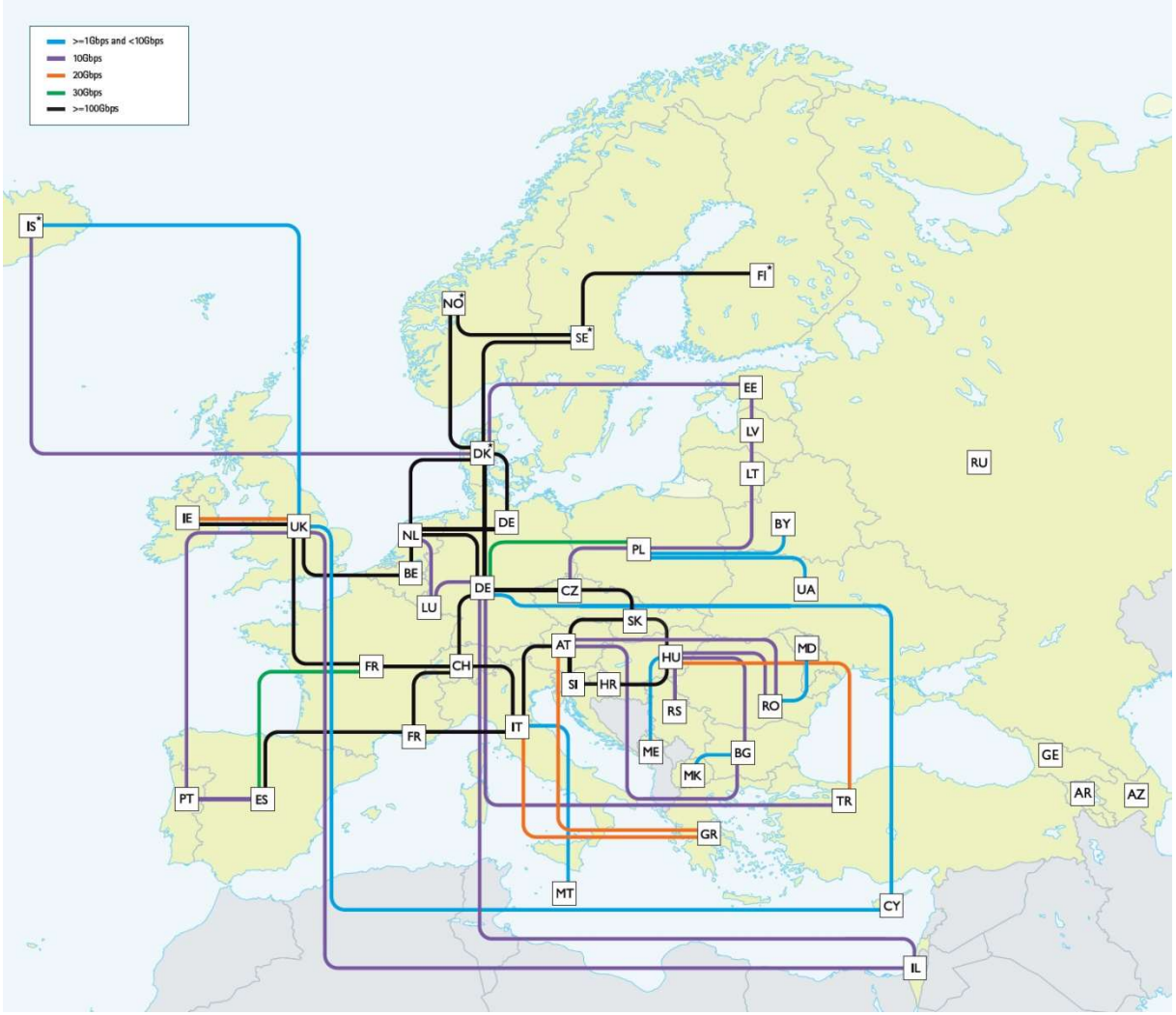
Exemples

- **Réseaux Sociaux**



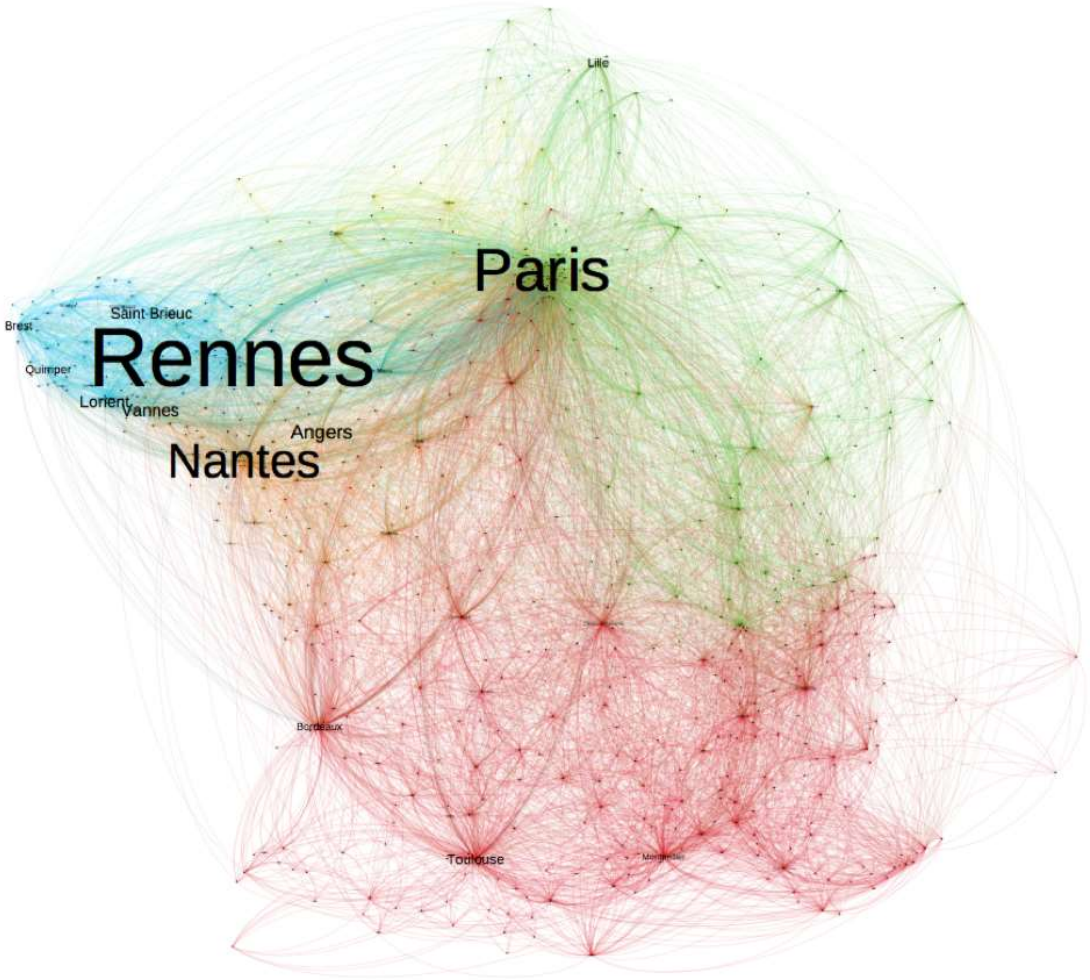
Exemples

- **Réseaux de communication**



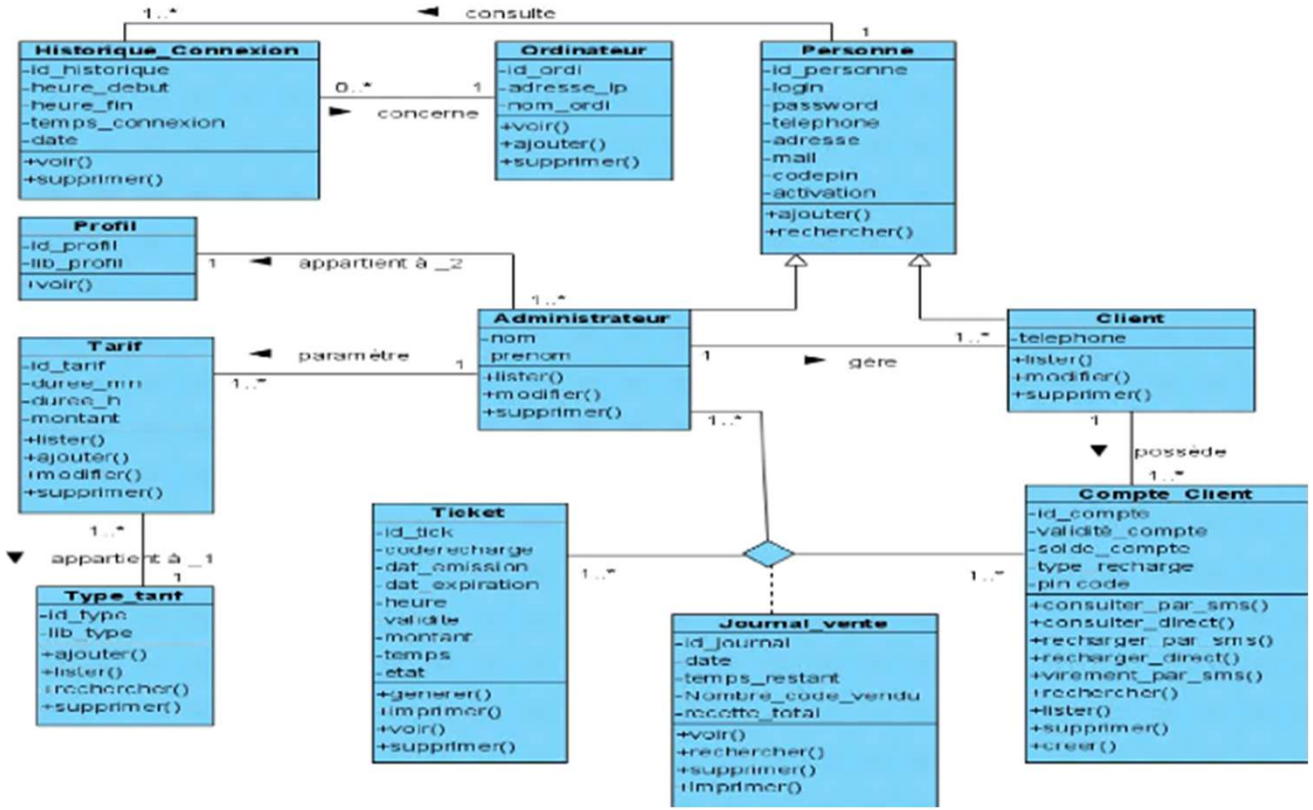
Exemples

- **Réseaux de transport**



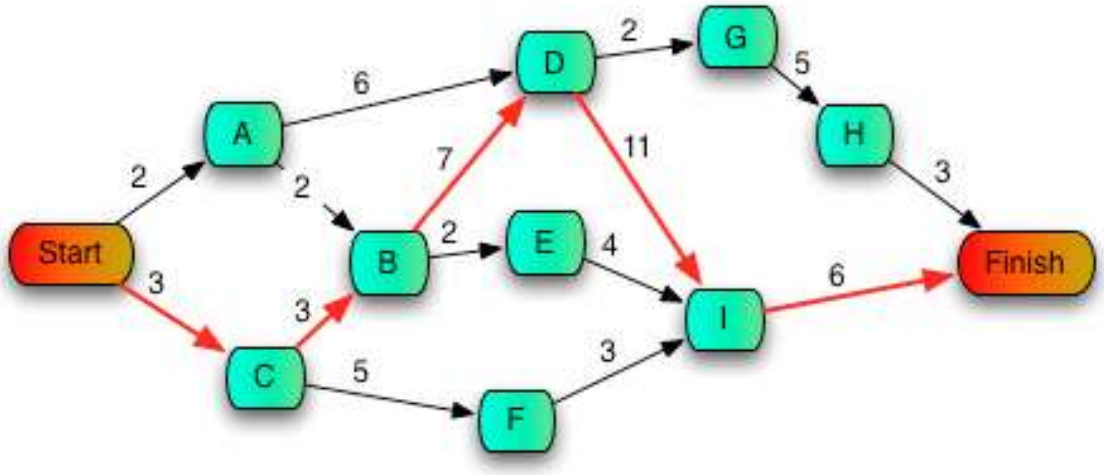
Exemples

- **Diagramme de conception**



Exemples

- **Gestion de projet**



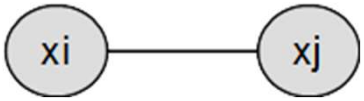
- **Complexité**

Plan

1. **Introduction**
   - Généralités
   - Quelques définitions
   - Représentations informatiques
2. **Parcours de Graphe**
   - Principe du parcours
   - Parcours en profondeur
   - Parcours en largeur
   - Premières applications d'un algorithme de parcours
     - Connexité – Forte connexité
     - Divers
3. **Optimisation et Graphes**
   - Plus courts chemins
   - Problèmes de flots

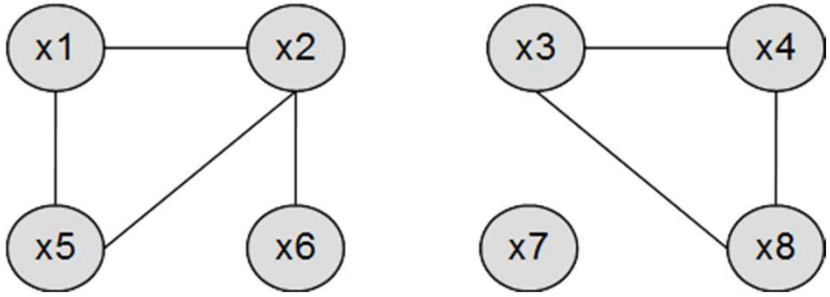
1.2. Définitions –
Graphes non orientés (1)

- **Graphe : $G(X, A)$**
  - Ensemble des nœuds (sommets) : $X = \{x_1, \dots, x_n\}$ nodes/vertices
  - Ensemble des arêtes: $A = \{e_1, \dots, e_m\}$ edge
    - Arête : relation binaire : $e_k = (x_i, x_j)$



- Un graphe G est non orienté si
  - $\forall\, x_i, x_j \in X \times X,\ (x_i, x_j) \in A \Leftrightarrow (x_j, x_i) \in A$
  - La relation binaire caractérisée par A est **symétrique**

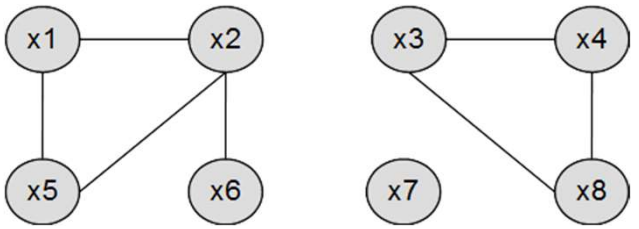
  - Exemple



- $X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$
- $A = \{(x_1, x_2), (x_2, x_1), (x_1, x_5), (x_5, x_1),$
$(x_2, x_5), (x_5, x_2), (x_2, x_6), (x_6, x_2)$
$(x_3, x_4), (x_4, x_3), (x_3, x_8), (x_8, x_3),$
$(x_4, x_8), (x_8, x_4)\}$

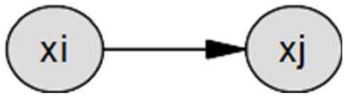
1.2. Définitions – Graphes non orientés (2)

- Graphes finis
  - Nombre de sommets : n Nombre d'arêtes : m
- Sommet adjacent / voisin
  - x_i est un sommet adjacent à x_j si $(x_i, x_j) \in A$
  - Ensemble des sommets adjacents à x_i : $\delta(x_i) = \{x_j | (x_i, x_j) \in A\}$
- Degré d'un sommet x_i :
  - nombre de sommets adjacents : $d(x_i) = |\delta(x_i)|$
  - Propriété : $\sum_{i=1}^{n} d(x_i) = 2m$



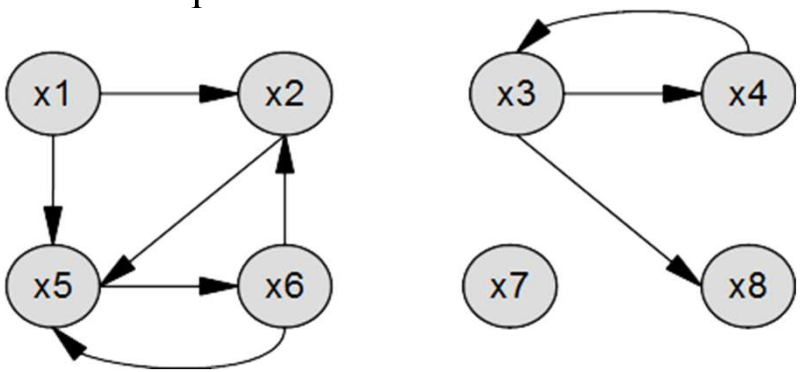
1.2. Définitions – Graphes orientés (1)

- **Graphe : $G(X, A)$**
  - Ensemble des nœuds (sommets) : $X = \{x_1, \dots, x_n\}$
  - Ensemble des arcs: $A = \{a_1, \dots, a_m\}$
    - Arc: relation binaire orientée: $a_k = (x_i, x_j)$



- Un graphe G est orienté si
  - $\exists\, x_i, x_j \in X \times X, \quad (x_i, x_j) \in A$ et $(x_j, x_i) \notin A$
  - La relation binaire caractérisée par A **n'est pas symétrique**

- Exemple



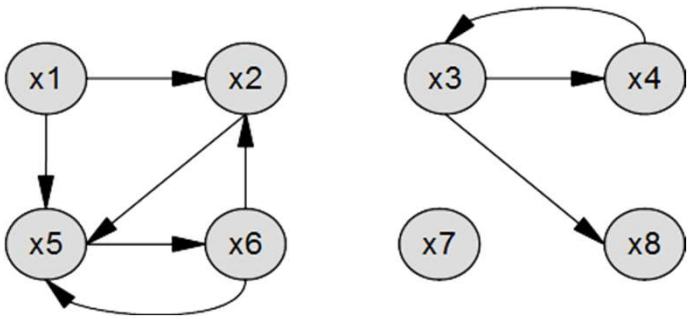
- $X = \{x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8\}$
- $A = \{(x_1, x_2), (x_1, x_5), (x_2, x_5), (x_5, x_6), (x_6, x_5), (x_3, x_4), (x_3, x_8), (x_4, x_3)\}$

1.2. Définitions – Graphes orientés (2)

- Sommet successeur :
  - x_j est un sommet successeur de x_i si $(x_i, x_j) \in A$
  - Ensemble des sommets successeurs de x_i : $\delta^+(x_i) = \{x_i | (x_i, x_j) \in A\}$
- Sommet prédécesseur :
  - x_i est un sommet prédécesseur de x_j si $(x_i, x_j) \in A$
  - Ensemble des sommets prédécesseurs de x_j : $\delta^-(x_j) = \{x_i | (x_i, x_j) \in A\}$
- Degré d'un sommet x_i :
  - **Demi-degré sortant** = nombre de sommets successeurs : $d^+(x_i) = |\delta^+(x_i)|$
  - **Demi-degré entrant** = nombre de sommets prédécesseurs : $d^-(x_i) = |\delta^-(x_i)|$
  - **Degré** : $d(x_i) = d^+(x_i) + d^-(x_i)$
  - Propriété :

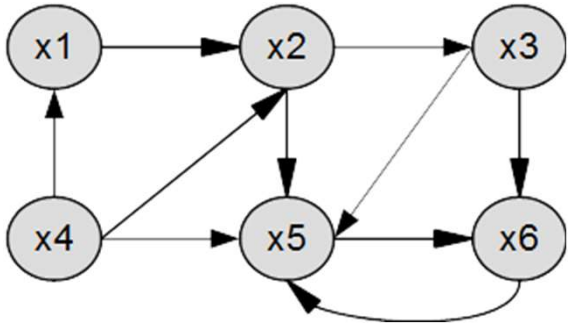
$$\sum_{i=1}^{n} d^+(x_i) = \sum_{i=1}^{n} d^-(x_i) = m$$

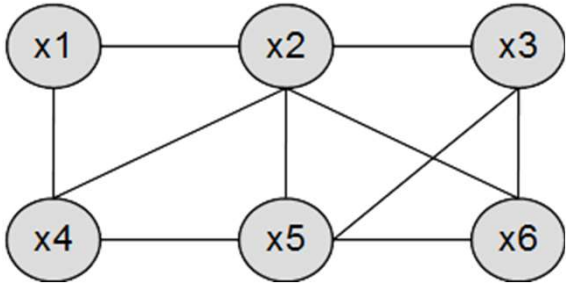
$$\sum_{i=1}^{n} d(x_i) = 2m$$

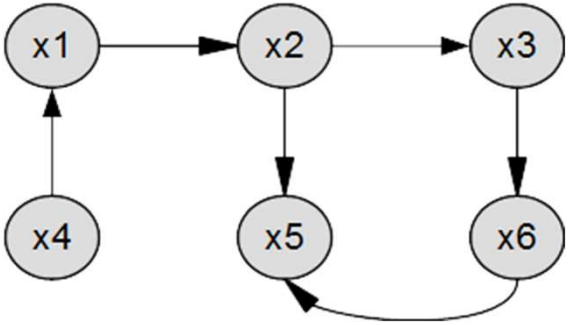


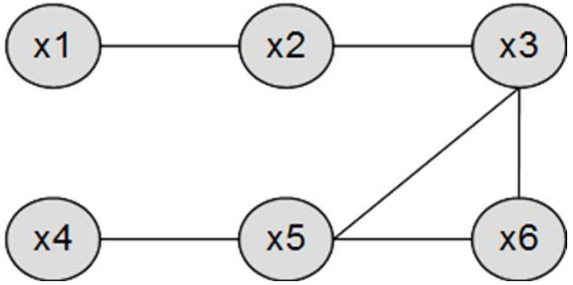
1.2. Définitions – Graphes partiels

- Soit un graphe $G(X, A)$
  - $G'(X, A')$ est un **graphe partiel** de G si $A' \subseteq A$



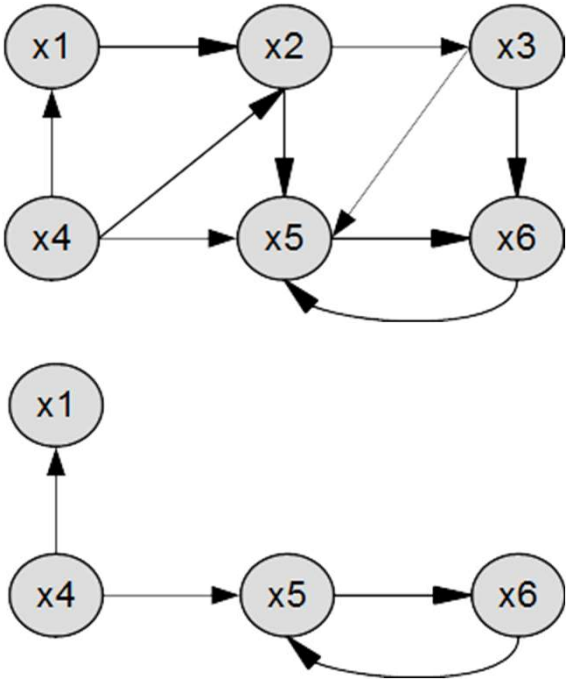


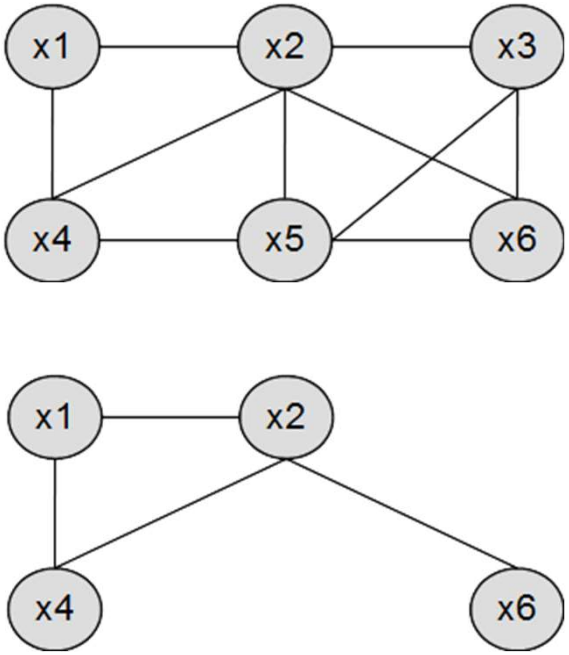




1.2. Définitions – Sous Graphes

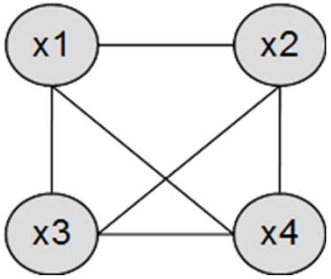
- Soit un graphe $G(X, A)$
  - $G'(X', A')$ est un **sous-graphe** de G si $X' \subseteq X$ et $A' = A \cap \{X' \times X'\}$
  - On dit que G' est **induit** de G par X'





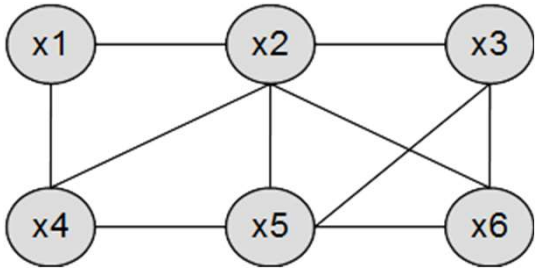
1.2. Définitions – Graphes Complets et Cliques

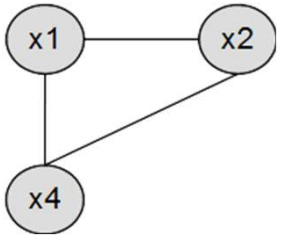
- **Graphes non orientés**
  - Graphe complet :
    - Toutes les arêtes sont présentes :
      - $A = \{(x_i, x_j), \forall x_i, x_j \in S,\ x_i \neq x_j\}$

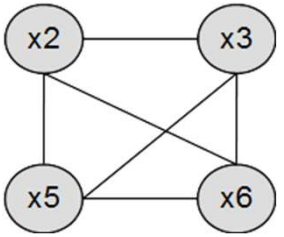


    - Les sommets sont tous deux à deux adjacents

  - Clique : un sous graphe complet



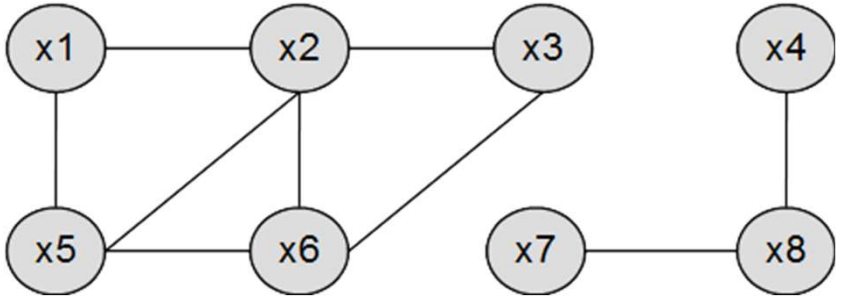




    - **Clique maximale** : clique ayant le plus grand nombre de sommets

1.2. Définitions – Cheminements et connexité / Cas non orienté

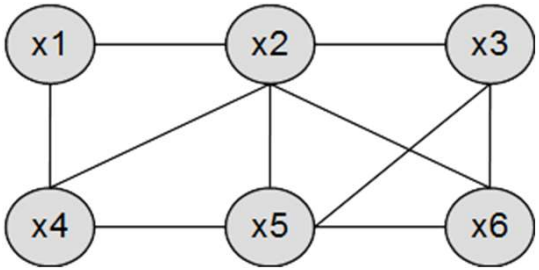
- Soit un graphe **non orienté** $G(X, A)$
  - **Chaine** : séquence de sommets $< x_1, x_2, \ldots, x_k >$ telle que $(x_{i-1}, x_i) \in A$
  - **Longueur** d'une chaine : nombre d'arêtes
  - **Chaine élémentaire** : tous les sommets sont distincts
  - **Cycle** : chaine commençant et terminant par un même sommet

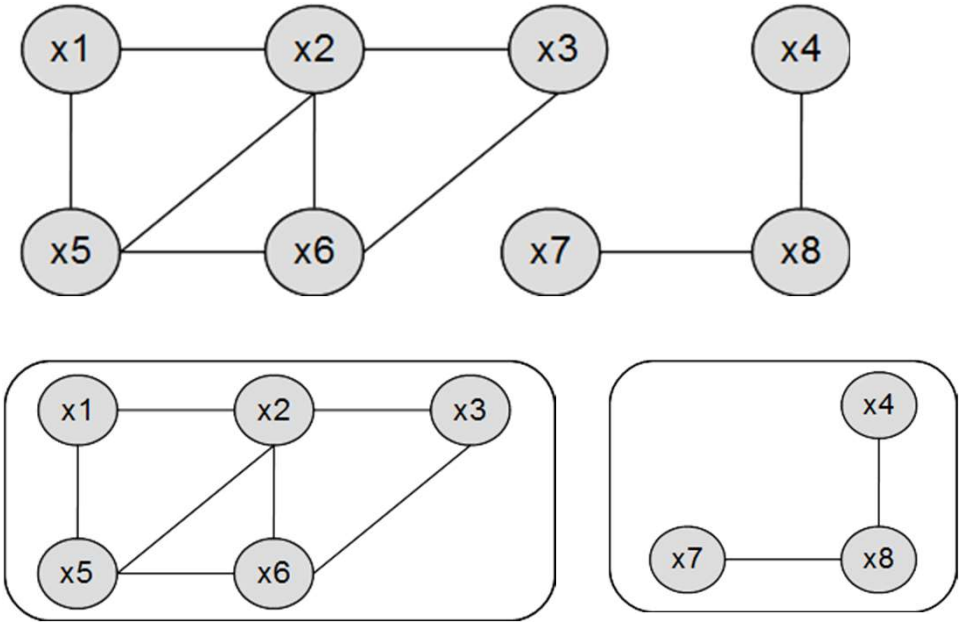


- $< x_2, x_1, x_5, x_2, x_6, x_3 >$: chaine de longueur 5
- $< x_2, x_1, x_5, x_6 >$: chaine élémentaire (longueur 3)
- $< x_1, x_5, x_2, x_1 >$: cycle (longueur 3)

1.2. Définitions – Cheminements et connexité / Cas non orienté

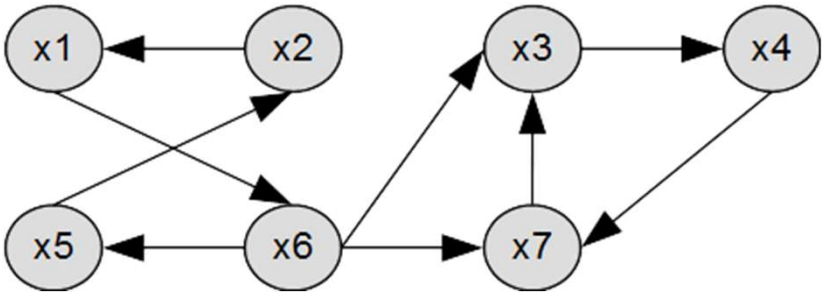
- Soit un graphe **non orienté** $G(X, A)$
  - Le graphe G est **connexe** si
    - $\forall(x_i, x_j) \in A$ il existe une chaine entre x_i et x_j ou $x_i = x_j$
  - **Composante connexe** de G : sous graphe **connexe et maximal**
    - Classe d'équivalence (relation réflexive, symétrique et transitive)





1.2. Définitions – Cheminements et connexité / Cas orienté

- Soit un graphe **orienté** $G(X, A)$
  - **Chemin**: séquence de sommets $< x_1, x_2, \ldots, x_k >$ telle que $(x_{i-1}, x_i) \in A$
  - **Longueur** d'une chemin: nombre d'arcs
  - **Chemin élémentaire** : tous les sommets sont distincts
  - **Circuit** : chemin commençant et terminant par un même sommet

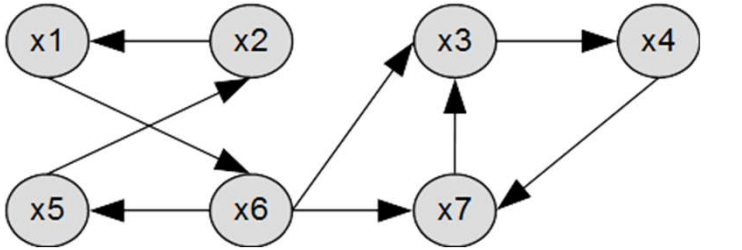


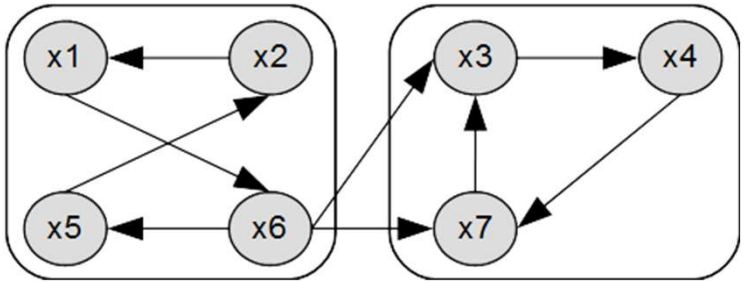
- $< x_2, x_1, x_6, x_3, x_4, x_7, x_3 >$: chemin de longueur 6
- $< x_2, x_1, x_6, x_5 >$: chemin élémentaire (longueur 3)
- $< x_1, x_6, x_5, x_2, x_1 >$: circuit (longueur 4)

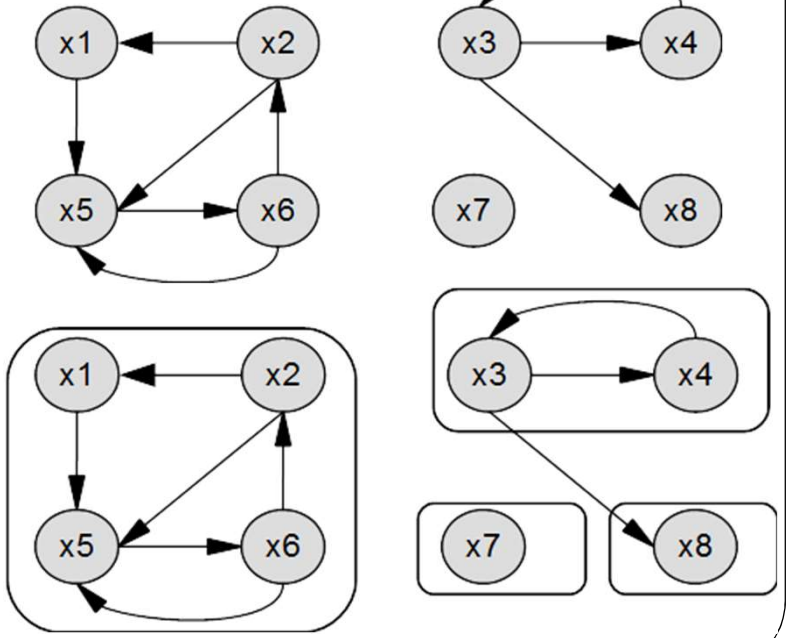
1.2. Définitions – Cheminements et connexité / Cas orienté

- Soit un graphe **orienté** $G(X, A)$
  - Le graphe G est **fortement connexe** si
    - $\forall(x_i, x_j) \in A$ il existe un chemin entre x_i et x_j ou $x_i = x_j$

  - **Composante fortement connexe** de G :
    - sous graphe **fortement connexe et maximal**
      - **Classe d'équivalence**

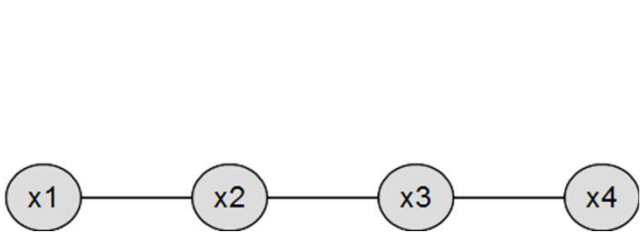


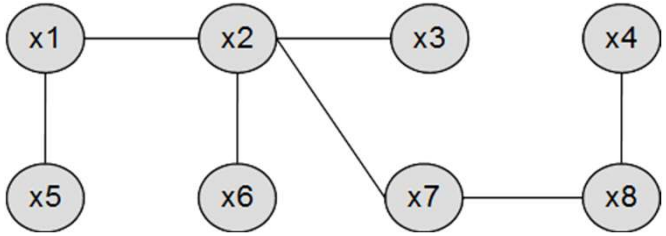


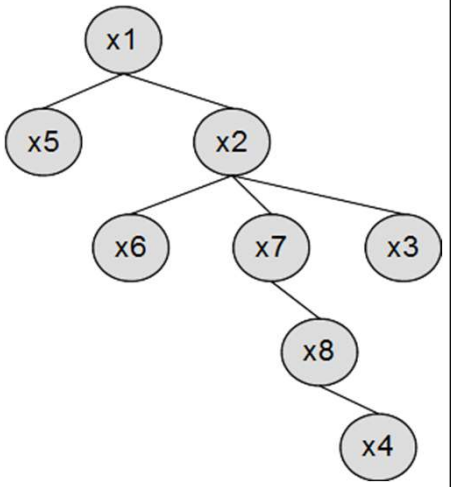


1.2. Définitions – Arbres

- Le graphe **non orienté** $G(X, A)$ **est un arbre si**
  - G est connexe et sans cycle
  - G est (sans cycle)/(connexe) et comporte $(|X| - 1)$arêtes
  - G est sans cycle et en ajoutant une arête, on crée un cycle élémentaire
  - G est connexe et en supprimant une arête, le graphe n'est plus connexe
  - Pour toute paire de sommets (x_i, x_j), il existe <u>exactement</u> une chaine les reliant



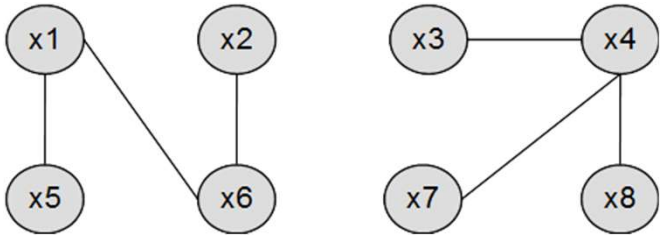




- **Feuilles** : sommets de degré 1
- **Nœuds internes** : sommet de degré > 1

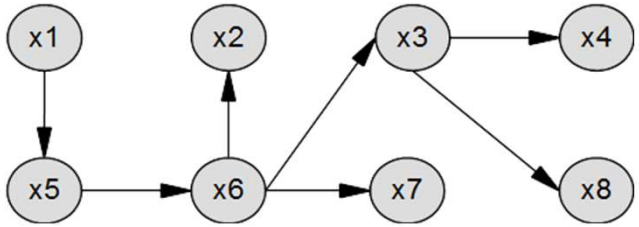
1.2. Définitions – Arbres

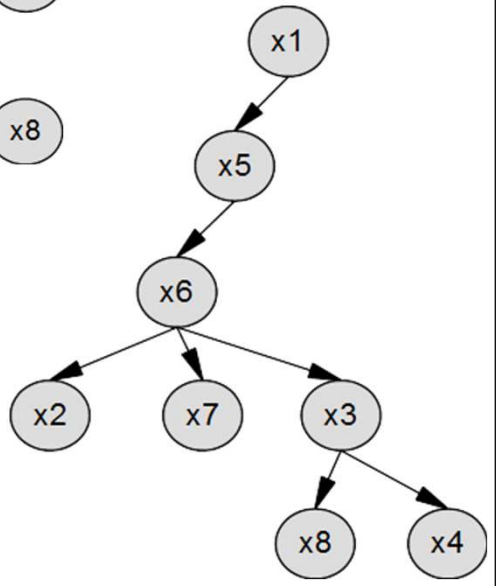
- Le graphe **non orienté** $G(X, A)$ **est une forêt**
  - si chacune de ses composantes connexes est un arbre



1.2. Définitions – Arborescence

- Le graphe orienté $G(X, A)$ est une **arborescence**
  - G a un sommet racine x_k tel que pour tout autre sommet x_i il existe un chemin unique de x_k vers x_i





  - **Racine** : sommet de degré entrant 0
  - **Feuilles** : sommets de degré sortant 0
  - Arbre k-aire : chaque sommet a au plus k successeurs

Plan

1. **Introduction**
   - Généralités
   - Quelques définitions
   - Représentations informatiques
2. **Parcours de Graphe**
   - Principe du parcours
   - Parcours en profondeur
   - Parcours en largeur
   - Premières applications d'un algorithme de parcours
     - Connexité – Forte connexité
     - Divers
3. **Optimisation et Graphes**
   - Plus courts chemins
   - Problèmes de flots

1.3. Représentations informatiques – Exemple

- Soit un graphe $G(X, A)$:

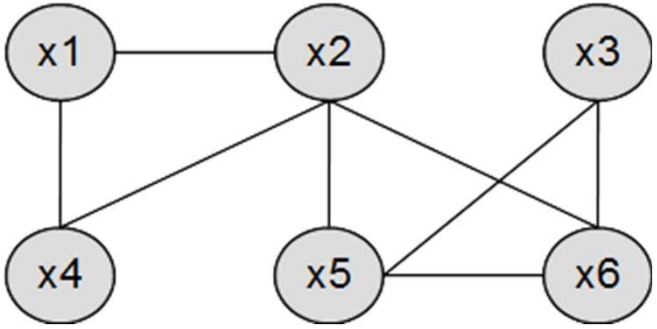
```
Afficher_Relations(G)
  Pour tout sommet x_i ∈ X faire
    Afficher_Successeur(G, x_j)
  Fin pour

Afficher_Successeur(G, x_i)
  Pour tout sommet x_j ∈ δ(x_i) (ie. (x_i, x_j) ∈ A) faire
    Afficher_Sommet(x_j)
  Fin pour
```

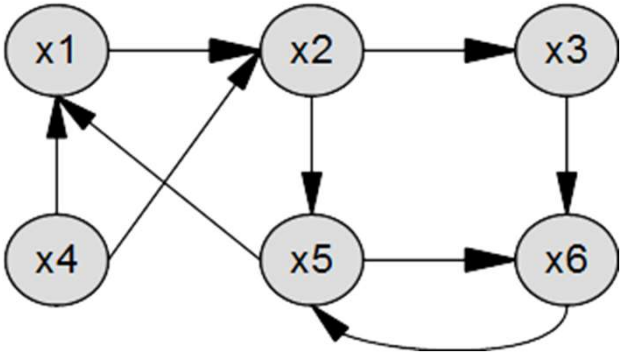
- Quelle est la complexité de ces algorithmes ?
  - Dépend des structures de données pour représenter un graphe

1.3. Représentations informatiques – Matrice d'Adjacence

- Matrice d'adjacence M
  - Graphes $G(X, A)$ orientés et non orientés
    - $M_{ij} = 1$ si $(x_i, x_j) \in A$ et $M_{ij} = 0$ sinon



M	x1	x2	x3	x4	x5	x6
x1	0	1	0	1	0	0
x2	1	0	0	1	1	1
x3	0	0	0	0	1	1
x4	1	1	0	0	0	0
x5	0	1	1	0	0	1
x6	0	1	1	0	1	0



M	x1	x2	x3	x4	x5	x6
x1	0	1	0	0	0	0
x2	0	0	1	0	1	0
x3	0	0	0	0	0	1
x4	1	1	0	0	0	0
x5	1	0	0	0	0	1
x6	0	0	0	0	1	0

1.3. Représentations informatiques – Matrice d'Adjacence

- Taille de la donnée
  - $\Theta(n^2)$ avec n le nombre de sommets
  - Indépendant du nombre de relations m

- Complexité de `Afficher_Successeur(`G, x_i`)`

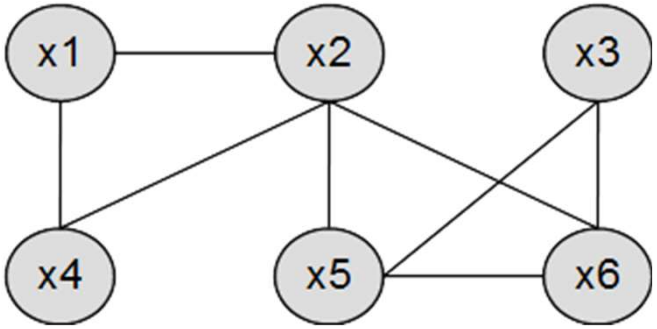
```
Pour tout sommet x_j ∈ δ(x_i) (ie. (x_i, x_j) ∈ A) faire
   Afficher_Sommet(x_j)
Fin pour
```

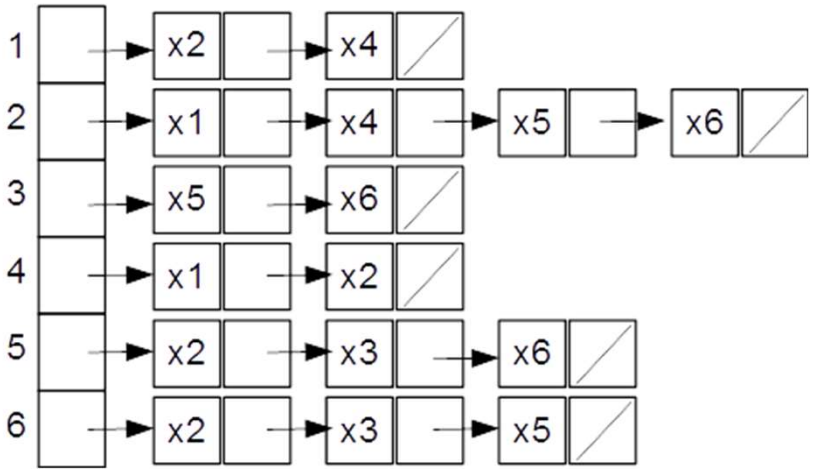
- Complexité de `Afficher_Relations(`G`)`

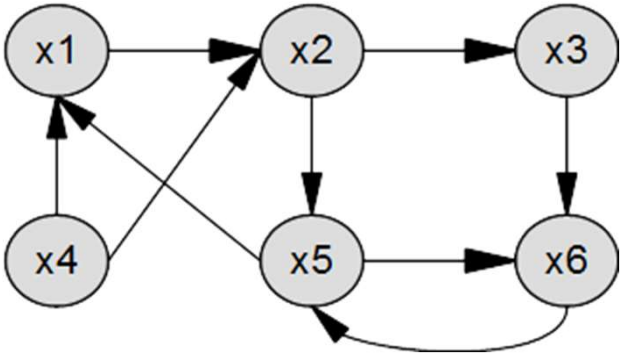
```
Pour tout sommet x_i ∈ E faire
   Afficher_Successeur(G, x_j)
Fin pour
```

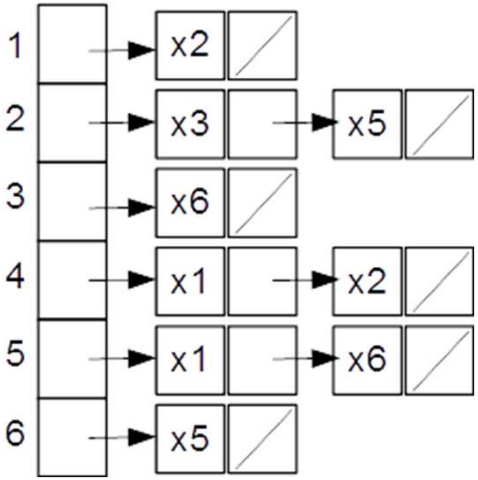
1.3. Représentations informatiques – Liste d'Adjacence

- Liste d'adjacence / Successeur
  - Tableau Tab_Succ tel que Tab_Succ_i = liste des successeurs (voisins) de x_i









1.3. Représentations informatiques – Liste d'Adjacence

- Taille de la donnée
  - $\Theta(n+m)$ avec n le nombre de sommets et m le nombre d'arcs

- Complexité de `Afficher_Successeur(`G, x_i`)`

```
Pour tout sommet x_j ∈ δ(x_i) (ie. (x_i, x_j) ∈ A) faire
   Afficher_Sommet(x_j)
Fin pour
```

- Complexité de `Afficher_Relations(`G`)`

```
Pour tout sommet x_i ∈ E faire
   Afficher_Successeur(G, x_j)
Fin pour
```

1.3. Représentations informatiques – Comparaison

- Intérêt Liste d'adjacence
  - Représentation plus compacte d'un graphe si $m \ll n^2$ (graphe peu dense)
  - On ne mémorise pas l'absence de voisins / successeurs
  - Remarque : nombre de relations
    - d'un graphe non orienté complet : $\frac{n(n-1)}{2}$ $(\Theta(n^2))$
    - d'un graphe orienté complet : $n(n-1)$ $(\Theta(n^2))$

- Complexité temporelle des opérations élémentaires

	Mat Adj	Liste Adj
Ajouter relation (x_i, x_j)	$\Theta(1)$	$\Theta(1)$
Supprimer relation (x_i, x_j)	$\Theta(1)$	$\Theta(d(x_i))$
Tester existence relation (x_i, x_j)	$\Theta(1)$	$\Theta(d(x_i))$
Parcourir tous les sommets :	$\Theta(n)$	$\Theta(n)$
Parcourir les voisins d'un sommet x_i	$\Theta(n)$	$\Theta(d(x_i))$

1.3. Représentations informatiques – Comparaison

	Matrice Adjacence	Liste Adjacence
Avantages	Implémentation simple Accès direct à l'information $\Theta(1)$	Occupation mémoire Accès voisinage efficace : $\Theta(d(x_i))$ Accès information assez rapide $\Theta(d(x_i))$
Inconvénients	Occupation mémoire Accès voisinage long : $\Theta(n)$ Inadapté aux graphes dynamiques	Implémentation

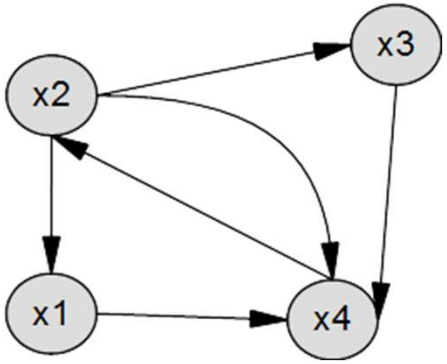
- Passage entre les deux représentations :
  - Matrice d'adjacence ➔ Liste d'adjacence : $\Theta(n^2)$
  - Liste d'adjacence ➔ Matrice d'adjacence : $\Theta(m)$

Question (1)

- Comment représenter un graphe pondéré ?

Question (2)

- Soit le graphe $G(X, A)$ représenté par sa matrice d'adjacence M
  - Que représente la matrice M^2 ?



M	x1	x2	x3	x4
x1	0	0	0	1
x2	1	0	1	1
x3	0	0	0	1
x4	0	1	0	0

M²	x1	x2	x3	x4
x1	0	1	0	0
x2	0	1	0	2
x3	0	1	0	0
x4	1	0	1	1

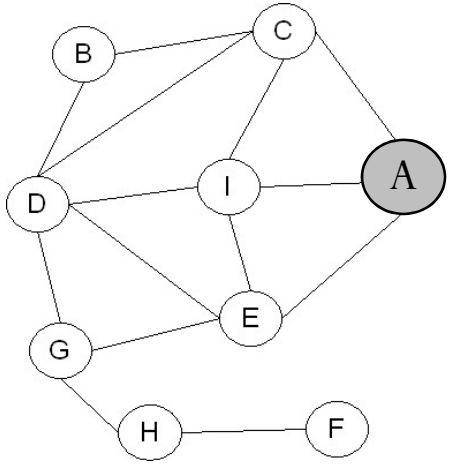
Complexité calcul M² ?

Plan

1. **Introduction**
   - Généralités
   - Quelques définitions
   - Représentations informatiques
2. **Parcours de Graphe**
   - Principe du parcours
   - Parcours en profondeur
   - Parcours en largeur
   - Premières applications d'un algorithme de parcours
     - Connexité – Forte connexité
     - Divers
3. **Optimisation et Graphes**
   - Plus courts chemins
   - Problèmes de flots

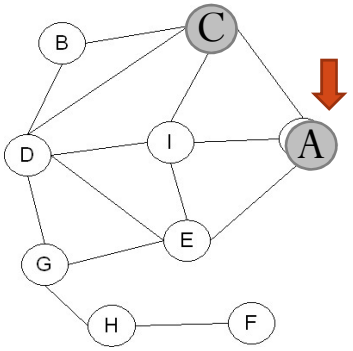
2.1. Principe du Parcours (1)

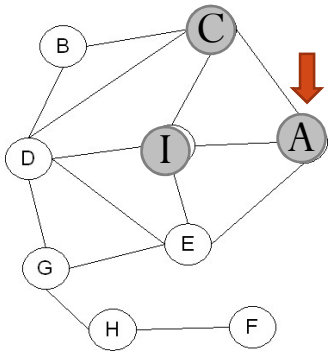
- **Parcours d'un graphe non orienté ou orienté**
  - Visiter tous les sommets accessibles depuis un sommet de départ
  - Base de nombreux algorithmes de graphe
- **Principe général d'un parcours**
  - Marquage des sommets : 3 états possibles
    - Blanc (inexploré) : sommet non visité
    - Gris (en traitement) : sommet en cours d'exploration
    - Noir (terminé) : sommet dont l'exploration est terminée
  - Initialement
    - Sommet de départ : gris
    - Autres sommets : blanc

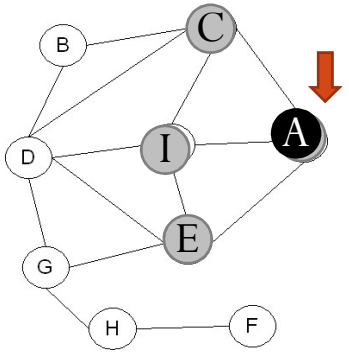


2.1. Principe du Parcours (2)

- Principe général
  - A chaque étape :
    - Sélectionner un sommet gris
    - Si tous ses voisins sont gris ou noir ➔ il devient noir
    - Sinon : marquage d'un voisin blanc qui devient gris
  - Arrêt : tous les sommets accessibles sont noirs







2.1. Principe du Parcours – Mise en œuvre (1)

- Deux types de parcours
  - Structure de mémorisation des sommets gris
    - Pile : Last In First Out (LIFO)
    - File : First In First Out (FIFO)
  - Deux possibilités de sélection du sommet gris
    - Pile : Sélectionner le sommet passé GRIS en dernier
      - **Parcours en profondeur (Depth First Search)**
    - File : Sélectionner le sommet passé GRIS depuis le plus longtemps
      - **Parcours en largeur (Breadth First Search)**

Plan

1. **Introduction**
   - Généralités
   - Quelques définitions
   - Représentations informatiques
2. **Parcours de Graphe**
   - Principe du parcours
   - Parcours en profondeur
   - Parcours en largeur
   - Premières applications d'un algorithme de parcours
     - Connexité – Forte connexité
     - Divers
3. **Optimisation et Graphes**
   - Plus courts chemins
   - Problèmes de flots

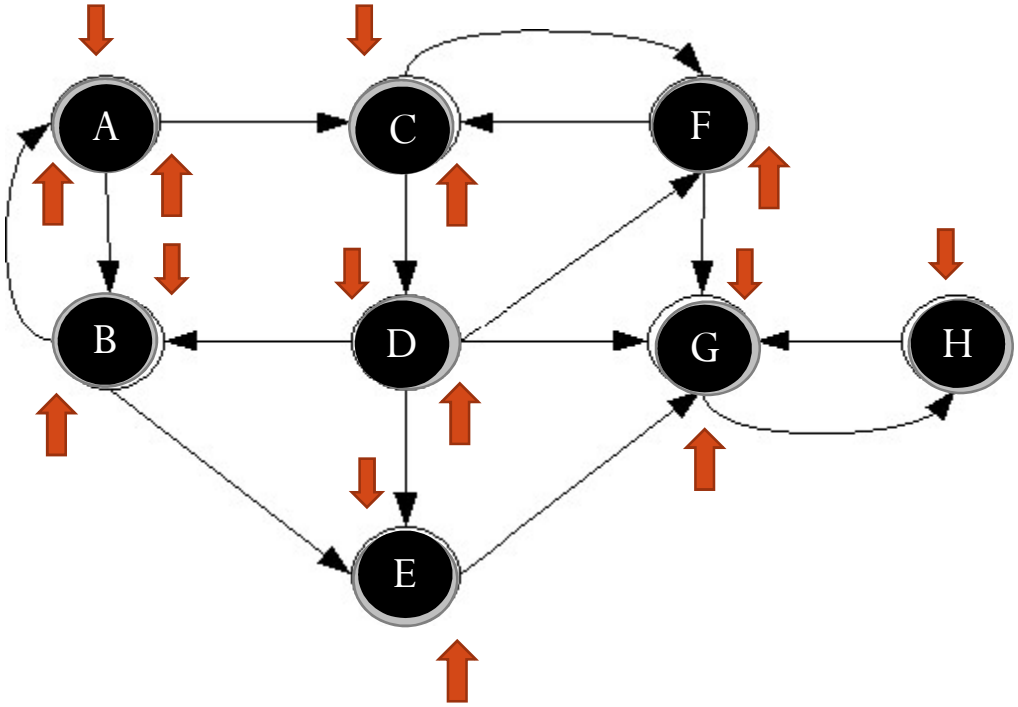
2.2. Parcours en profondeur – Utilisation d'une pile

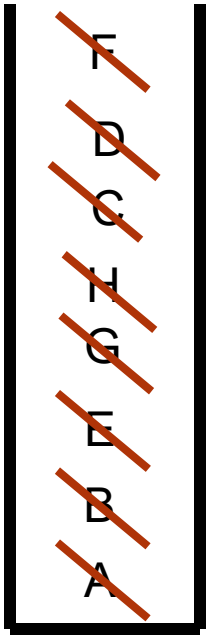
- **Opérations sur une PILE (LIFO : Last In First Out)**
  - CRÉER (pile) : Créer une pile vide
  - ESTVIDE (pile) : Renvoie vrai si la pile est vide
  - EMPILER(pile, element) : Ajouter un élément au début de la pile
  - DEPILER(pile) : Renvoie l'élément situé en début de la pile (et le retire de la pile)
  - LIRE(pile) : Renvoie l'élément situé en début de la pile (sans modifier la pile)

- **Dans le parcours**
  - Pour gérer les sommets dans l'état GRIS
  - Permet d'explorer en priorité les voisins des sommets parcourus en dernier

2.2. Parcours en profondeur – Déroulement algorithme

- **Pile : pour la liste des sommets en traitement**
- **Exemple :** à partir de A

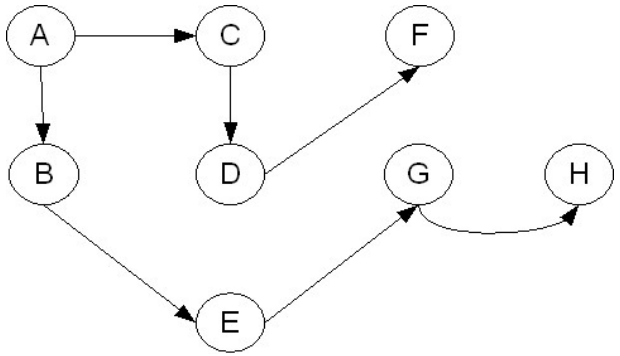




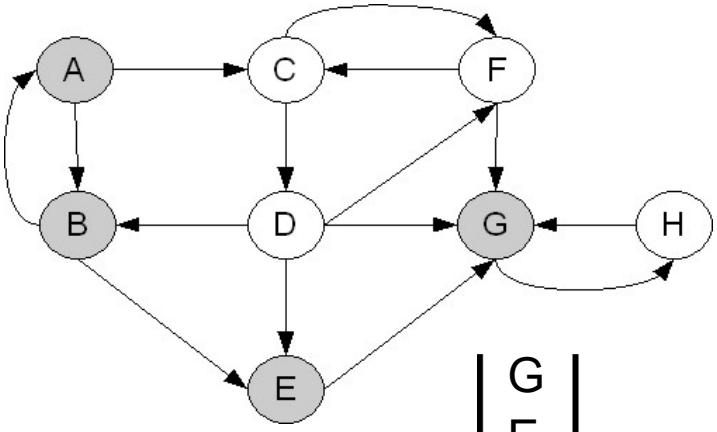
- Ordre de marquage visité : H G E B F D C A

2.2. Parcours en profondeur – Propriété de l'algorithme

- **Exploration réalisée lors du parcours en profondeur**

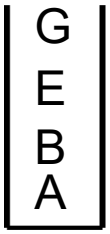


  - Arborescence du parcours



- **A chaque étape**
  - Un sommet courant (haut pile)
  - Il est accessible par tous les

autre sommets en traitement (reste pile)



2.2. Parcours en profondeur – Algorithme (1)

- **Algorithme itératif DFS(s, ETAT)**

```
DATA : Graphe G(X, A), Sommet s de départ du parcours
RESULT : liste de sommets atteints depuis s
INIT : Pour tout sommet x, ETAT(x) ← BLANC

ETAT(s) ← GRIS;   {TRAITER(s)}
CRÉER(pile); EMPILER(pile, s)
while not ESTVIDE(pile) do
  x ← LIRE(pile)
  if il existe y voisin(x) tque ETAT(y) == BLANC then
       ETAT(y) ← GRIS; {TRAITER(x)} EMPILER(pile, y);
  else
       DEPILER(pile); ETAT(x) ← NOIR
  endif
end while
```

2.2. Parcours en profondeur – Algorithme (2)

- **Algorithme récursif DFS(s, ETAT)**

```
DATA : Graphe G(X, A), Sommet s de départ du parcours
RESULT : liste de sommets atteints depuis s
INIT : Pour tout sommet x, ETAT(x) ← BLANC

ETAT(s) ← GRIS;   TRAITER(s)
for tous les voisins x de s do
   if ETAT(x) == BLANC then
       DFS(x, ETAT)
   endif
endFor
ETAT(s) ← NOIR
```

2.2. Parcours en profondeur – Propriété

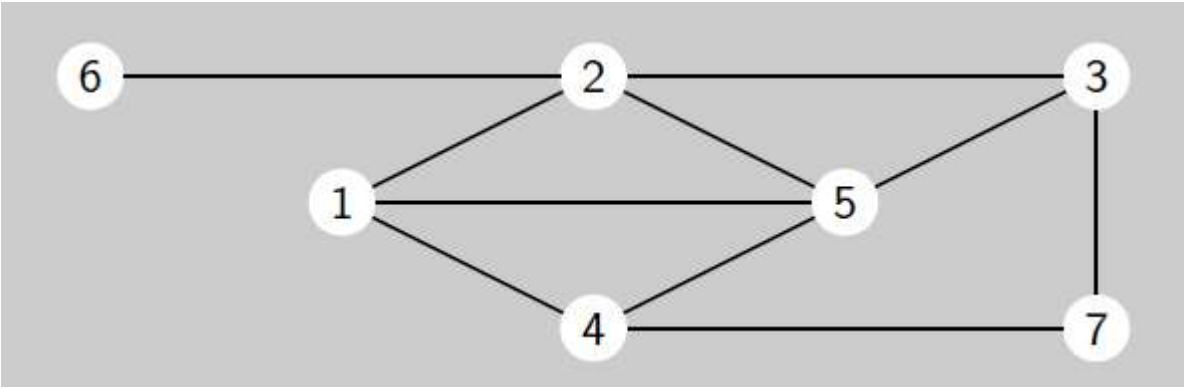
- **Propriétés de DFS(s, Etat)**
  - Visite tous les sommets accessibles depuis s via des sommets inexplorés (les autres sommets ne changent pas d'état)
  - A tout moment, les sommets en traitement représentent un chemin (une chaine) depuis l'origine du parcours jusqu'au sommet courant

2.2. Parcours en profondeur d'abord – Complexité

- **Complexité**
  - Initialisation : $O(n)$
  - Itération : $O(m)$
    - Chaque arc/arête n'est traité qu'une seule fois
    - Opérations sur la pile : temps constant
  - $O(n+m)$
    - soit $O(m)$ car $m > n$

2.2. Parcours en profondeur – Application

- **Déroulement l'algorithme de parcours en profondeur sur le graphe ci-dessous (partir du sommet 1)**
  - Fournir le contenu de la pile



  - Donner l'ordre dans lequel les sommets passent dans l'état `NOIR`

Plan

1. **Introduction**
   - Généralités
   - Quelques définitions
   - Représentations informatiques
2. **Parcours de Graphe**
   - Principe du parcours
   - Parcours en profondeur
   - Parcours en largeur
   - Premières applications d'un algorithme de parcours
     - Connexité – Forte connexité
     - Divers
3. **Optimisation et Graphes**
   - Plus courts chemins
   - Problèmes de flots

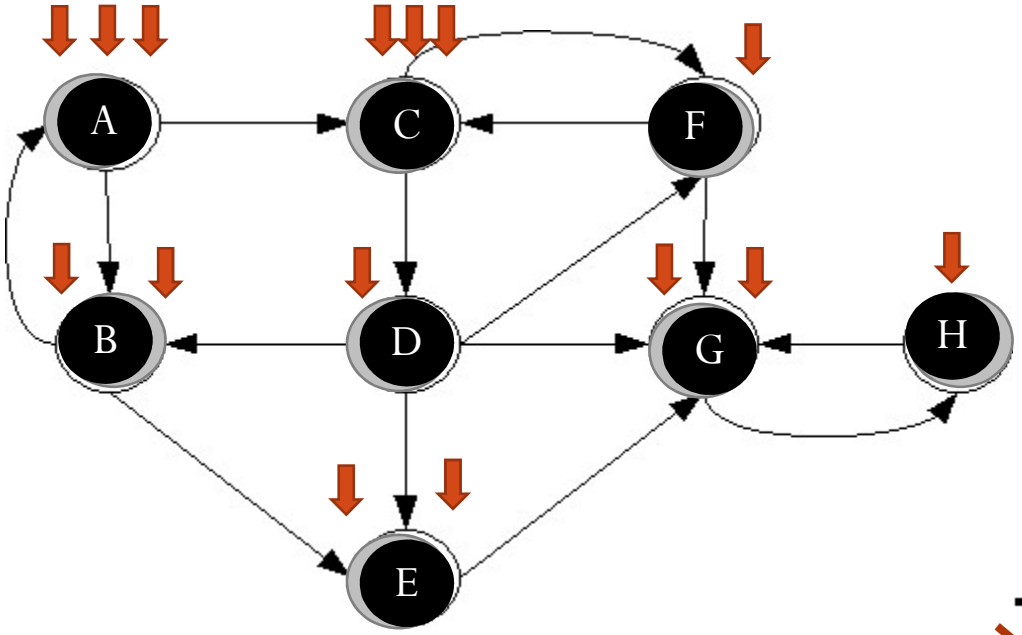
2.3. Parcours en largeur– Utilisation d'une file

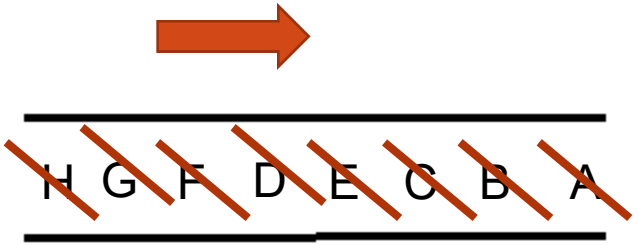
- **Opérations sur une FILE (FIFO : First In First Out)**
  - CRÉER (file) : Créer une file vide
  - ESTVIDE (file) : Renvoie vrai si la file est vide
  - ENFILER(file, element) : Ajouter un élément au fin de la file
  - DEFILER(file) : Renvoie l'élément situé en début de la file (et le retire de la file)
  - LIRE(file) : Renvoie l'élément situé en début de la file (sans modifier la file)

- **Dans le parcours**
  - Pour gérer les sommets dans l'état TRAITEMENT
  - Permet d'explorer en priorité les voisins des sommets parcourus en dernier

2.3. Parcours en largeur – Déroulement algorithme

- **File : pour la liste des sommets en traitement**
- **Exemple (à partir de A)**

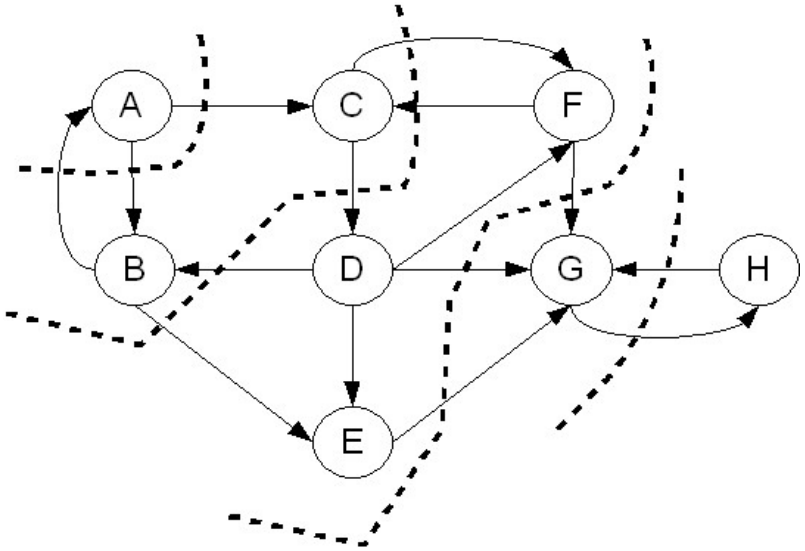


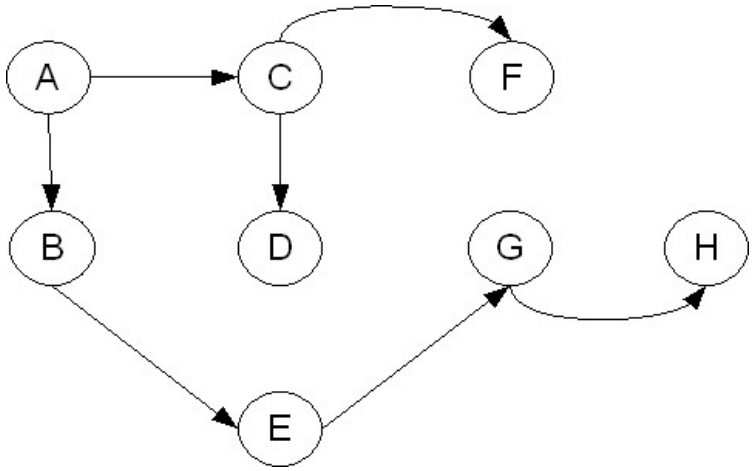


- Ordre de marquage visité : A B C E D F G H

2.3. Parcours en largeur – Exploration effectuée

- **Exploration réalisée lors du parcours en largeur**





  - Arborescence du parcours

2.3. Parcours en largeur – Algorithme (1)

- **Algorithme itératif BFS(s, ETAT)**

```
DATA : Graphe G(X, A), Sommet s de départ du parcours
RESULT : liste de sommets atteints depuis s
INIT : Pour tout sommet x, ETAT(x) ← BLANC

ETAT(s) ← GRIS;    // TRAITER(s)//
CRÉER(file); ENFILER(file, s)
while not ESTVIDE(file) do
   x ← LIRE(file)
   for tous y voisin(x) do
       if ETAT(y) == BLANC then
            ENFILER(file, y); ETAT(y) ← GRIS;   // TRAITER(x)//
        endif
   endfor
   x ← DEFILER(file); ETAT(x) ← NOIR;   { TRAITER(x);}
endwhile
```

2.3. Parcours en largeur – Algorithme (2)

- **Algorithme itératif BFS(s, ETAT)**
  - Remarque:
    - On peut retirer directement x de la file et le marquer comme NOIR

```
ETAT(s) ← GRIS; {TRAITER(x)}
CRÉER(file); ENFILER(file, s)
while not ESTVIDE(file) do
    x ← DEFILER(file); ETAT(x) ← NOIR;  {TRAITER(x)}
    for tous y voisin(x) do
        if ETAT(y) == BLANC then
            ENFILER(file, y); ETAT(y) ← GRIS; {TRAITER(x)}
        endif
    endfor
endwhile
```

2.3. Parcours en largeur – Algorithme (2)

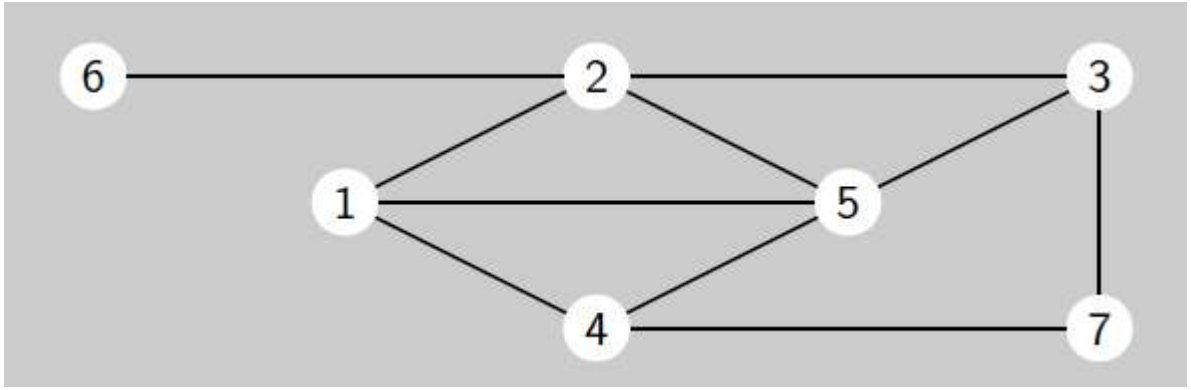
- **Algorithme itératif** **BFS(s, ETAT)**
  - Remarque:
    - Pas besoin des 3 états (un booléen suffit)
      - Quand un sommet est entré dans la file : il ne peut plus y re-entrer
      - Quand un sommet sort de la file : son exploration est terminée

2.3. Parcours en largeur – Complexité

- **Complexité**
  - Initialisation : $O(n)$
  - Itération : $O(m)$
    - Chaque arc/arête n'est traité qu'une seule fois
    - Opérations sur la file : temps constant
  - $O(n+m)$
    - soit $O(m)$ car $m > n$

2.3. Parcours en largeur – Application

- **Déroulement l'algorithme de parcours en largeur sur le graphe ci-dessous (partir du sommet 1)**
  - Fournir le contenu du tableau ETAT
  - Donner également le contenu de la file



  - Donner l'ordre dans lequel les sommets passent dans l'état `TRAITEMENT`
  - Donner l'ordre dans lequel les sommets passent dans l'état `VISITE`

Plan

1. **Introduction**
   - Généralités
   - Quelques définitions
   - Représentations informatiques
2. **Parcours de Graphe**
   - Principe du parcours
   - Parcours en profondeur
   - Parcours en largeur
   - Premières applications d'un algorithme de parcours
     - Connexité – Forte connexité
     - Divers
3. **Optimisation et Graphes**
   - Plus courts chemins
   - Problèmes de flots

2.4. Application du parcours – Premiers exemples

- Nombre d'arcs/arêtes minimal entre 1 sommet et tous les autres
- Connexité / Forte Connexité
- Détection de cycles / circuits
- Ordre topologique sur les sommets (numérotation des sommets)
- Exploration d'un graphe (ex. sortir d'un labyrinthe)

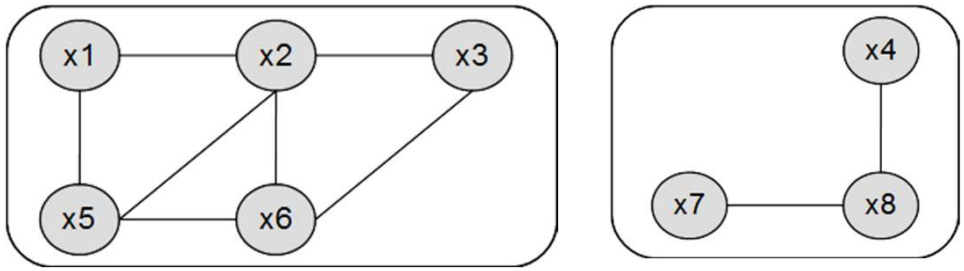
- Dans la suite
  - Focus sur quelques applications

2.4. Application du parcours – Connexité

- **Soit un graphe non orienté**
- **Définitions (rappel)**
  - Deux sommets x et y sont **connexes** ssi il existe une chaine entre x et y
  - **Composante connexe (CC)** :
    - ensemble des sommets connexes entre eux
  - **Graphe connexe** :
    - ne possède qu'une seule composante connexe
    - Il existe une chaine entre toute paire de sommets
  - **Graphe non connexe** :
    - est composé de plusieurs CC maximales (=sous graphes de plus grande taille)
  - **Remarque** : un sommet isolé (degré nul) = une CC maximale

2.4. Application du parcours – Connexité

- Soit un graphe **non orienté** $G(X, A)$
  - Le graphe G est **connexe** si $\forall(x_i, x_j) \in A$ il existe une chaine entre x_i et x_j
  - **Composante connexe (CC)** de G : sous graphe **connexe et maximal**



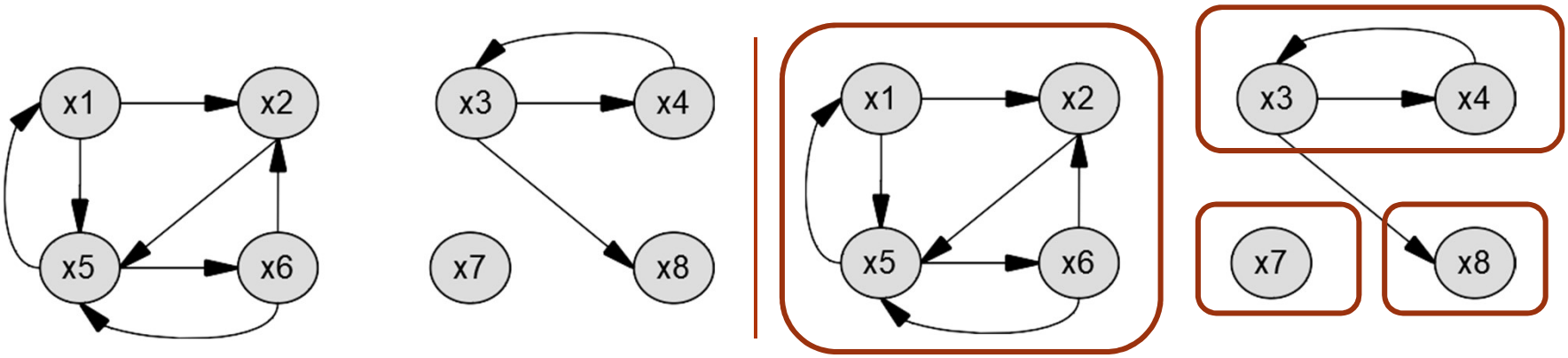
  - Application itérative d'algorithmes de parcours
  - A chaque parcours :
    - Ensemble de sommets d'une composante connexe

2.4. Application du parcours – Connexité : Algorithme

- **Principe algorithme :**
  - Structure pour mémoriser le numéro de CC de chaque sommet
    - Tableau (initialisé à 0)
  - Compteur du nombre de CC
  - Parcourir le graphe à partir d'un sommet donné x pour chercher les sommets accessibles depuis x
    - Tous les sommets noirs du parcours ➔ même numéro de CC
  - Recommencer tant qu'il existe des sommets sans CC
  - Complexité : (celle d'un parcours)
    - Autant d'itérations que de CC
    - A chaque itération : l'ensemble des sommets accessibles
    - Visite toutes les arêtes d'un graphe : $O(m)$

2.4. Application du parcours – Forte Connexité

- Soit un graphe **orienté** $G(X, A)$
  - Le graphe G est **fortement connexe** si
    - $\forall(x_i, x_j) \in A$ il existe un chemin entre x_i et x_j
  - **Composante fortement connexe (CFC)** de G :
    - sous graphe **fortement connexe et maximal**



  - Une CFC = un circuit

2.4. Application du parcours – Forte Connexité : Algorithme

- **Principe algorithme**
  - **Parcourir le graphe des successeurs** à partir d'un sommet donné x pour chercher les sommets **successeurs accessibles** depuis x
  - **Parcourir le graphe des prédécesseurs** à partir d'un sommet donné x pour chercher les sommets **prédécesseurs accessibles** depuis x
  - Les sommets accessibles par les 2 parcours sont dans la CFC de x
  - Recommencer tant qu'il existe des sommets sans CFC

- Adaptation du parcours
  - Mémoriser les sommets visités lors du parcours (marquage booléen par exemple)
  - Attribuer un numéro de CFC aux sommets

2.4. Application du parcours – Forte Connexité : Algorithme

- **Algorithme**
  - Initialisation

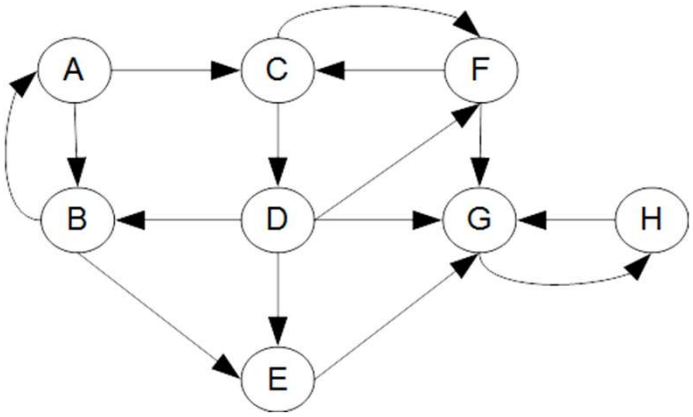
```
TabCFC(x) ← 0, pour tous les sommets x; NumCFC ← 0
Construire le graphe inverse
```

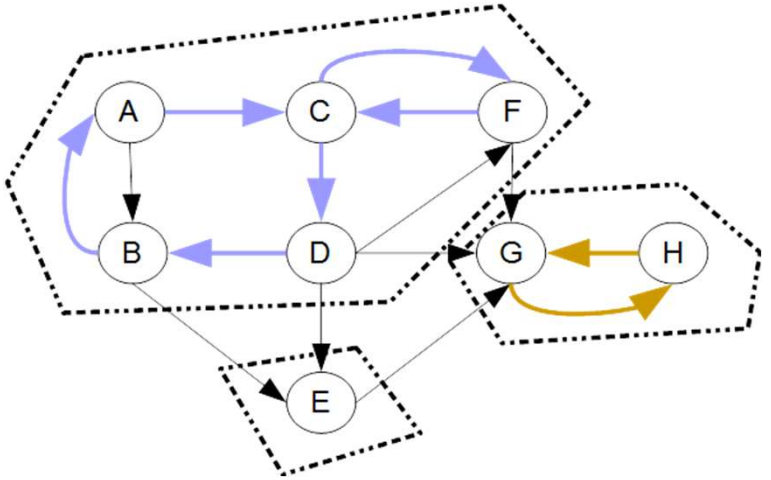
  - Itérations

```
for tous les sommets s
  if TabCFC(s) = 0 then
    NumCFC ← NumCFC + 1
    MarqueS(x) ← false, MarqueP(x) ← false, pour tout sommet x
    Parcourir-Gsucc(s, MarqueS)
    Parcourir-Gpred(s, MarqueP)
    for tous les sommets x
      if MarqueS(x)=MarqueP(x)=true then TabCFC(x) ← NumCFC
    end for
  end if
end for
```

2.4. Application du parcours – Forte Connexité : Complexité

- **Complexité**
  - Au pire n parcours de Graphes : $O(n \times m)$
  - Exemple



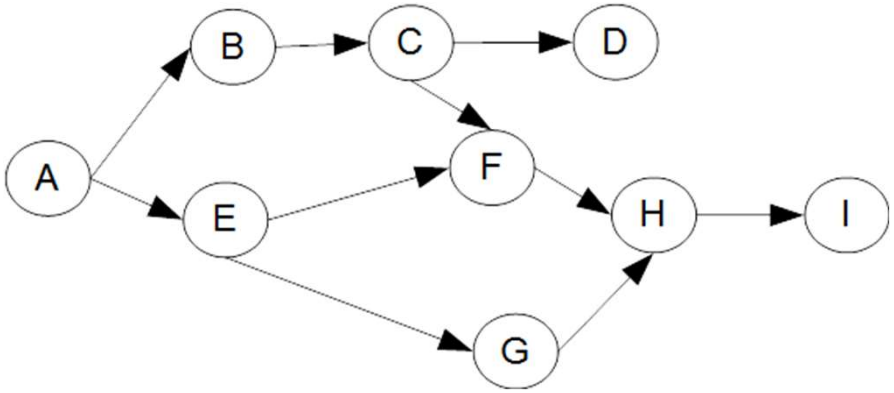


  - Meilleur algorithme connu
    - 1 seul parcours : $O(m)$
    - Algorithme de Tarjan (1972)

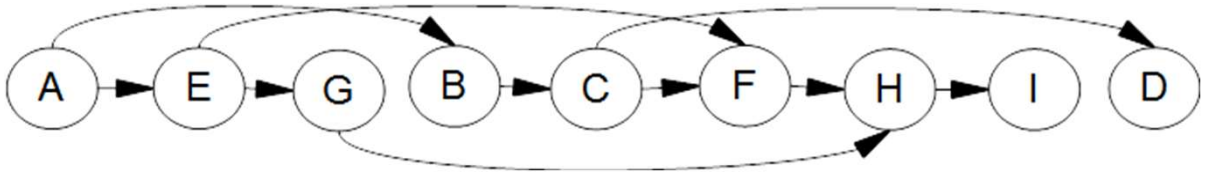
2.4. Application du parcours – Ordre topologique

- Graphe orienté acyclique

Directed Acyclic Graphs (DAG)



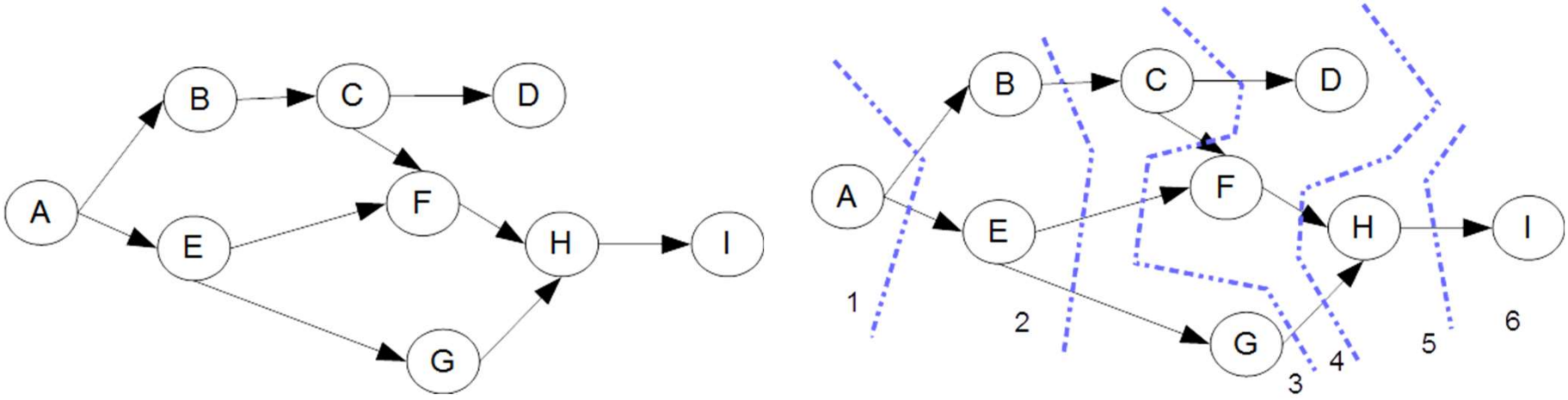
- Ordre topologique dans un DAG $G = (X, A)$
  - Numérotation des sommets telle que $n(x_i) < n(x_j)$ si $(x_i, x_j) \in A$



  - Application algorithme de parcours en profondeur (voir TD)
  - **Propriété** : un graphe acyclique admet un ordre topologique

2.4. Application du parcours – Décomposition en niveaux

- Graphe orienté acyclique

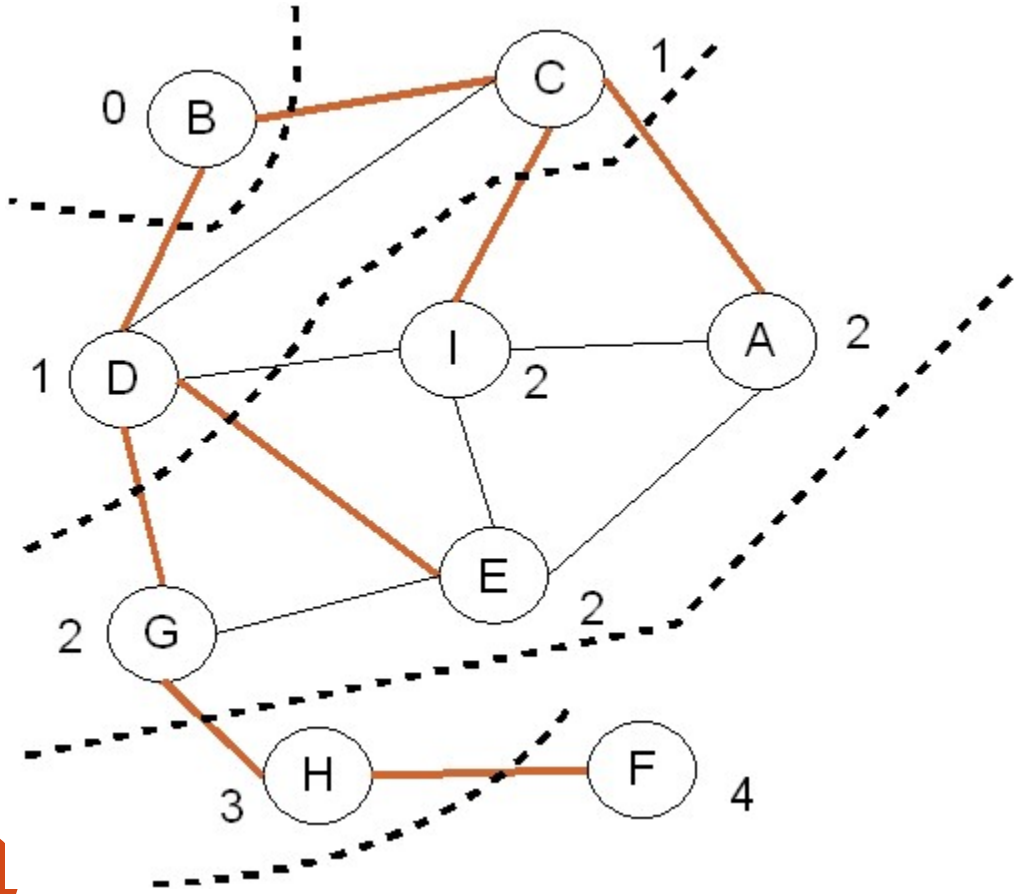


- **Niveau** : ensemble de sommets dont tous les prédécesseurs sont traités
  - Ordre partiel entre les sommets (plusieurs sommets peuvent avoir le même niveau)
  - On peut en déduire un ordre topologique

  - Variante algorithme parcours (voir TD)

2.4. Application du parcours – Plus petit nombre d'arcs/arêtes

- Minimiser le nombre d'étapes pour relier deux sommets
  - Distance = nombre d'arcs / arêtes (cout arcs/arêtes = 1)



- **Application parcours en largeur**

2.4. Application du parcours – Fermeture transitive

- **Fermeture Transitive**
  - Pour un sommet x : ensemble de tous les sommets accessibles à partir de x
  - Pour un graphe : fermeture transitive de tous les sommets

    tous les chemins d'un graphe
  - La longueur maximale des chemins est $n - 1$ (pour n sommets)
  - Utiliser un algorithme de parcours pour chaque sommet (graphe peu dense)

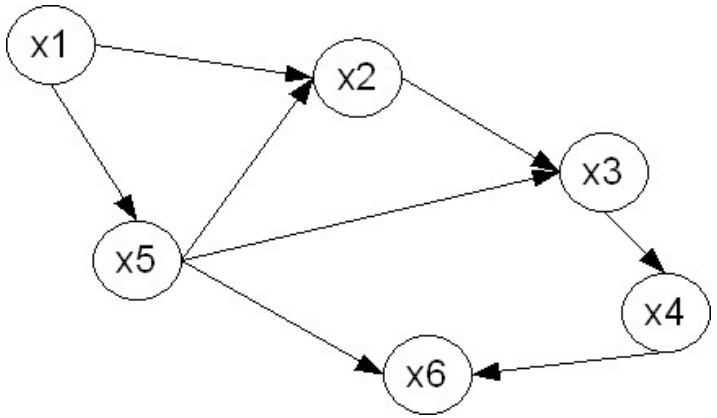
```
For tous sommets x de 1 à n
    Fermeture(x) ← Parcours(x)
```

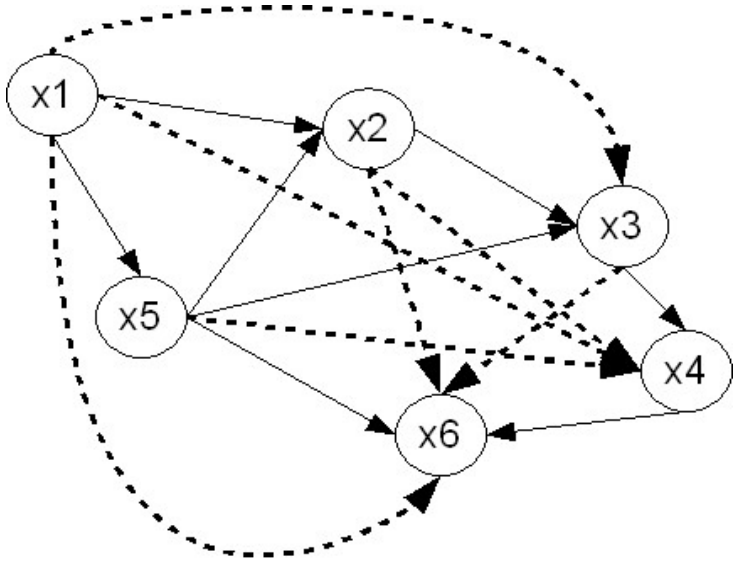
    - A chaque parcours pour récupérer la liste des sommets marqués
    - Complexité : $O(n \times m)$

2.4. Application du parcours – Fermeture transitive

- **Application :**

  - Fermeture(x1) = {x2, x3, x4, x5, x6}
  - Fermeture(x2) = {x3, x4, x6}
  - Fermeture(x3) = {x4, x6}
  - Fermeture(x4) = {x6}
  - Fermeture(x5) = {x2, x3, x4, x6}
  - Fermeture(x6) = {}





2.4. Application du parcours – Fermeture transitive

- Algorithme de Warshall basé sur la matrice d'adjacence
  - Calculs successifs de matrice M_k
    - M_0 matrice initiale, et M_n matrice recherchée
    - Les matrices M_k vérifient
      - $M_k(i,j) = 1$ s'il existe un chemin de x_i i à x_j passant uniquement par des sommets inférieurs ou égaux à x_k,
      - $M_k(i,j) = 0$ sinon

```
pour k de 1 à n
  pour i de 1 à n
    pour j de 1 à n
      Mk[i,j] = Mk-1[i,j] or (Mk-1[i,k] and Mk-1[k,j])
```

  - Complexité : $O(n^3)$

Plan

1. **Introduction**
2. **Parcours de Graphe**
3. **Optimisation et Graphes**
   - Plus courts chemins
     - Généralités
     - De 1 vers n - Valuations positives - Algorithme de Dijkstra
     - De 1 vers n - Valuations quelconques - Algorithme de Bellman-Ford
     - De 1 vers n - Graphe orienté sans circuit – Valuations quelconques – Algorithme de Bellman
     - Plus courts chemins de n vers n
   - Problèmes de flots

3.1. Plus courts chemins – Généralités

- **Quel est le problème posé ?**
  - Graphes valués :
    - Distance, Temps, ….
  - Cout d'un chemin/chaine
    - Somme des valuations des arcs/arêtes qui le composent

  - Chercher un chemin (chaine) de cout minimal entre deux sommets donnés sur un graphe valué → de 1 vers 1
  - Chercher les chemins (chaines) de cout minimal entre 1 sommet donné et tous les autres sur un graphe valué → de 1 vers n

3.1. Plus courts chemins – Généralités

- **Variantes**
  - PCC entre toute paire de sommets → de n vers n
  - PCC en nombre d'arcs/arêtes (distance)
    - Méthode : Adaptation du parcours en largeur

- **Intérêt**
  - Problème facile (complexité)
  - Nombreuses applications
  - Sous-Problème de problèmes difficiles

3.1. Plus courts chemins – Généralités

- **Condition d'optimalité**
  - Les sous-chemins de plus courts chemins sont des plus courts chemins
    - Soit P=(i, ….., j) un plus court chemin de i vers j
    - **Montrons que** si Q=(a, ….b) un sous chemin tel que $i \leq a \leq b \leq j$ alors Q=(a, …b) est un plus court chemin de a vers b
    - **Démonstration**
      - Décomposition de P = (i, ….., a, ……, b, ….., j)
      - Cout de P : W(P) = W(i, …., a) + W(a, …., b) + W(b, …,j)
      - Supposons qu'il existe R chemin de a à b tel que W(R) < W(Q)
      - Le cout du nouveau chemin de i à j est donc : W(T) = W(i, …, a) + W(R) + W(b, …, j), et donc W(T) < W(P) car W(R) < W(Q)
      - Impossible car P est un plus court chemin de i vers j par hypothèse

3.1. Plus courts chemins – Généralités

- **Conséquences**
  - Les plus courts chemin d'un sommet s (origine) vers tous les autres forment une arborescence appelé « arborescence des plus courts chemins partant de s »
  - Si d_k = valeur du PCC de s vers k alors le chemin P_{sk} est un plus court chemin ssi $d_j = d_i + w_{ij}$ pour tout arc $(i,j) \in P_{sk}$

- **Principe des algorithmes : PCC depuis sommet s**
  - Chercher l'arborescence des plus courts chemin partant de s et vérifiant la condition d'optimalité
    - Définir des étiquettes (labels) pour chaque sommet : d_i
      - Estimation du PCC de s vers i
    - Mettre à jour itérativement ces étiquettes (principe de relaxation)

$$d_i \leftarrow \min(d_k + w_{ki},\ \forall k \in \delta^-(i));\ \forall i \in X; i \neq s$$

Plan

1. **Introduction**
2. **Parcours de Graphe**
3. **Optimisation et Graphes**
   - Plus courts chemins
     - Généralités
     - De 1 vers n - Valuations positives - Algorithme de Dijkstra
     - De 1 vers n - Valuations quelconques - Algorithme de Bellman-Ford
     - De 1 vers n - Graphe orienté sans circuit – Valuations quelconques – Algorithme de Bellman
     - Plus courts chemins de n vers n
   - Problèmes de flots

3.2. Plus courts chemins – Algorithme de Dijkstra

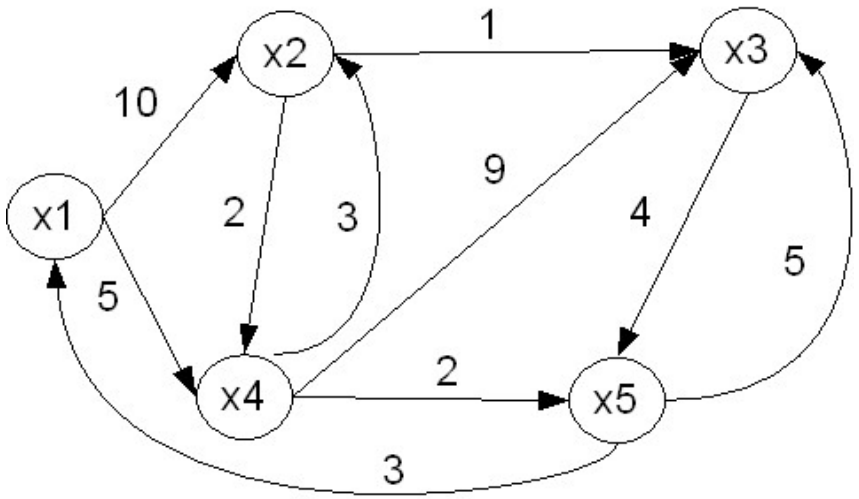
- **PCC de 1 vers n et de 1 vers 1**
- **Conditions d'application**
  - Graphes orientés et Graphes non orientés
  - Valuations positives
- **Condition d'existence d'un chemin de cout minimal**
  - Le graphe ne comporte pas de circuit/cycle de longueur négative
    - Vérifié avec l'hypothèse : valuations positives
- **Recherche de chemins/chaines élémentaires**
  - Chemins/Chaines passant au maximum une fois par les sommets
    - Tous les sommets du (des) chemin(s)/chaine(s) solution sont deux à deux distincts

3.2. Plus courts chemins – Algorithme de Dijkstra : Principe

- **But :**
  - **Plus court chemin d'une origine vers tous les autres sommets**
- **Principe**
  - Label :
    - Cout du chemin à l'origine s = 0
    - Cout du chemin aux autres sommets ← ∞
  - Adaptation d'un parcours en largeur
    - A chaque étape :
      - **Sélectionner le sommet de cout le plus faible** et marquer ce sommet comme visité (son cout ne changera plus)
      - **Actualiser les couts des sommets adjacents** non marqués
    - Arrêt
      - Tous les sommets sont marqués

3.2. Plus courts chemins – Algorithme de Dijkstra : Exemple

- **Application : plus court chemin à partir de x1**



x1	x2	x3	x4	x5
0	∞	∞	∞	∞
0	10, x1	∞	5, x1	∞
•	8, x4	14, x4	**5, x1**	7, x4
•	8, x4	12,x5	•	**7,x4**
•	**8, x4**	9, x2	•	•
•	•	**9,x2**	•	•

- Pour chaque sommet, mémoriser :
  - Son meilleur cout courant,
  - le sommet précédent pour ce meilleur cout
  - un marquage

3.2. Plus courts chemins – Algorithme de Dijkstra

- **Déroulement de l'algorithme**
  - Associer un label (étiquette) à chaque sommet
    - Cout Cost(i)
    - Précédent Father(i)
    - Marquage Mark(i)
  - Structure pour obtenir le sommet de plus petit cout
    - File de priorité = Tas **Priority queue , binary heap** RAPPEL
  - Algorithme
    - Basé sur un parcours en largeur
    - Remplacer la file par un tas (file de priorité)

3.2. Plus courts chemins – Algorithme de Dijkstra

- **Initialisation**

```
For tous les sommets i  de 1 à n
  Mark(i) ← Faux
  Cost(i) ← +∞
  Father(i) ← 0    // sommet inexistant
end for

Cost(s) ← 0
Insert(s, Tas)
```

  - A l'initialisation :
    - la file de priorité ne contient que le sommet origine

3.2. Plus courts chemins – Algorithme de Dijkstra

- **Itérations**

```
While il existe des sommets non marqués
  x ← ExtractMin(Tas)
  Mark(x) ← true
  For tous les y successeurs de x
    If not Mark(y) then
      Cost(y) ← Min(Cost(y), Cost(x)+W(x,y))
      If Cost(y) a été mis à jour then
         Placer(y, Tas)
         Father(y) ← x
      end if
    end if
  end for
end while
```

3.2. Plus courts chemins – Algorithme de Dijkstra

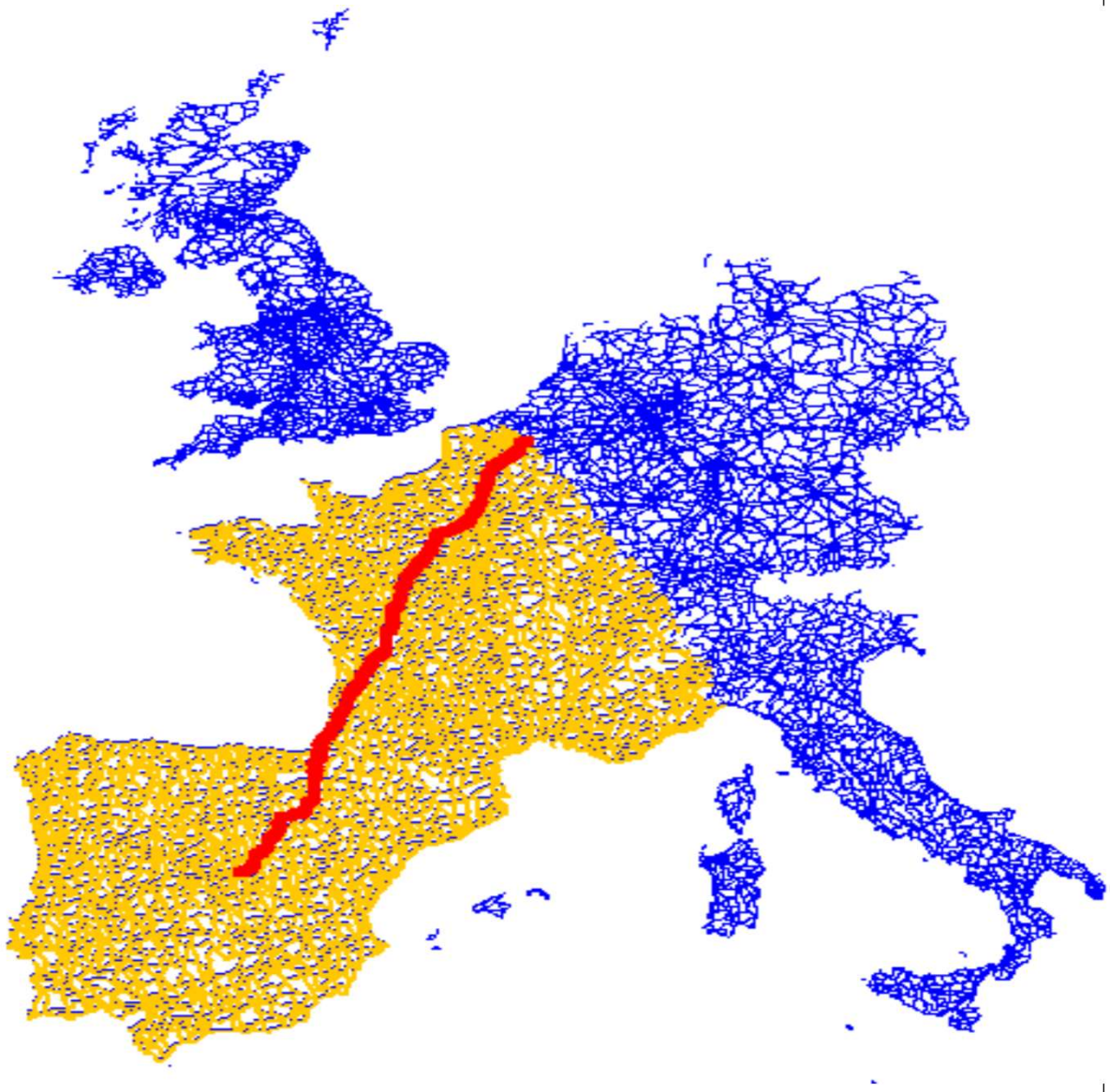
- **Itérations**

```
While il existe des sommets non marqués
  x ← ExtractMin(Tas)
  Mark(x) ← true
  For tous les y successeurs de x
    If not Mark(y) then
      
    end if
  end for
end while
```

```
if Cost(y) > Cost(x)+W(x, y) then
    Cost(y) ← Cost(x) + W(x, y)
    if Exist(y, Tas) then
          Update(y, Tas)
    else
          Insert(y, Tas)
    end if
end if
```

Déroulement Dijkstra

Madrid → Bruxelle



3.2. Plus courts chemins – Algorithme de Dijkstra : Complexité

- **Complexité**
  - Initialisation : O(n)
  - Itérations
    - N passages dans la boucle while : chaque sommet ne peut être marqué qu'une seule fois
    - A chaque passage dans la boucle
      - Extraction du sommet de plus petit cout dans le tas : eq
      - Mise à jour des successeurs et insertion dans le tas : $d^+(x)$.iq
    - Total : O(n eq + m iq)
    - Avec un tas implémenté par un arbre binaire
      - eq=iq = O(log nb éléments)
      - Au maximum : il y a n éléments dans le tas
  - D'où la complexité : O(n + (m+n).log n) = O((m+n).log n)

3.2. Plus courts chemins – Algorithme de Dijkstra : Compréhesion

- **Questions**
  - Comment mettre en œuvre la condition du while

```
while il existe des sommets non marqués
```

  - Pourquoi tester si un sommet est déjà dans le tas avant de réaliser l'insertion ?
  - A quoi correspond l'opération « update » sur le tas ?
  - Lien avec l'algorithme de parcours en largeur
    - Ici 2 états : marqué / non marqué
    - En fait 3 cas : non marqué et cout infini, non marqué et cout, marqué

3.2. Plus courts chemins – Algorithme de Dijkstra : Adaptation de 1 vers 1

- **Résultat de l'algorithme**
  - Les plus courts chemins d'une origine vers tous les autres sommets
    - **Question** : comment obtenir le plus court chemin d'une origine vers une destination ?

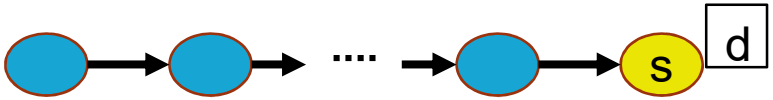
3.2. Plus courts chemins – Algorithme de Dijkstra : Correction

- **Correction de l'algorithme**
  - 2 propriétés
    - **P1 : Sommets Marqués** : le cout d'un sommet marqué est la longueur du plus court chemin de l'origine jusqu'à ce sommet
    - **P2 : Sommets Non Marqués** : le cout (non infini) d'un sommet non marqué est la longueur du plus court chemin depuis l'origine parmi ceux qui ne passent que par des sommets Marqués
    - Initialement ces propriétés sont vraies
    - Montrons qu'elles sont vraies au cours de l'algorithme

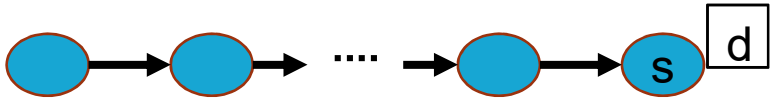
3.2. Plus courts chemins – Algorithme de Dijkstra : Correction

- **P1 . Considérons un sommet s**
  - Non marqué ➔ Marqué
    - s a un cout d, ce cout est le plus petit parmi les sommets non marqués

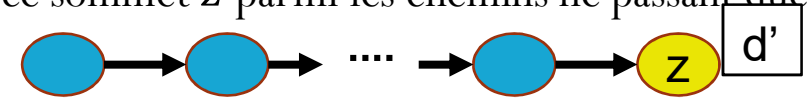
P1 : Sommets Marqués : le cout d'un sommet marqué est la longueur du plus court chemin de l'origine jusqu'à ce sommet



    - Marquer s



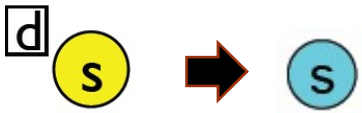
    - On veut montrer que d est le plus court chemin depuis l'origine
      - Le chemin de l'origine vers s débute par des sommets marqués et se termine par un sommet z non marqué de cout d'
        - d' est la meilleure distance pour ce sommet z parmi les chemins ne passant que par des sommets marqués (P2)



        - Le chemin arrive ensuite à s
        - le cout de s est alors $d' + dist(z, s)$
      - Or $d \leq d'$ car s a été marqué avant z ➔ donc $s = z$ car les distances sont positives

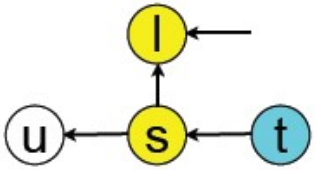
3.2. Plus courts chemins – Algorithme de Dijkstra : Correction

- **P2. Considérons un sommet s de cout non infini**
  - s va être marqué, que devient P2 pour les voisins de s ?

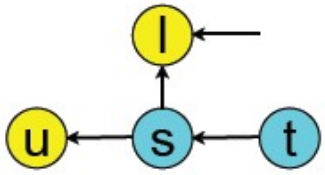


P2 : Sommets Non Marqués :
le cout (non infini) d'un sommet non marqué est la longueur du plus court chemin depuis l'origine parmi ceux qui ne passent que par des sommets Marqués

  - 2 cas :
    - u voisin de s avait un cout infini, il obtient un cout et ce cout est celui du chemin de l'origine jusqu'à s puis de s vers u : avant u ce chemin ne passe que par des sommets marqués. P2 est donc vérifiée



    - l voisin de s avait un cout non infini (qui vérifiait donc P2): un nouveau chemin de l'origine vers l apparait lorsque s est marqué. Le cout de l est actualisé (minimum entre l'ancien cout et le cout du nouveau chemin). P2 est donc vérifiée.

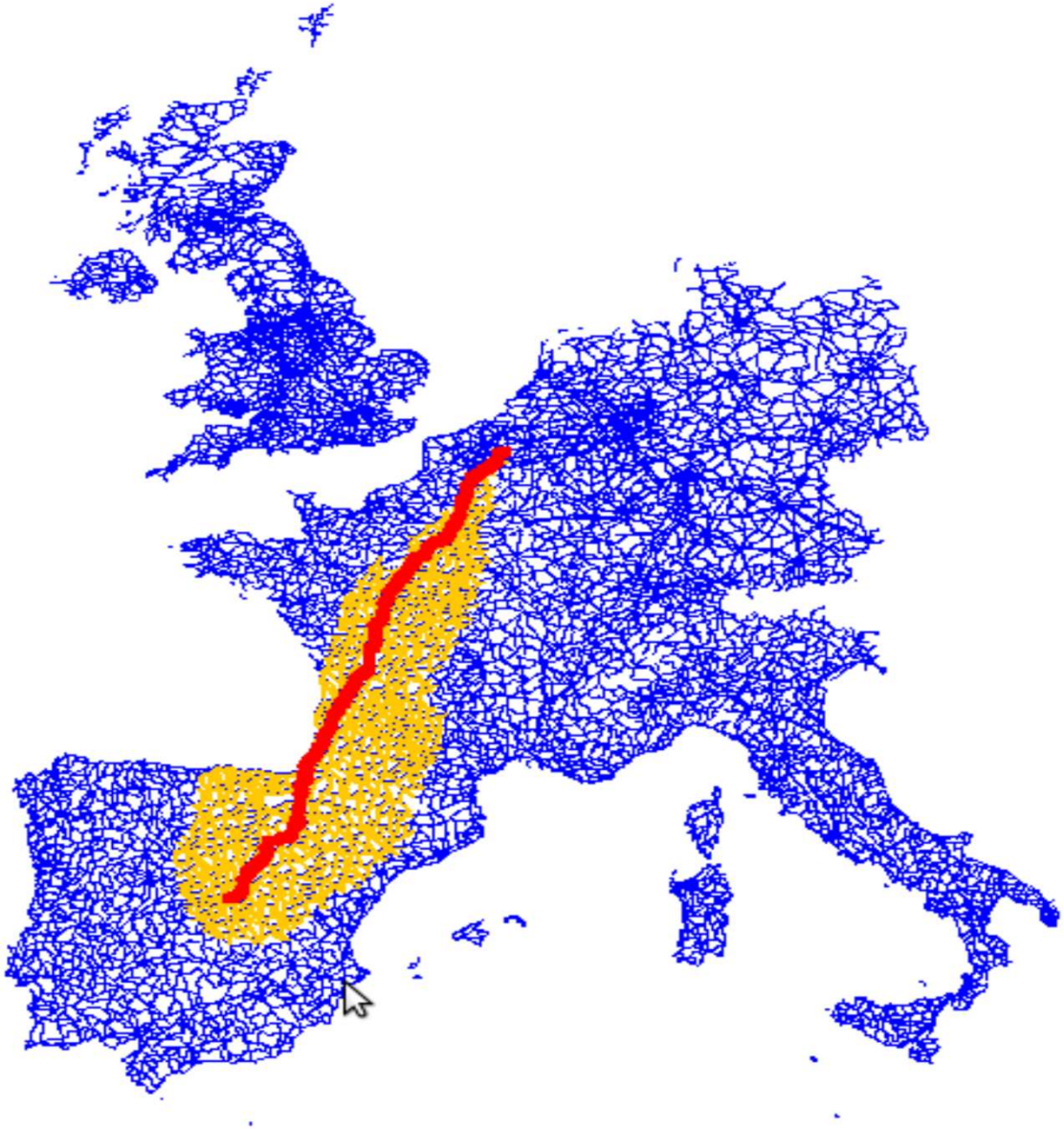


3.2. Plus courts chemins – Algorithme de Dijkstra - Adaptations

- **Variantes**
  - L'algorithme de Dijkstra permet de déterminer :
    - les plus courts chemins d'un sommet vers tous les autres
    - Le plus courts chemin d'une origine vers une destination
  - Pour le plus court chemin origine-destination : peut-on avoir un meilleur algorithme ?
    - Dijkstra guidé vers la destination (voir le BE Graphes)
      - Utilisation d'une estimation (optimiste) du cout restant à la destination
    - Dijkstra bidirectionnel (non traitée en TP)
      - Algorithmes en Forward depuis l'origine et en Backward depuis la destination
      - Condition d'arrêt

Déroulement Dijkstra guidé

Madrid → Bruxelle



3.2. Plus courts chemins – Algorithme de Dijkstra : Variantes

- **Autres techniques d'accélération**
  - Pour des grands graphes
  - Méthodes de pré-calcul
    - Très nombreuses
    - Principe
      - Calculer certains plus courts chemins : mais on ne peut pas stocker tous les résultats
        - Faire des pré-calculs pertinents
        - Faire des pré-calculs utilisables pour accélérer la réponse à une requête

Plan

1. **Introduction**
2. **Parcours de Graphe**
3. **Optimisation et Graphes**
   - Plus courts chemins
     - Généralités
     - De 1 vers n - Valuations positives - Algorithme de Dijkstra
     - De 1 vers n - Valuations quelconques - Algorithme de Bellman-Ford
     - De 1 vers n - Graphe orienté sans circuit – Valuations quelconques – Algorithme de Bellman
     - Plus courts chemins de n vers n
   - Problèmes de flots

3.3. Plus courts chemins – Valuations quelconques

- **Et pour des valuations quelconques**
  - L'algorithme de Dijkstra n'est plus correct
- **Existence de chemins de cout minimal**
  - Pas de circuit de longueur négative
- **Principe de programmation dynamique**
  - Initialisation
    - $Cost\_0(s) = 0$
    - $Cost\_0(x) = \infty$
  - Récurrence
    - $Cost\_k(x) = MIN\ (Cost\_k\text{-}1(x), Cost\_k\text{-}1(y) + W(y, x)\ )$, pour tout y prédécesseur de x

3.3. Plus courts chemins – Valuations quelconques

- **Principe de la méthode**
  - Calcul incrémental de Cost_k(x)
  - Arrêt :
    - Les couts n'évoluent pas
    - Un nombre maximal d'itérations est atteint
    - En effet, si le graphe ne contient pas de circuit absorbant
      - Un chemin solution comportera au plus n-1 arcs (arêtes) car on ne recherche que des chemins élémentaires
      - Si le nombre d'itérations atteint n alors un circuit absorbant a été détecté

3.3. Plus courts chemins – Valuations quelconques

- **Algorithme dit Moore-Bellman-Ford**
  - Implémentation des équations de récurrence
  - Pour chaque sommet : actualisation basée sur le cout des prédécesseurs

- **Algorithme dit FIFO (Ahuja-Magnati-Orlin)**
  - Parcourir les sommets grâce à une file initialisée avec le sommet origine du calcul
  - Les successeurs d'un sommet dont les couts sont améliorés sont insérés dans la file
  - Arrêt : file vide ou détection de circuit absorbant

3.3. Plus courts chemins – Algorithme de Moore-Bellman-Ford

- **Condition d'application**
  - Graphes orientés ou non orientés
  - Valuations quelconques
  - Pas de circuit/cycle absorbant

- **Structure de données**
  - Associer un label (étiquette) à chaque sommet
    - Cout Cost(i)
    - Précédent Father(i)

3.3. Plus courts chemins – Algorithme de Moore-Bellman-Ford

- **Initialisation**

$Cost\_0(s) = 0$
$Cost\_0(x) = \infty$, pour tout $x \neq s$

```
For tous les sommets i  de 1 à n
  Cost(i) ← +∞
  Father(i) ← 0    // sommet inexistant
end for

Cost(s) ← 0
k ← 0          // pour compter les itérations
```

3.3. Plus courts chemins – Algorithme de Moore-Bellman-Ford

- **Itérations**

$Cost\_k(x) = MIN\ (Cost\_k\text{-}1(x), Cost\_k\text{-}1(y) + W(y, x)\ ),$ pour tout $y \in \delta^{-}(x)$

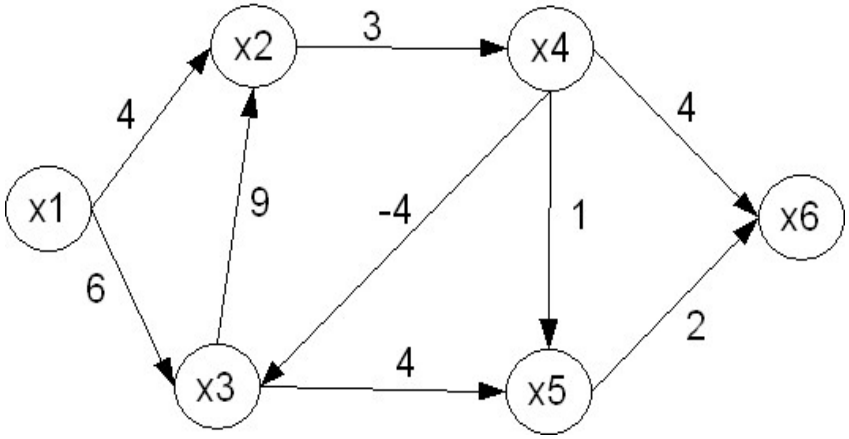
```
repeat
  continuer ← false
  k ← k+1
  For tous les sommets x (/= de s)
      for tous les prédecesseurs y de x
       Cost(x) ← Min(Cost(x), Cost(y)+W(y,x))
       if Cost(x) a été mis à jour then
         Father(x) ← y
         continuer ← true
        end if
      end for
  end for
until (k=n) or (not continuer)
```

3.3. Plus courts chemins – Algorithme de Moore-Bellman-Ford : Complexité

- **Complexité**
  - Initialisation : O(n)
  - Itérations
    - Boucle repeat: au maximum n itérations
    - A chaque itération
      - Parcourir tous les prédécesseurs de tous les sommets : O(m)
  - D'où la complexité : O(n+ n.m) = O(n.m)

3.3. Plus courts chemins – Algorithme de Moore-Bellman-Ford : Exemple

- **Exemple : PCC à partir de x1**



x1	x2	x3	x4	x5	x6
0	∞	∞	∞	∞	∞
0	4, x1	6,x1	7,x2	8,x4	10,x5
0	4, x1	3,x4	7,x2	7,x3	9, x5
0	4, x1	3,x4	7,x2	7,x3	9, x5

Arrêt : pas de mise à jour

- 1 itération :
  - Parcours (et mise à jour) de tous les sommets

3.3. Plus courts chemins – Algorithme de Moore-Bellman-Ford : Questions

- **Questions**
  - L'algorithme de Moore-Bellman-Ford détermine les plus courts chemins d'une origine vers tous les sommets du graphe
    - Peut-on adapter (comment) l'algorithme pour déterminer le plus court chemin d'une origine à une destination ?
  - L'ordre de traitement des sommets a-t-il une influence sur le nombre d'itérations de l'algorithme ?

3.3. Plus courts chemins – Variante Algo MBF: Algorithme dit FIFO (Annexe)

- **Utilisation d'une file de sommets mis à jour lors des itérations**

- **Initialisation**

```
For tous les sommets i de 1 à n
   Cost(i) ← +∞
   Father(i) ← 0 // sommet inexistant
end for
Cost(s) ← 0
Insert(s, F)
```

  - A l'initialisation :
    - la file ne contient que le sommet origine

3.3. Plus courts chemins – Variante Algo MBF: Algorithme dit FIFO (Annexe)

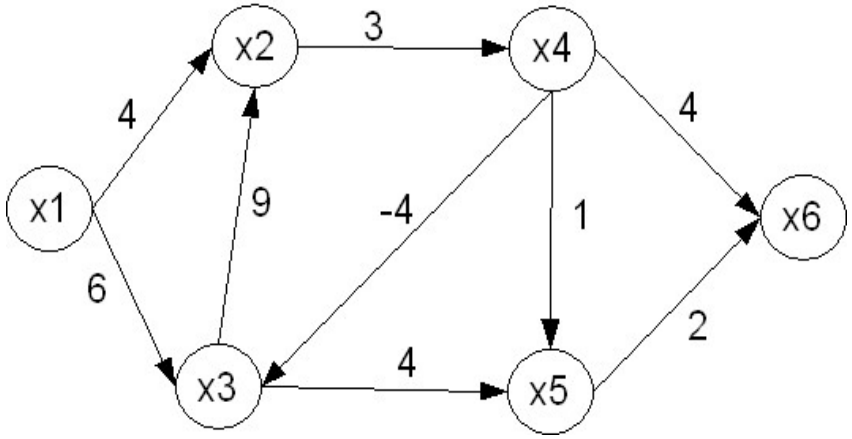
- **Itérations**

```
While not IsEmpty(F)
   x ← Extract(File)
   For tous les y successeurs de x
      Cost(y) ← Min(Cost(y), Cost(x)+W(x,y))
      If Cost(y) a été mis à jour then
          Father(y) ← x
          If not Exist(y, File) then Insert(y, F')
          end if
      end if
   end for
   F ← F'
end while
```

- Détection de circuits/cycles
  - Compter les itérations :
    - circuit/cycle si F non vide au bout de n itérations

3.3. Plus courts chemins – Variante Algo MBF: Algorithme dit FIFO (Annexe)

- **Exemple**



	x1	x2	x3	x4	x5	x6	F
(init)	0	∞	∞	∞	∞	∞	x1
x1	0	**4,x1**	**6,x1**	∞	∞	∞	x2,x3
x2,x3	0	4,x1	6,x1	**7,x4**	**10,x3**	∞	x4,x5
x4,x5	0	4,x1	**3,x4**	7,x4	**8,x4**	**11,x4** **10,x6**	x3,x5,x6
x3,x5,x6	0	4,x1	3,x4	7,x4	7,x3	9,x5	x5,x6
x5,x6	0	4,x1	3,x4	7,x4	7,x3	9,x5	/

3.3. Plus courts chemins – Variante Algo MBF: Algorithme dit FIFO (Annexe)

- **Complexité**
  - Initialisation : O(n)
  - Itérations
    - Boucle while : au maximum n itérations
    - A chaque itération
      - Extraction d'un sommet de la file : ef
      - Mise à jour des successeurs et insertion dans la file : d+(x).if
    - Total n.ef + m.if
    - Opérations sur la file
      - ef = if = O(1)
  - D'où la complexité : O(n+ n.m) = O(n.m)

Plan

1. **Introduction**
2. **Parcours de Graphe**
3. **Optimisation et Graphes**
   - Plus courts chemins
     - Généralités
     - De 1 vers n - Valuations positives - Algorithme de Dijkstra
     - De 1 vers n - Valuations quelconques - Algorithme de Bellman-Ford
     - De 1 vers n - Graphe orienté sans circuit – Valuations quelconques – Algorithme de Bellman
     - Plus courts chemins de n vers n
   - Problèmes de flots

3.4. Plus courts chemins – Valuations quelconques & Graphe sans circuit

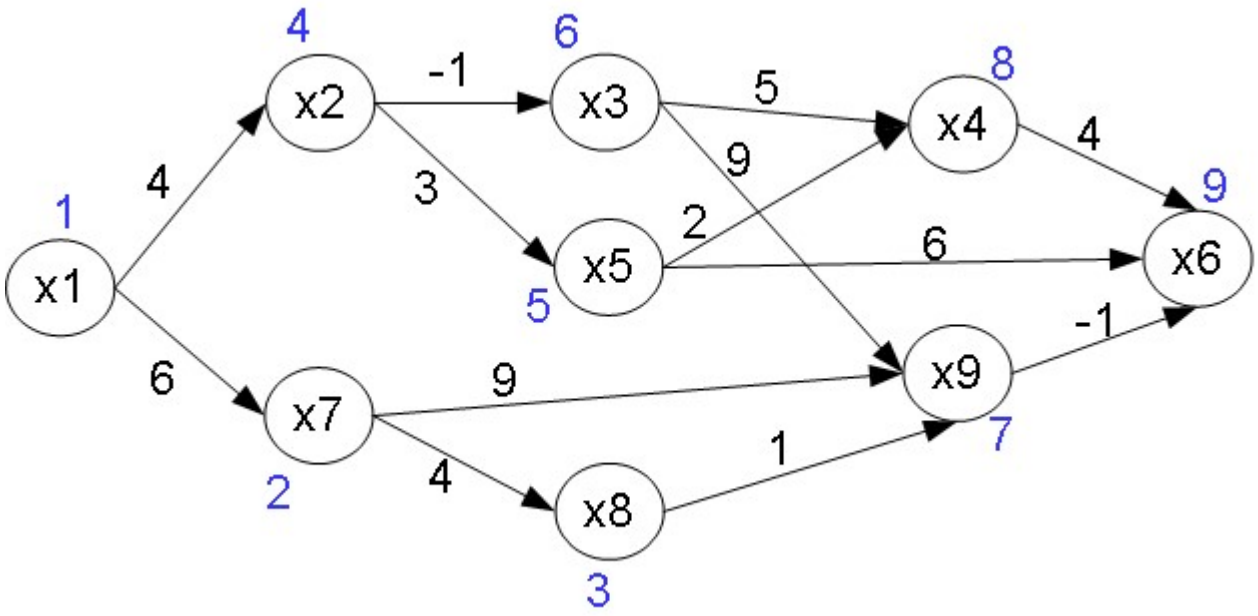
- **Hypothèse : Graphe sans circuit (valuations quelconques)**
  - On peut ordonner les sommets en fonction de leur ordre topologique (ou de leur rang)
  - Principe du calcul des plus courts chemins :
    - À partir des couts des sommets d'un numéro donné k, on peut actualiser les couts des sommets d'un numéro supérieur
    - Simple parcours en suivant l'ordre topologique
    - Algorithme de Bellman
  - Complexité : O(m+m) = O(m)

3.4. Plus courts chemins – Valuations quelconques & Graphe sans circuit

- **Exemple**

1. Calculer ordre topo

2. Effectuer le parcours dans l'ordre topologique en calculant les couts



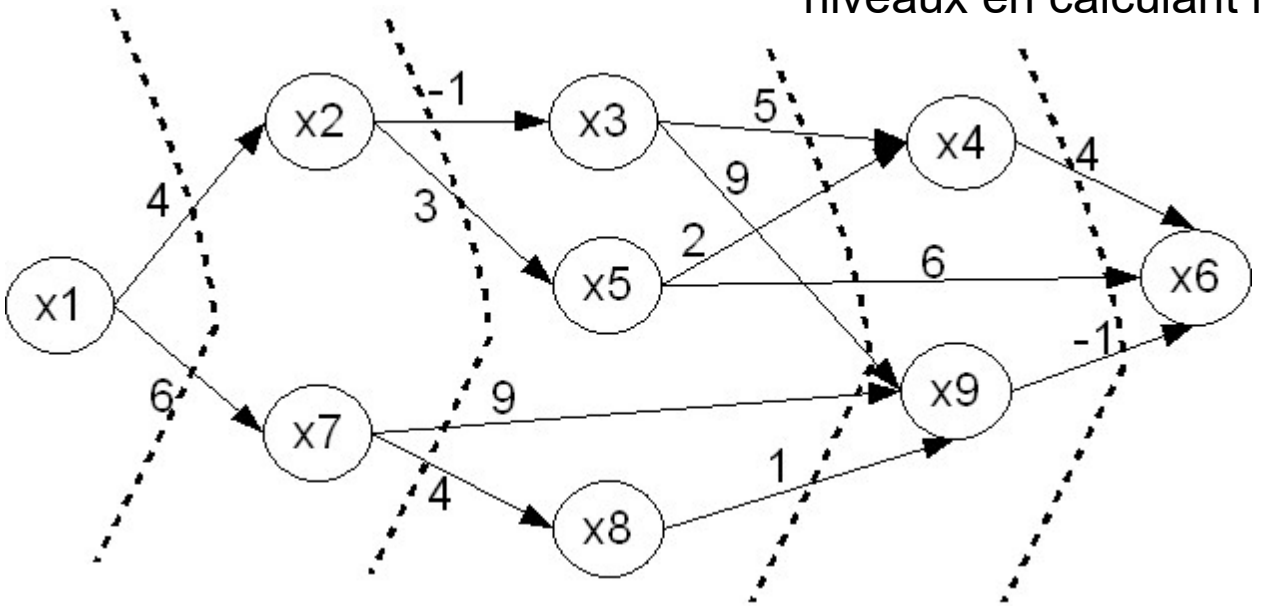
x1	x7	x8	x2	x5	x3	x9	x4	x6
0	6,x1	10,x7	4,x1	7,x2	3,x2	11,x8	8,x3	10,x9

3.4. Plus courts chemins – (Annexe)
Valuations quelconques & Graphe sans circuit

- **Exemple**

1. Calculer les niveaux

2. Effectuer le parcours dans l'ordre des niveaux en calculant les couts



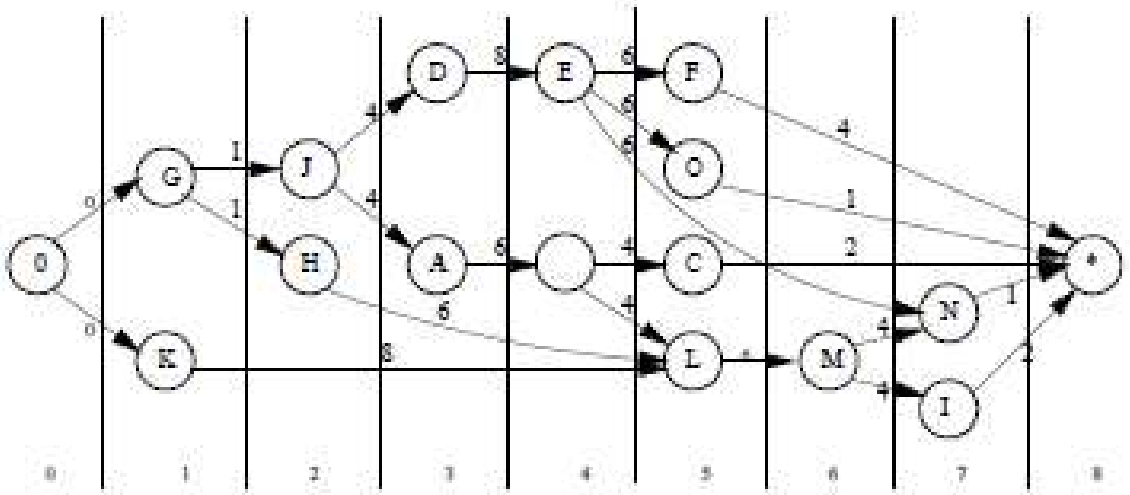
x1	x2	x7	x3	x5	x8	x4	x9	x6
0	4,x1	6,x1	3,x2	7,x2	10,x7	**8,x3**	**11,x8**	12,x4 10,x9

3.4. Plus courts chemins – Graphe sans circuit

- **Question**
  - Pour un graphe orienté sans circuit (DAG) sans ne comportant que des valuations positives :
    - Quels sont les algorithmes utilisables pour calculer les plus courts chemins d'un sommet vers tous les autres ?
    - Parmi ces algorithmes, quel est l'algorithme ayant la plus faible complexité (au pire)?

3.4. Plus courts chemins – Graphe sans circuit : Application

- **Application**
  - Calcul de la durée minimale d'un projet
    - Modélisation d'un projet par un graphe
      - Sommet = activités (début d'activités)
      - Arcs = relation de précédence entre activités
      - Valuation des arcs : durée minimale entre 2 début d'activités
      - Le graphe est orienté sans circuit



3.4. Plus courts chemins – Graphe sans circuit : Application

- **Application**
  - **Déterminer la durée minimale du projet** (avec cette modélisation)
    - Plus long chemin entre le début et la fin du projet
  - **Déterminer les dates de début au plus tôt des activités**
    - Plus long chemin entre le début et la fin du projet
  - **Déterminer les dates de début au plus tard des activités**
    - Plus long chemin entre le début et la fin du projet
  - Voir TD
- Adaptation de l'algorithme Bellman sans circuit au calcul des plus longs chemins

Plan

1. **Introduction**
2. **Parcours de Graphe**
3. **Optimisation et Graphes**
   - Plus courts chemins
     - Généralités
     - De 1 vers n - Valuations positives - Algorithme de Dijkstra
     - De 1 vers n - Valuations quelconques - Algorithme de Moore-Bellman-Ford
     - De 1 vers n - Graphe orienté sans circuit – Valuations quelconques – Algorithme de Bellman
     - Plus courts chemins de n vers n
   - Problèmes de flots

3.5. Plus courts chemins – De tous vers tous

- **Graphe quelconque**
  - On cherche à déterminer les plus courts chemins de tous les sommets vers tous les sommets
  - Algorithme dédié : Algorithme de Floyd(dit Floyd-Warshall)
    - Basé sur la représentation par matrice d'adjacence
    - Variante de l'algorithme de Warshall pour la fermeture transitive
  - Algorithmes précédents
    - Appliquer un des algorithmes calculant les plus courts chemins de 1 vers tous en itérant sur le sommet d'origine

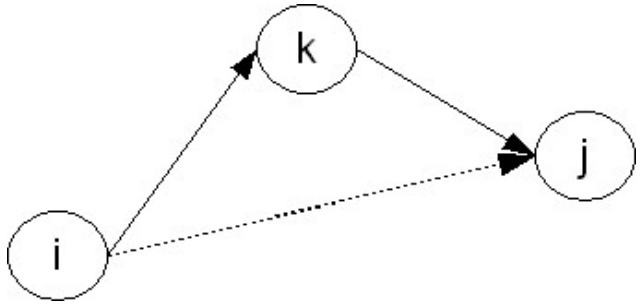
3.5. Plus courts chemins – De tous vers tous (Annexe)

- **Algorithme de Floyd**
  - Variante de l'algorithme de Warshall pour le calcul de la fermeture transitive
  - Basé sur la représentation par matrice d'adjacence

Valuation des arcs

- $C^0(i,i)=0$
- $C^0(i, j) =$ wij si l'arc (i,j) existe , $+\infty$ sinon
- $C^p(i, j) \leftarrow \text{Min}\ (\ C^{p-1}(i,j),\ C^{p-1}(i,k) + C^{p-1}(k,j)\ )$, pour tout $k \neq i, j$

- Opération ou ➔ Min
- Opération et ➔ +



3.5. Plus courts chemins – De tous vers tous

- **Itérer un algorithme de 1 vers tous**
  - N fois Dijkstra (valuations positives)
    - Complexité : $O(n.m.\log n) << O(n^3)$ pour graphes peu denses
  - N fois Bellman (valuations quelconques)
    - Complexité : $O(n^2.m)$
- **Algorithme dédié (Floyd-Warshall)**
  - Complexité : $O(n^3)$

3.5. Plus courts chemins – De tous vers tous (Annexe)

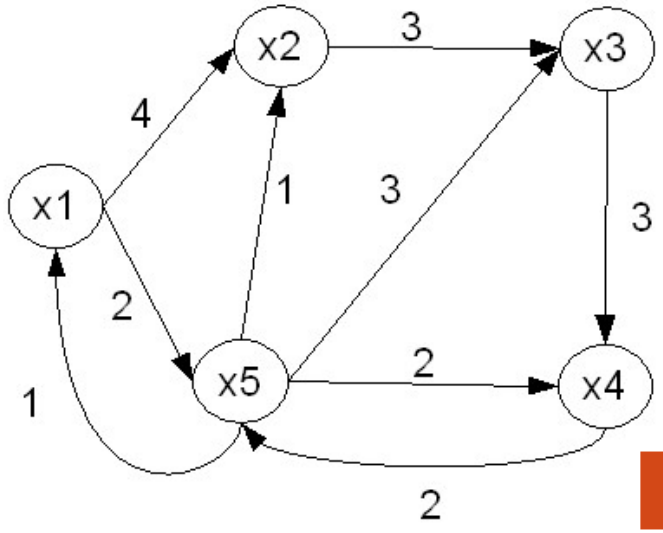
- **Algorithme de Floyd**
  - Itérations

```
For tous sommets k in 1..n
   For tous sommets i in 1..n sauf k
      if C(i,k)+C(k,i) < 0 then EXIT end if
      if C(i,k) ≠ +∞ then
         For tous sommets j in 1..n sauf i
            C(i, j) ← Min( C(i, j), C(i, k)+ C(k, j))
         end for
      end if
   end for
end for
```

  - Complexité : $O(n^3)$

3.5. Plus courts chemins – De tous vers tous : Exemple

- **Exemple**



	x1	x2	x3	x4	x5
x1	0	4	-	-	2
x2	-	0	3	-	-
x3	-	-	0	3	-
x4	-	-	-	0	2
x5	1	1	3	2	0

3.5. Plus courts chemins – De tous vers tous : Exemple

- **Solution :**
  - obtention du distancier

Plan

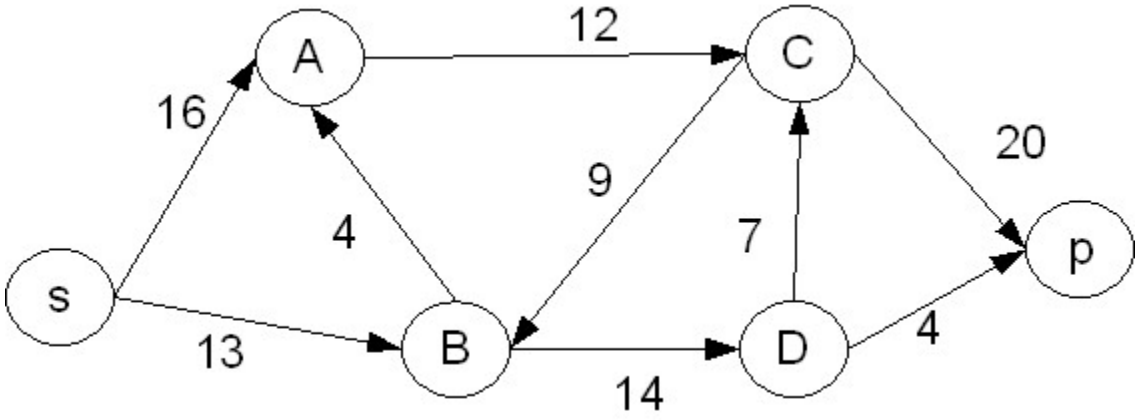
1. **Introduction**
2. **Parcours de Graphe**
3. **Optimisation et Graphes**
   - Plus courts chemins
   - Problèmes de flots
     - Généralités et Définitions
     - Problème de Flot Max & Coupe Min
     - Algorithmes de Ford Fulkerson / Edmunds - Karps
     - Problème de Flot Max à Cout Min et Algorithme de Busacker Gowen
     - Applications des flots

3.6. Problèmes de flots – Généralités (1)

- **Graphes de Flots** **Network Flow**
  - Modélisation de nombreux problèmes
  - Interprétation
    - Circulation d'un flot dans un réseau
    - Acheminer la plus grande quantité possible d'une origine (source) vers une destination (puits)
    - Les liens du graphe ont une capacité limitée
    - Il n'y a pas de création ni de perte de flot lors de l'acheminent
  - Autre terme : réseau de transport (transportation network)

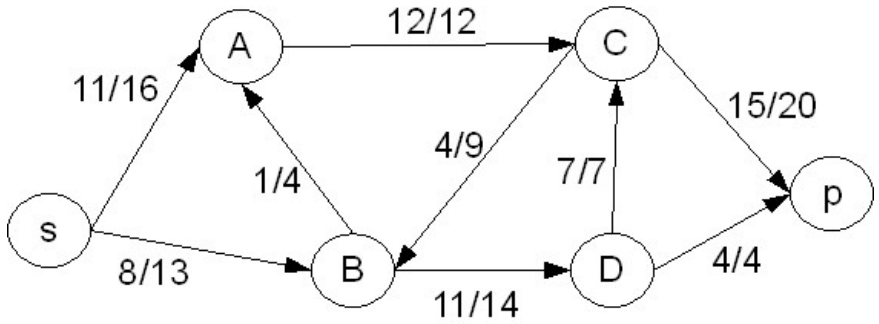
3.6. Problèmes de flots – Généralités (2)

- **Graphe de flot ou réseau de transport : G(X, A)**
  - Un graphe orienté valué avec
    - Un sommet source s : tous les sommets de X sont accessibles depuis s
    - Un sommet puits p : p est accessible depuis tous les sommets de X
    - Les valuations = les capacités de circulation sur les arcs
  - Exemple :
    - Trouver le trafic maximal entre 2 villes compte tenu des capacités des tronçons routiers



3.6. Problèmes de flots – Généralités (3)

- **Problème de flot maximal**
  - Maximiser la quantité de flot passant dans un réseau entre un sommet source et un sommet puits en respectant les contraintes de capacité des arcs
  - Exemple de valeurs pour les flots sur chaque arc :



- **Problème de flot maximal à cout minimal**
  - Un cout de transport est associé à chaque lien du réseau (en plus des capacités)
  - Déterminer un flot maximal de cout minimal entre un sommet source et un sommet puits en respectant les contraintes de capacité des arcs

3.6. Problèmes de flots – Généralités (4)

- **Applications des flots**
  - Problèmes d'approvisionnement
    - Réseau d'offres et de demandes
    - Les offres peuvent-elles satisfaire les demandes ?
  - Problèmes d'affectation (couplage biparti)
    - Est-il possible d'affecter un ensemble de personnes X à un ensemble d'activités Y (compte tenu des possibilités de X) ?
  - Problèmes de fiabilité
    - Existe-t-il plusieurs chemins entre deux points d'un réseau tels que ces chemins n'ont pas de sommets (ou arcs) intermédiaires communs

3.6. Problèmes de flots – Définitions (1)

- **Graphe de flots :** $G = (X, A, C)$
  - X : sommets A : arcs (i, j)
  - C : capacités : C_{ij} capacité de l'arc

- **Flot F : une solution**
  - F : de A → | N telle que F_{ij} : flot de l'arc (i, j)
    - La valeur de flot respecte la capacité sur chaque arc

$$0 \leq F_{ij} \leq C_{ij}, \forall (i, j) \in A$$

    - La quantité de flot est conservée à chaque sommet du graphe

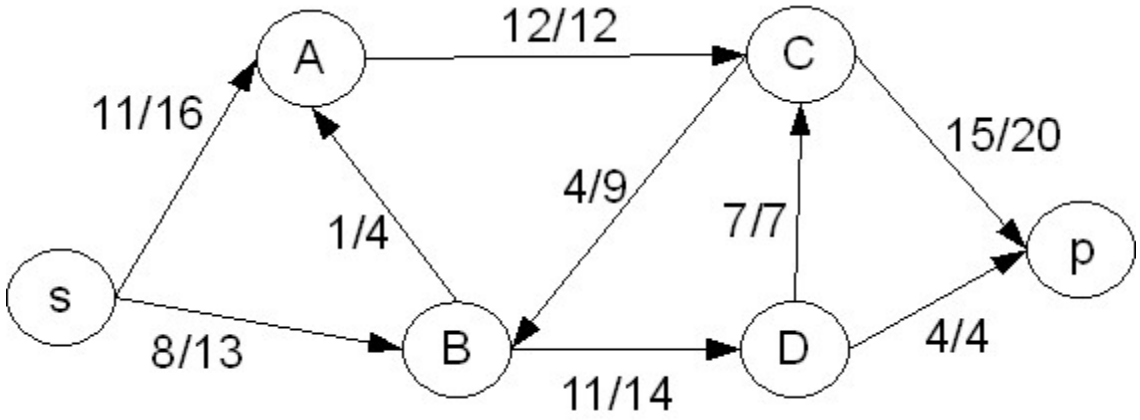
$$\sum_{x \in \delta^-(i)} F_{xi} = \sum_{y \in \delta^+(i)} F_{iy}, \forall i \neq s, p \in X$$

    - Débit total = quantité de flot partant du sommet source = quantité de flot arrivant au sommet puits

$$D = \sum_{x \in \delta^+(s)} F_{sx} = \sum_{y \in \delta^-(p)} F_{yp}$$

3.6. Problèmes de flots – Définitions (2)

- **Exemple**



  - Le flot F est admissible :
    - toutes les valeurs de flot des arcs respectent les contraintes
  - Le débit total D = 19

3.6. Problèmes de flots – Définitions (3)

- Arc saturé $F_{ij} = C_{ij}$
  - La valeur de flot de l'arc est égal à la capacité
- Arc nul $F_{ij} = 0$
  - La valeur de flot sur l'arc est nulle

- Solution
  - Flot complet
    - Tout chemin de s à p contient au moins un arc saturé
  - Flot nul
    - Le flot circulant dans tout le graphe est égal à 0 (débit total = 0)
    - Le flot nul est un flot admissible

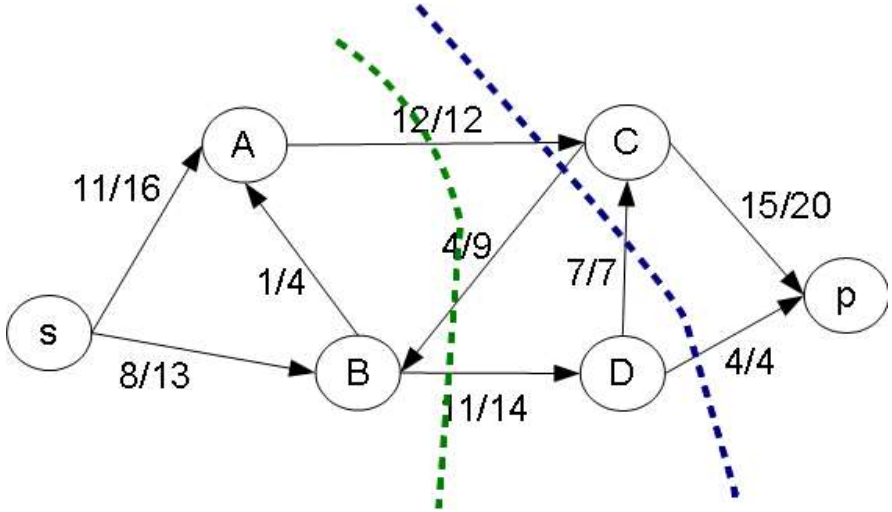
3.6. Problèmes de flots – Coupe (1)

- **Coupe d'un graphe de flot** $G = (X, A, C)$
  - Une partition de l'ensemble X en 2 ensembles S et $\overline{S}$:
    - $s \in S$; $p \in \overline{S}$ et $S \cap \overline{S} = \emptyset$
    - Peut également être définie par l'ensemble des arcs ayant
      - Une extrémité dans S et l'autre extrémité dans $\overline{S}$

  - Soit une coupe $K = (S, \overline{S})$
    - Arcs entrants dans K = arcs orientés de $\overline{S}$ vers S
    - Arcs sortants de K = arcs orientés de S vers $\overline{S}$
    - **Flot net de K = somme des flots des arcs entrants (-) et sortant (+)**
    - **Capacité de K = somme des capacités des arcs sortants**

3.6. Problèmes de flots –Coupe (2)

- **Exemple**
  - Evaluer les flots et les capacités des coupes



- Coupe verte
  - Flot = 12 – 4 + 11 = 19
  - Capacité = 12 + 14 = 26

Coupe bleue

Flot =

Capacité =

Plan

1. **Introduction**
2. **Parcours de Graphe**
3. **Optimisation et Graphes**
   - Plus courts chemins
   - Problèmes de flots
     - Généralités et Définitions
     - Problème de Flot Max & Coupe Min
     - Algorithmes de Ford Fulkerson / Edmunds - Karps
     - Problème de Flot Max à Cout Min et Algorithme de Busacker Gowen
     - Applications des flots

3.7. Flot maximal – Coupe Minimale (1)

- **Soit un graphe de flots G(X, A, C)**
  - Flot Maximal
    - On cherche à déterminer les valeurs de F pour tout arc de telle sorte que le débit total soit maximal
  - Coupe Minimale
    - On cherche une coupe (une partition des sommets) de capacité minimale
  - Résolution
    - Modélisation et outils de programmation linéaire
    - Modélisation et algorithmes de graphes

3.7. Flot maximal

- **Exemple de modélisation en PL :**

$$\text{Max } D$$

$$\sum_{i \in \delta^+(s)} (F_{si} - D) = 0 \text{ (source } s)$$

$$\sum_{i \in \delta^i(p)} (F_{ip} + D) = 0 \text{ (puits } p)$$

$$-\sum_{x \in \delta^-(i)} F_{xi} + \sum_{y \in \delta^+(i)} F_{iy} = 0, \forall i \neq s, p \in X$$

$$F_{ij} \leq C_{ij} \ \forall (i,j) \in A$$

$$F_{ij} \geq 0 \ \forall (i,j) \in A$$

$$\text{Variables entières } F_{ij}, D$$

PL :
Matrice unimodulaire
Solutions entières

3.7. Coupe minimale (1)

- **Théorème : Le débit de tout flot est inférieur ou égal à la capacité de toute coupe (Th. Ford Fulkerson)**
  - Démonstration
    - Flot sortant des sommets de S = Flot entrant dans les sommets de S

      = Flot entrant en s + Flot entrant dans les autres sommets de S

      $$\sum_{i \in S} \sum_{x \in \delta^+(i)} F_{ix} = D + \sum_{y \in \delta^-(i)} \sum_{i \in S-s} F_{yi}$$

    - Supprimer des 2 cotés les arcs ayant les 2 extrémités dans S

      $$\sum_{i \in S} \sum_{x \in \overline{S}} F_{ix} = D + \sum_{y \in \overline{S}} \sum_{i \in S-s} F_{yi}$$

    - C'est-à-dire : $$D = \sum_{i \in S} \sum_{x \in \overline{S}} F_{ix} - \sum_{y \in \overline{S}} \sum_{i \in S-s} F_{yi}$$

    - Comme $F_{ij} \leq C_{ij}$ on a : $$D \leq \sum_{i \in S} \sum_{x \in \overline{S}} C_{ix} - \sum_{y \overline{S}} \sum_{i \in S-s} F_{yi}$$

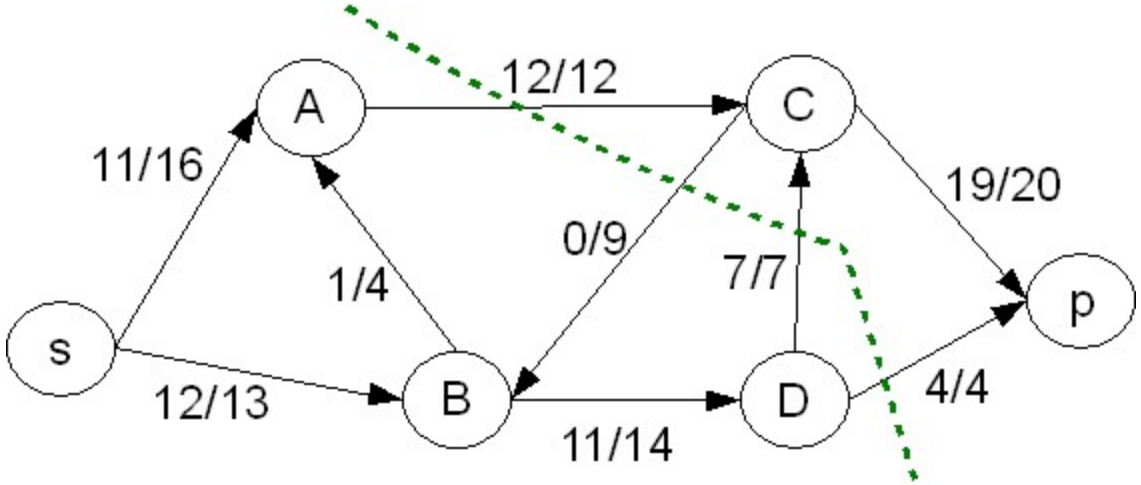
3.7. Coupe minimale (2)

- On a : $D \leq \sum_{i \in S} \sum_{x \in \overline{S}} C_{ix} - \sum_{y \overline{S}} \sum_{i \in S-s} F_{yi}$
- Or par définition : $C(K) = \sum_{i \in S} \sum_{j \in X-S} C_{ij}$
- D'où $D \leq C(K) - \sum_{y \in \overline{S}} \sum_{i \in S-s} F_{yi}$
- Comme $F_{ij} \geq 0$ on a la propriété : $D \leq C(K)$

- Résoudre le problème de Coupe Min revient à résoudre le problème de Flot Max

3.7. Coupe minimale (3)

- **Application**
  - Exemple
    - Coupe capacité 23
    - Débit flot = 23



  - Ce flot est maximal et cette coupe est minimale

Plan

1. **Introduction**
2. **Parcours de Graphe**
3. **Optimisation et Graphes**
   - Plus courts chemins
   - Problèmes de flots
     - Généralités et Définitions
     - Problème de Flot Max & Coupe Min
     - Algorithmes de Ford Fulkerson / Edmunds - Karps
     - Problème de Flot Max à Cout Min et Algorithme de Busacker Gowen
     - Applications des flots

3.8. Flot maximal – Graphe d'écart (1)

- **Soit un graphe de flots G(X, A, C) et soit un flot F**
- **Le graphe d'écart de G par rapport à F** G_F^e
  - est un graphe orienté valué tel que
    - Les sommets sont ceux de de G
    - Les arcs sont définis par :
      - Tout arc (i,j) de G non saturé $F_{ij} \neq C_{ij}$ est dans G_F^e
      - Il est valué par $C_{ij} - F_{ij}$

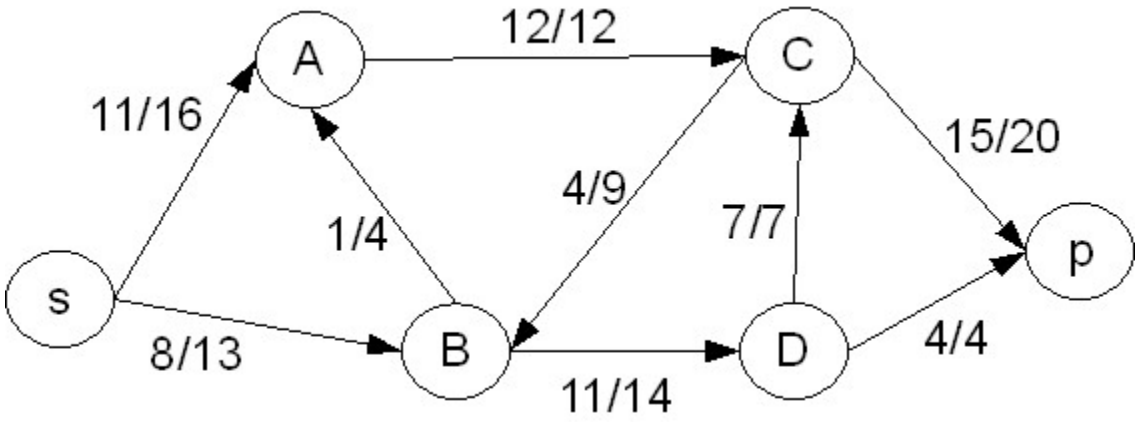
Quantité de flot que l'on peut ajouter de i à j

      - Tout arc (i,j) de G de flot non nul $F_{ij} \neq 0$ est présent en sens inverse dans G_F^e
      - Il est valué par F_{ij}

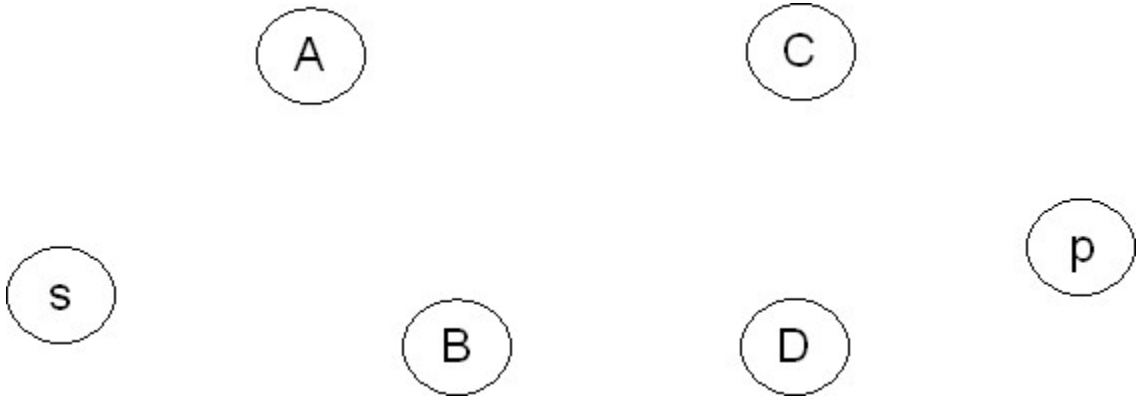
Quantité de flot que l'on peut retirer de i à j

3.8. Flot maximal – Graphe d'écart (2)

- **Exemple.**
  - Graphe de Flots



  - Graphe d'écart



3.8. Flot maximal – Graphe d'écart (3)

- **Le graphe d'écart G_F^e est composé de :**
  - Arcs A+ : permettant une augmentation du flot
  - Arcs A- : permettant une diminution du flot

- **Propriété**
  - On peut augmenter le flot F dans un réseau de transport G ssi il existe un chemin de s à p dans le graphe d'écart G_F^e
  - Le flot est maximal ssi il n'existe pas de chemin de s à p dans le graphe d'écart

3.8. Problèmes de flot maximal – Algorithme de Ford Fulkerson (1)

- **Initialisation**
  - Flot initial Nul $D = 0,\ F_{ij} = 0,\ \forall (i,j) \in A$

- **Itérations**

```
repeat
    chercher un chemin de s à p dans G_F^e
    calculer la variation de flot de ce chemin: δ
    mettre à jour le graphe de flot
    mettre à jour D : D = D + δ
until (plus de chemin)
```

3.8. Problèmes de flot maximal – Algorithme de Ford Fulkerson (2)

- **Chercher un chemin de s à p**
  - Utiliser un algorithme de parcours
    - Il existe un chemin si le sommet p est marqué en fin du parcours

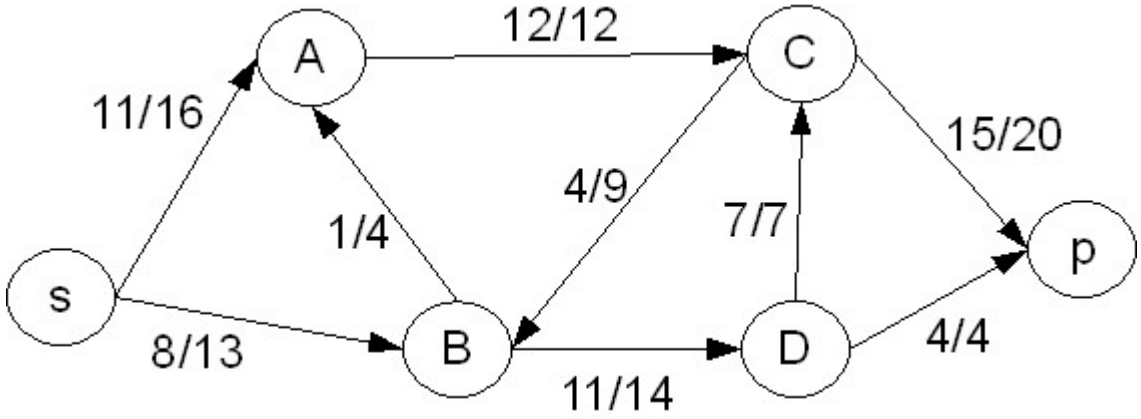
- **Calculer la variation de flot pour ce chemin**
  - Le chemin traverse des arcs « avant » A^+ et des arcs « arrière » A^-

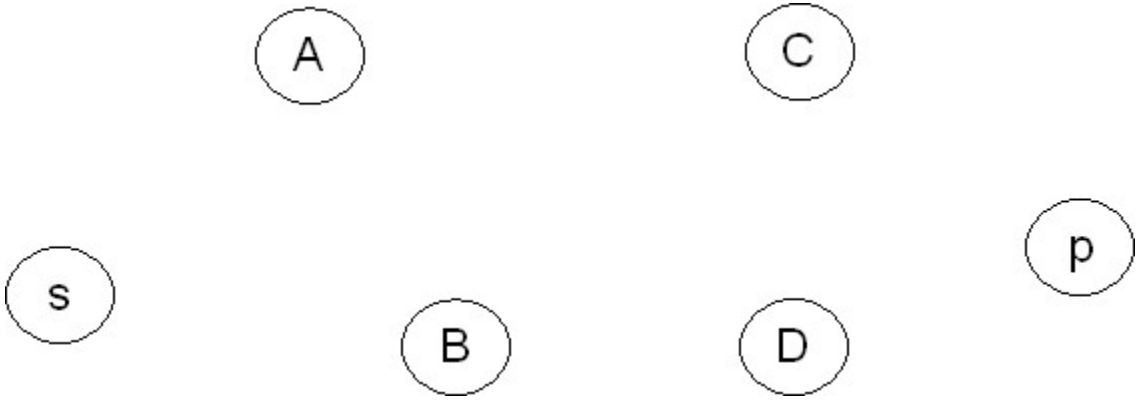
$$\delta = \min(\min_{(ij)\in A^+}(C_{ij} - F_{ij}), \min_{(ij)\in A^-}(F_{ij}))$$

- **Mettre à jour le graphe de flot**
  - Augmenter de δ les arcs associés à ceux de A^+ dans G_F^e
  - Diminuer de δ les arcs associés à ceux de A^- dans G_F^e

3.8. Problèmes de flot maximal – Algorithme de Ford Fulkerson (3)

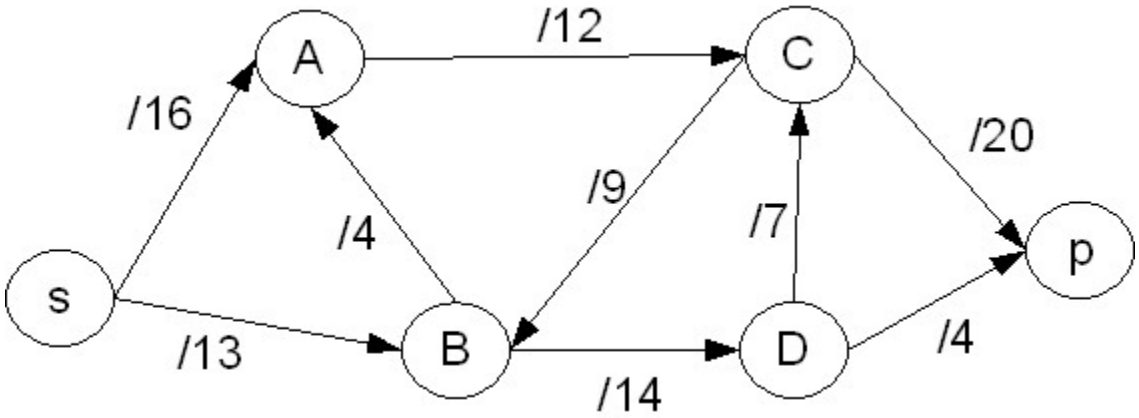
- **Exemple. Déterminer le flot maximal à partir du graphe de flot ci-dessous**

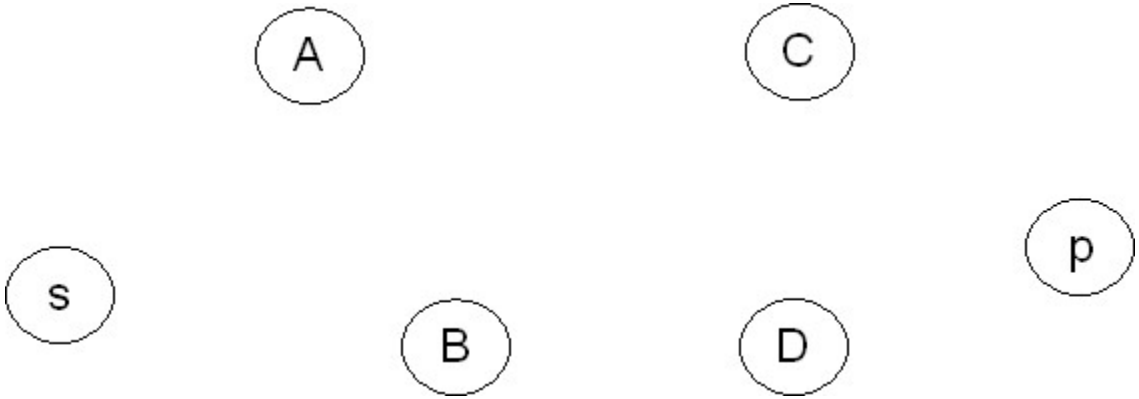




3.8. Problèmes de flot maximal – Algorithme de Ford Fulkerson (4)

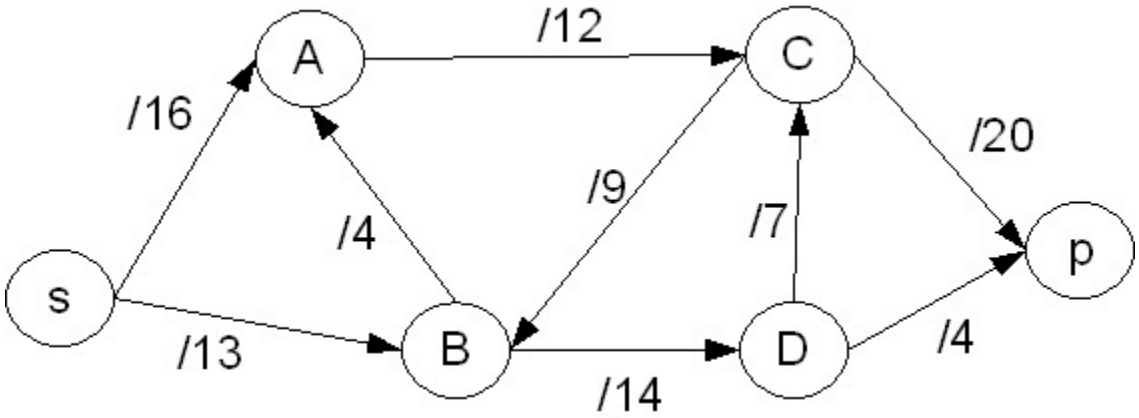
- **Exemple (suite)**

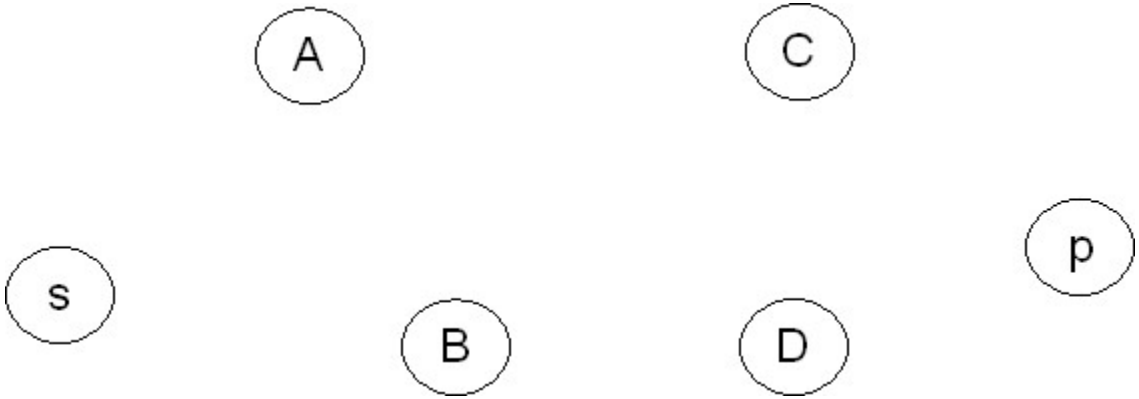




3.8. Problèmes de flot maximal – Algorithme de Ford Fulkerson (5)

- **Exemple (suite)**

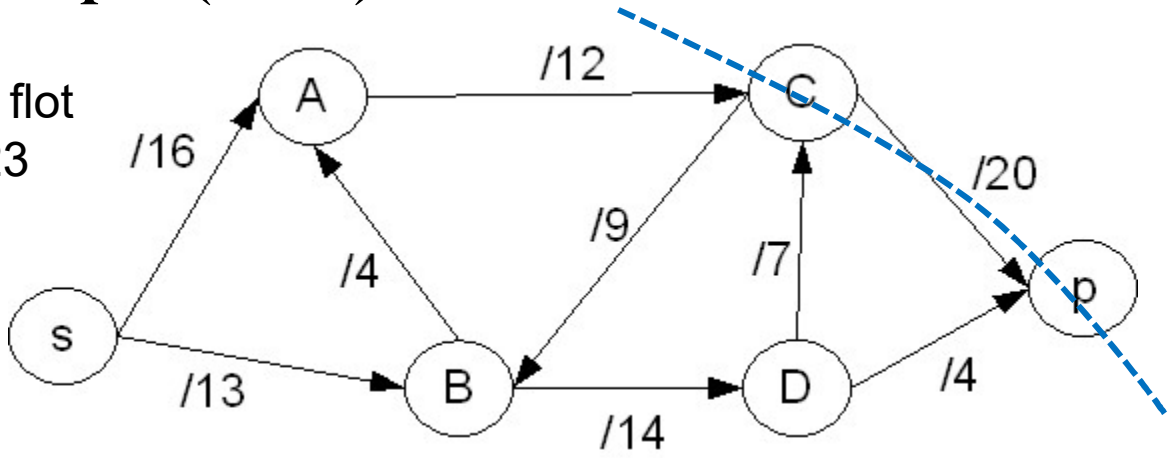




3.8. Problèmes de flot maximal – Algorithme de Ford Fulkerson (6)

- **Exemple (suite)**

Graphe flot
Débit=23



Débit 23 (graphe de flot) est optimal

Coupe :
{s, A, B, D} {C,p}

3.8. Problèmes de flot maximal – Algorithme de Ford Fulkerson (6)

- **Remarques**
  - Implémentation
    - 1 seul graphe (et « simuler » le graphe d'écart)
      - On considère des « arêtes » (parcours avant et arrière); on cherche une « chaine » de s à p
  - Initialisation : Flot nul ou tout flot admissible

- **Propriétés :**
  - En fin d'algorithme
    - Un ensemble de sommets marqués par le dernier parcours
    - Un ensemble de sommets non marqués
  - Si les capacités sont entières alors
    - Le débit du flot maximal F trouvé par est entier
    - Sur chaque arc, le flot est un entier

3.8. Problèmes de flot maximal – Algorithme de Ford Fulkerson (7)

- **Recherche d'un chemin dans le graphe d'écart**
  - Algorithme de parcours
    - En profondeur ?
    - En largeur ?
      - En largeur ➔ Edmund Karp
      - Parcours déterminant un chemin améliorant le plus court en nombre d'arcs
    - Convergence de l'algorithme

3.8. Problèmes de flot maximal – Algorithme de Ford Fulkerson (8)

- **Complexité**
  - A chaque itération :
    - Recherche d'un chemin : O(m)
    - Variation de flot sur ce chemin : O(n)
  - Total : O(m+n) = O(m) si graphes peu denses
  - Nombre d'itérations
    - À chaque étape : variation de 1 de la capacité d'une coupe
    - Capacité maximale d'une coupe : (n-1).Kmax où Kmax représente la valeur maximale des capacités des arcs
    - D'où (n-1)Kmax itérations
  - D'où la complexité : O(m.n.Kmax)

3.8. Problèmes de flot maximal – Algorithme de Ford Fulkerson (9)

- **Correction de l'algorithme**
  - Par définition on a $D = \sum_{i \in S} \sum_{x \in \overline{S}} F_{ix} - \sum_{y \in \overline{S}} \sum_{i \in S-s} F_{yi}$ (p 142)

  - Le flot D obtenu en fin de l'algorithme est maximal
    - On ne peut plus atteindre le sommet puits depuis le sommet source
    - On a donc obtenu une séparation des sommets en deux ensembles :
      - S (contenant la source) et $\overline{S}$ (le complémentaire)

    - Dans la coupe obtenue en fin d'algorithme :
      - Tout arc sortant de S est saturé $F_{ij} = C_{ij} \ \forall i \in S \ \forall j \in \overline{S}$
      - Tout arc entrant dans $\overline{S}$ est de flot nul $F_{ji} = 0 \ \forall j \in \overline{S} \ \forall i \in S$
        - Sinon on aurait trouvé un chemin de la source vers le puits

    - D'où la preuve $D = \sum_{i \in S} \sum_{j \in \overline{S}} C_{ij} = C(K)$

3.8. Problèmes de flot maximal – Autre algorithme : Edmunds-Karp

- **Algorithme d'Edmunds-Karp**
  - Variante de l'algorithme de Ford Fulkerson avec
    - Parcours BFS déterminant un chemin améliorant le plus court en nombre d'arcs
    - Convergence garantie de cet algorithme quelque soit les capacités
    - Complexité $O(nm^2)$ (indépendante de la valeur des capacités)

Plan

1. **Introduction**
2. **Parcours de Graphe**
3. **Optimisation et Graphes**
   - Plus courts chemins
   - Problèmes de flots
     - Généralités et Définitions
     - Problème de Flot Max & Coupe Min
     - Algorithmes de Ford Fulkerson / Edmunds - Karps
     - Problème de Flot Max à Cout Min et Algorithme de Busacker Gowen
     - Applications des flots

3.9. Problèmes de flot max de cout minimal (1)

- **Graphe de flots :** $G = (X, A, C, W)$
  - X : sommets (avec s source et p puits)
  - A : arcs
  - C : C_{ij} capacité de l'arc (i, j)
  - W : W_{ij} cout de l'arc (i, j)
- **Flot**
  - Mêmes caractéristiques (respect des capacités, conservation du flot, débit total)
- **Flot maximal de cout minimal**
  - Trouver un flot maximal de s à p et de cout minimal

3.9. Problèmes de flot max de cout minimal (2)

- **Graphe d'écart** G_F^e
  - Même principe que pour le problème de flot max
    - Tout arc (i,j) de G non saturé $F_{ij} \neq C_{ij}$ est dans G_F^e
      - Il est valué par $C_{ij} - F_{ij}$ et un cout W_{ij} arcs « en avant »
    - Tout arc (i,j) de G de flot nul $F_{ij} \neq 0$ est **en sens inverse** dans G_F^e
      - Il est valué par F_{ij} et un cout $-W_{ij}$ arcs « en arrière »
  - Recherche d'un chemin augmentant dans le graphe d'écart
    - Variation du flot d'une valeur le long du chemin : δ
    - Variation du cout total (par unité de flot) : m

3.9. Problèmes de flot max de cout minimal – Propriété

- **Obtention d'un flot de cout minimal**
  - Un flot de débit D de s à p est de cout minimal si parmi tous les flots de même débit il n'existe pas de circuit de longueur négative dans le graphe d'écart
    - Propriété admise (voir PL ?)

- **2 types d'algorithme**
  - V1 : partir d'une solution de flot maximal et rendre le cout minimal en supprimant les circuits de longueur négative
  - V2 : construire progressivement la solution avec des débits croissants mais tous de cout minimal (Busacker-Gowen)

3.9. Problèmes de flot max de cout minimal – Algorithme de Busacker Gowen

- **Initialisation**
  - Flot initial nul $D = 0,\ F_{ij} = 0,\ \forall (i,j) \in A$
  - Cout total nul $Z = 0$
- **Itérations**

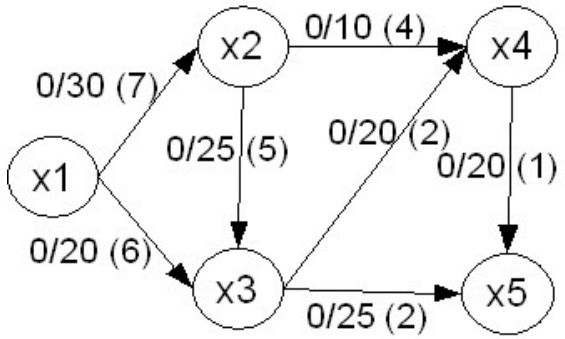
repeat

chercher un chemin de cout minimal m de s à p dans G_F^e
calculer la variation de flot de ce chemin: δ
mettre à jour le graphe de flot
mettre à jour D : $D = D + \delta$
mettre à jour Z : $Z = Z + m.\delta$

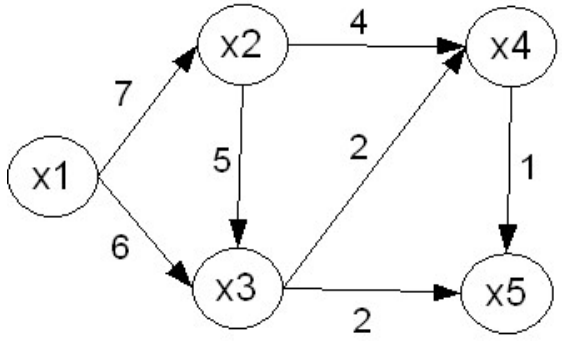
until (plus de chemin)

3.9. Problèmes de flot max de cout minimal – Exemple Algorithme de Busacker Gowen (1)

- **Graphe de Flot**

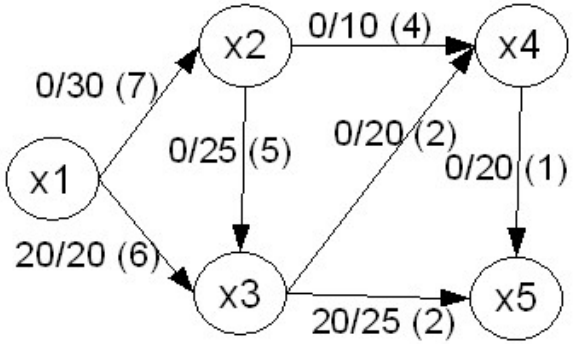


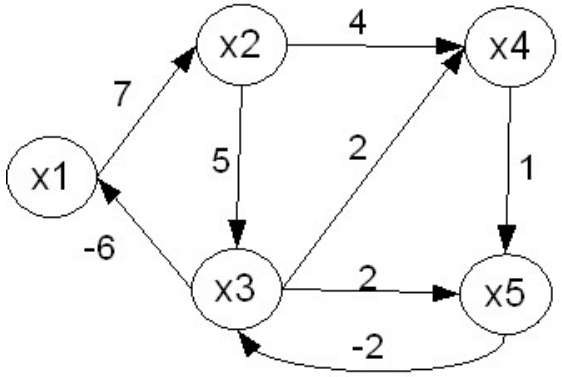
Graphe d'écart



PCC avec Dijkstra

- Cout min en x5=8; Var. flot = 20 (x1→ x3→ x5)



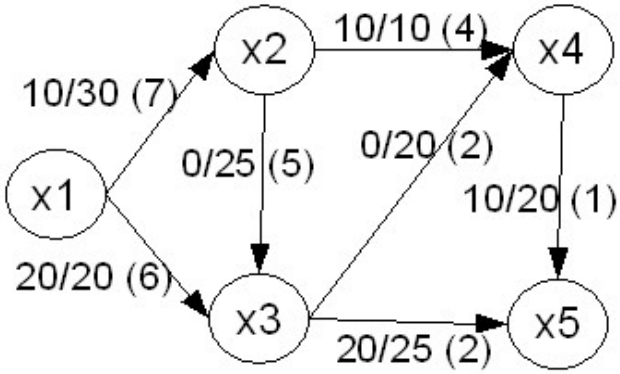


PCC avec Bellman-Ford

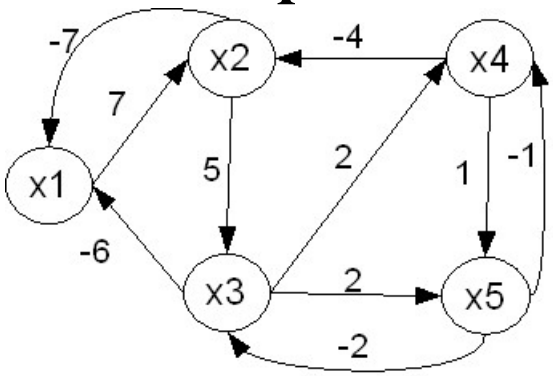
- Cout min en x5=12; Var flot=10 (x1→x2→x4→x5)

3.9. Problèmes de flot max de cout minimal – Exemple Algorithme de Busacker Gowen (2)

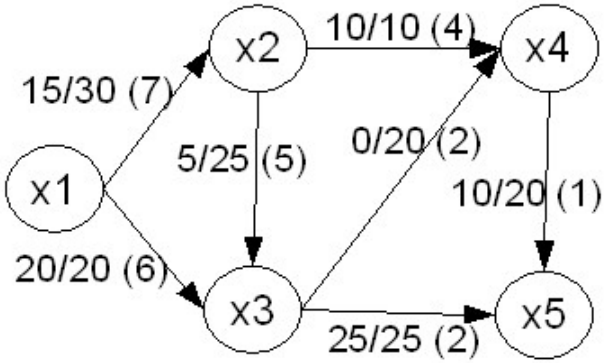
- **Graphe de Flot**

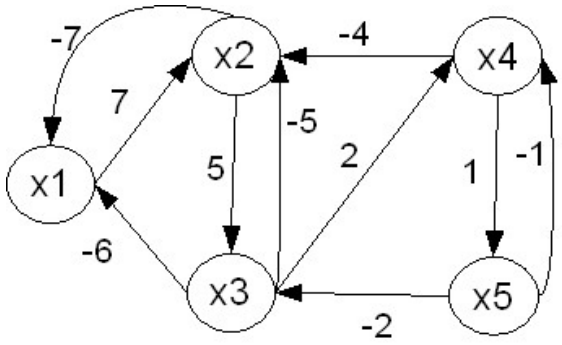


Graphe d'écart



Cout min x5=14; Var. flot = 5 (x1→x2→x3→ x5)

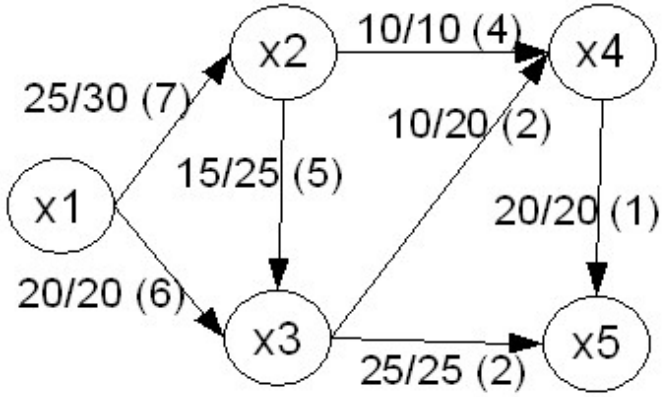




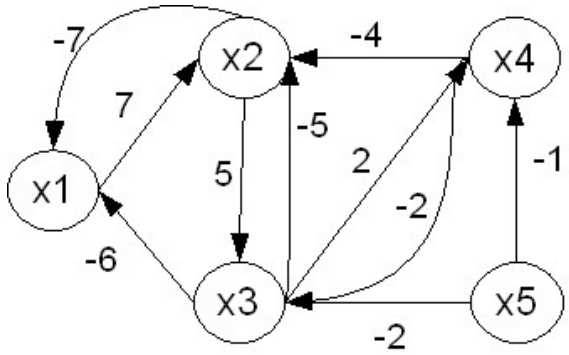
- Cout min x5=15; Var flot=10 (x1→x2→x3→ x4→x5)

3.9. Problèmes de flot max de cout minimal – Exemple Algorithme de Busacker Gowen (3)

- **Graphe de Flot**



Graphe d'écart

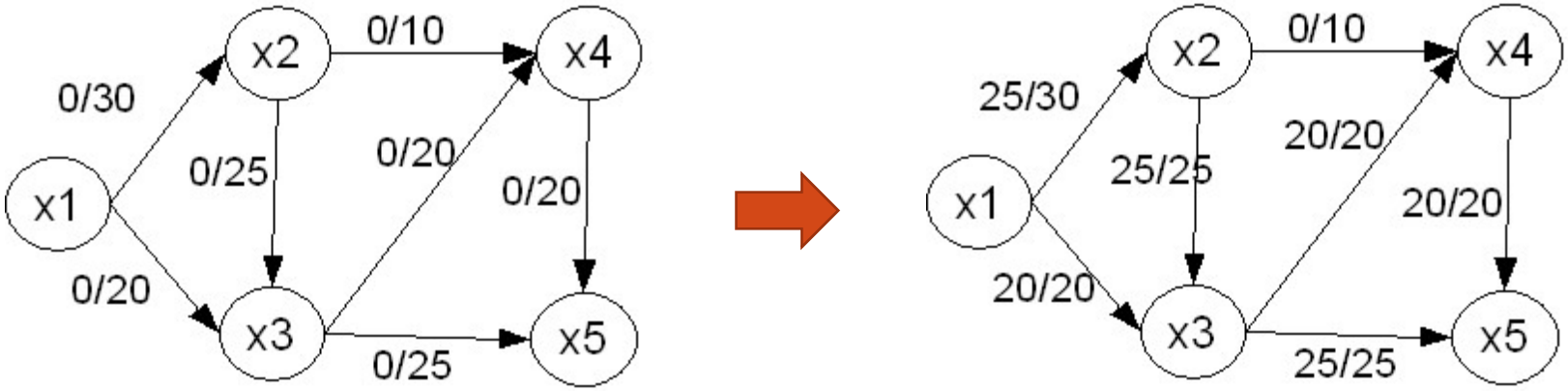


Plus de chemin de x1 vers x5

- Solution
  - Flot max de débit D = 45
  - Cout = 8 * 20 + 12 * 10 + 14 * 5 + 15 * 10 = 500
  - = 25*7 + 20*6 + 15*5 + 10*4 + 10*2 + 25*2 + 20*1

3.9. Problèmes de flot max de cout minimal – Exemple Algorithme de Busacker Gowen (4)

- **Si on calcule le flot maximal sur le même problème, on obtient par exemple :**



  - Le débit total est égal à 45
  - Mais le cout de cette solution de flot maximal est égal à :
    - 25*7+20*6+25*5+20*2+25*2+20*1=530

Plan

1. **Introduction**
2. **Parcours de Graphe**
3. **Optimisation et Graphes**
   - Plus courts chemins
   - Problèmes de flots
     - Généralités et Définitions
     - Problème de Flot Max & Coupe Min
     - Algorithmes de Ford Fulkerson / Edmunds - Karps
     - Problème de Flot Max à Cout Min et Algorithme de Busacker Gowen
     - Applications des flots

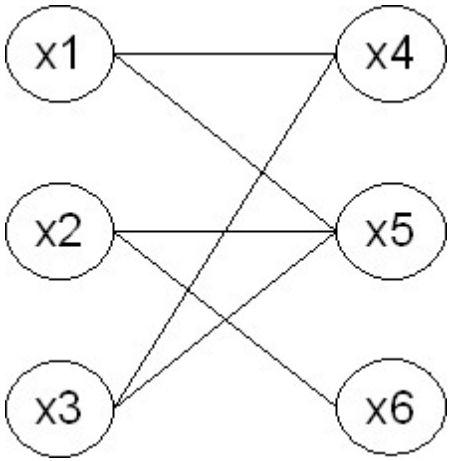
3.10. Applications des flots – Couplage biparti (1)

- **Couplage**
  - Soit G(X, E) un graphe non orienté
    - Un couplage C : un sous ensemble des arêtes de E tel que les arêtes prises 2 à 2 n'ont pas de sommet en commun
    - Un sommet x est saturé par un couplage C :
      - Il est extrémité d'une des arêtes de C
    - Sinon x est dit insaturé (ou libre)
  - Un couplage C est parfait ssi
    - tous les sommets du graphe sont saturés par C
  - Problèmes de couplage
    - Couplage maximal = couplage de cardinalité maximale
    - Couplage maximal à cout minimal/maximal (cout sur les arêtes)

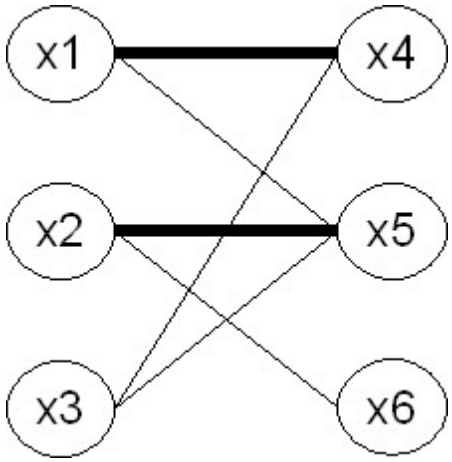
3.10. Applications des flots – Couplage biparti (2)

- **Graphe biparti G(X, Y, E)**
  - Arêtes uniquement entre les sommets de X et de Y

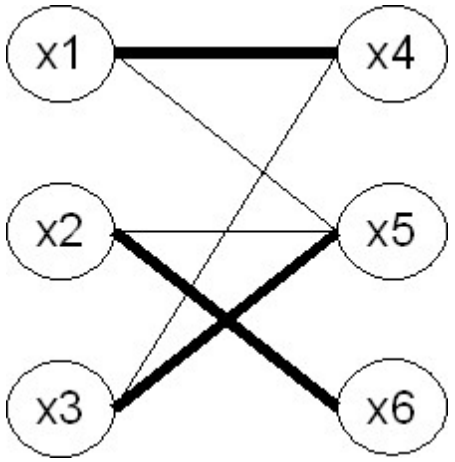
  - Exemple :



Couplage cardinalité 2



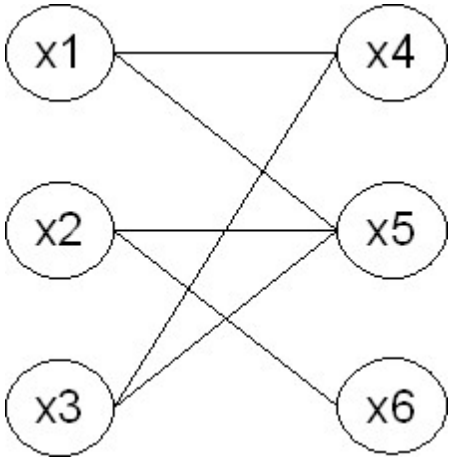
Couplage cardinalité 3

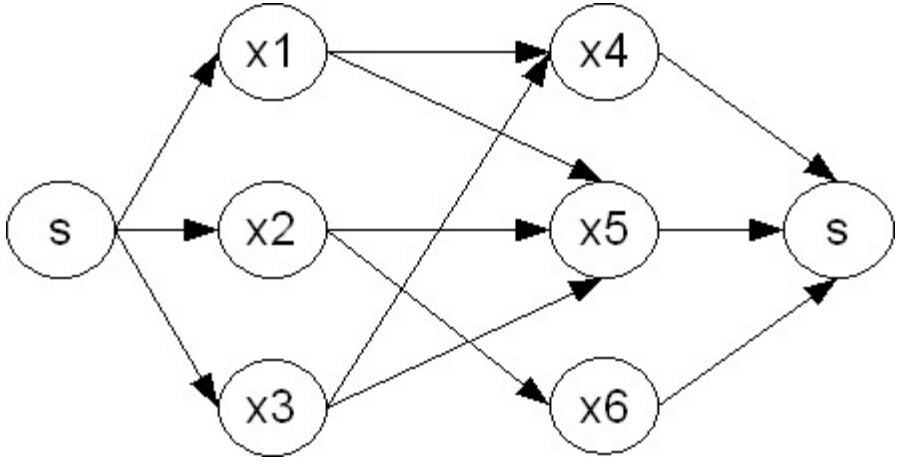


Couplage parfait

3.10. Applications des flots – Couplage biparti (3)

- **Modélisation via un graphe de flot**
  - Ajout d'un sommet source et d'un sommet puits
  - Orienter les arêtes (de s vers X, de X vers Y et de Y vers p)

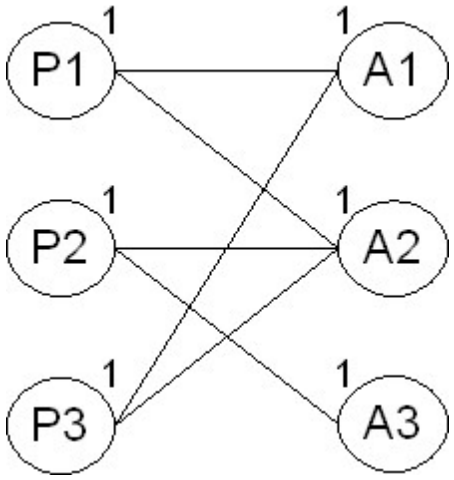


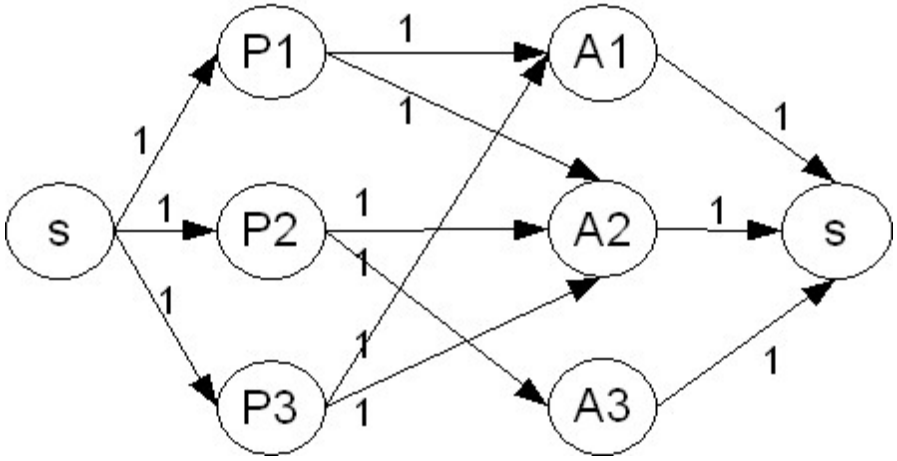


  - Capacités des arêtes
    - Cela dépend du problème de couplage ….

3.10. Applications des flots – Couplage biparti (4)

- **Exemple**
  - Modélisation du problème d'affectation suivant :
    - 3 personnes (P1, P2, P3) pour 3 tâches (A1, A2, A3)
    - P1 → A1 ou A2; P2 → A2 ou A3 et P3 → A1 ou A2
    - Chaque personne ne peut faire qu'une seule tâche
    - 1 tâche ne nécessite qu'une seule personne

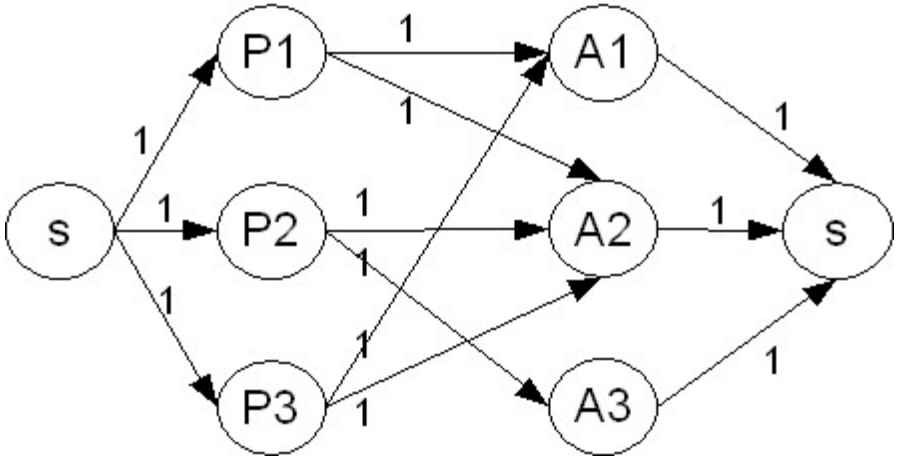




3.10. Applications des flots – Couplage biparti (5)

- **Exemple (suite)**
  - Algorithme de Flot Max
    - Débit total =

      cardinalité couplage maximal



    - La valeur du flot max = le nombre de personnes affectées à une tâche = le nombre de tâches ayant une personne allouée

  - Variantes
    - Valeurs des capacités
    - Ajout de cout (par exemple préférences)

  - Voir TD

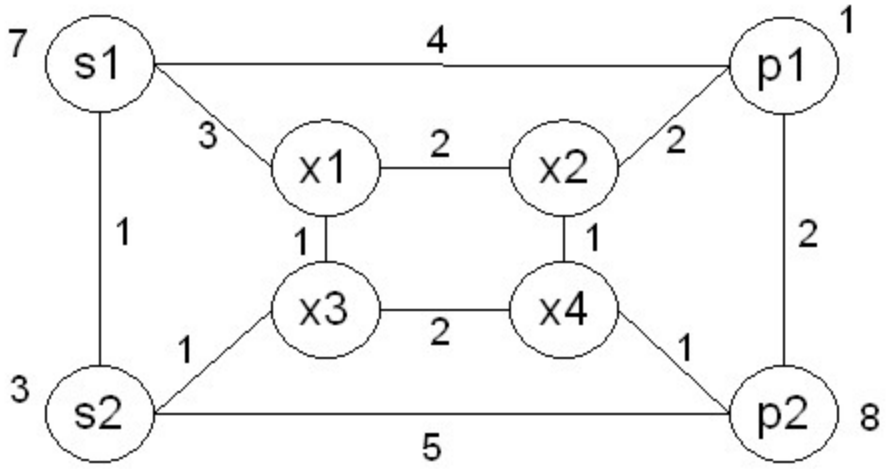
3.10. Applications des flots – Réseaux d'offres et de demandes (1)

- **Soit un graphe composé**
  - D'un ensemble S de sommets source ayant chacun une offre
  - D'un ensemble P de sommets puis ayant chacun une demande
  - D'un ensemble V de sommets intermédiaires permettant une circulation des sommets de S vers les sommets de P
  - Réseaux de transport reliant les sommets S, P et V, avec capacités (et couts) sur les arcs

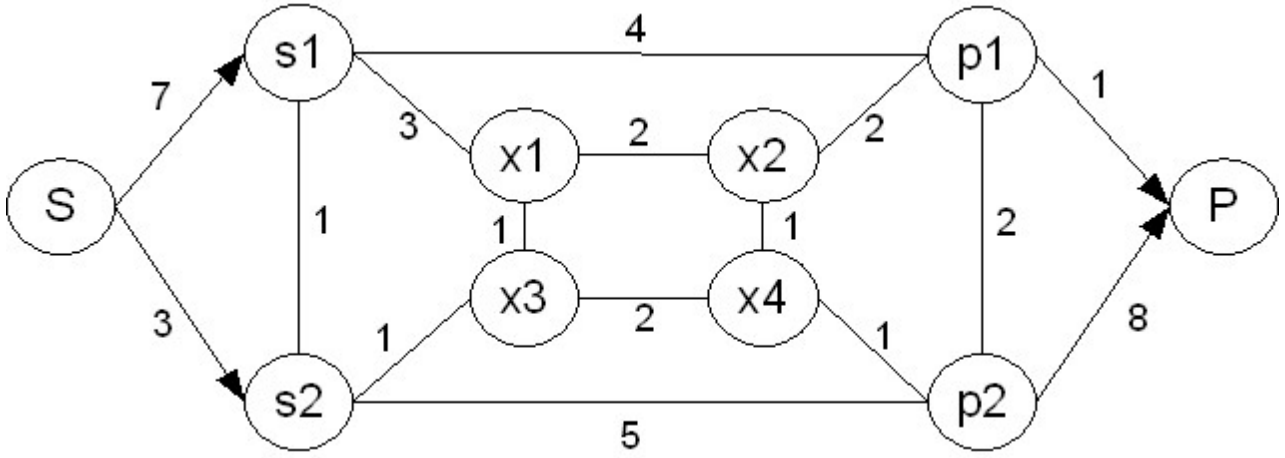
- Problème de l'offre et de la demande :
  - Existe-t-il une façon de faire circuler les produits des sources vers les puits de manière à satisfaire les demandes ?
  - Rechercher un flot de cout minimal satisfaisant les demandes
  - Transformation en un problème de flot maximum (à cout minimal)

3.10. Applications des flots – Réseaux d'offres et de demandes (2)

- **Exemple**



- **Transformation en un graphe de flots**



3.10. Applications des flots – Réseaux d'offres et de demandes (3)

- **Résolution**
  - Chercher un flot max de S vers P
    - Si la solution sature les arcs des sommets Pi vers P alors
      - Les demandes sont satisfaites
    - Sinon les demandes ne peuvent être satisfaites
  - Application (voir TD)

3.10. Applications des flots – Réseaux d'offres et de demandes (4)

- **Autre exemple :**
  - On veut acheminer un produit à partir de 3 entrepôts (1,2,3) vers 4 clients (a,b,c,d)
    - Quantités en stock : 45, 25, 25
    - Demande des clients : 30,10, 20, 30
  - Limitations en matière de transport d'un entrepôt à un client

	a	b	c	d
1	10	15	-	20
2	20	5	5	-
3	-	-	10	10

  - Donner le graphe de flots associé à ce problème
  - Est-il possible de satisfaire les demandes des clients ?

3.10. Applications des flots – Chemins disjoints

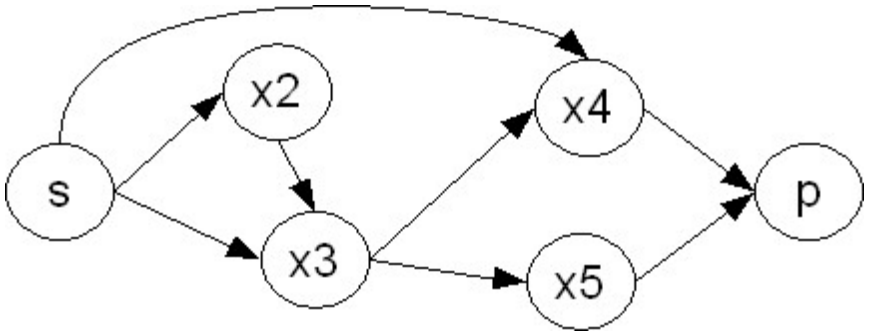
- **Chemins disjoints**
  - En termes de sommets
  - En termes d'arcs

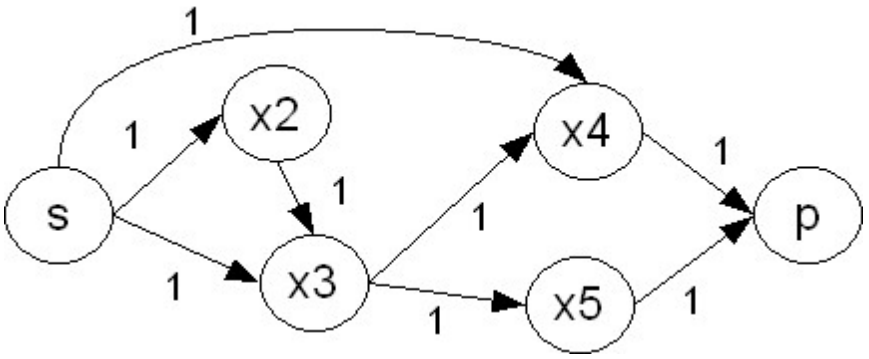
- **Recherche de chemins arcs-disjoints**
  - Soit un graphe G(X, A)
    - On détermine le graphe de flot
      - Même ensemble de sommets
      - Même ensemble d'arcs
      - Capacité = 1 sur chaque arc

      - Le débit total obtenu en cherchant le flot maximal = le nombre de chemins disjoints

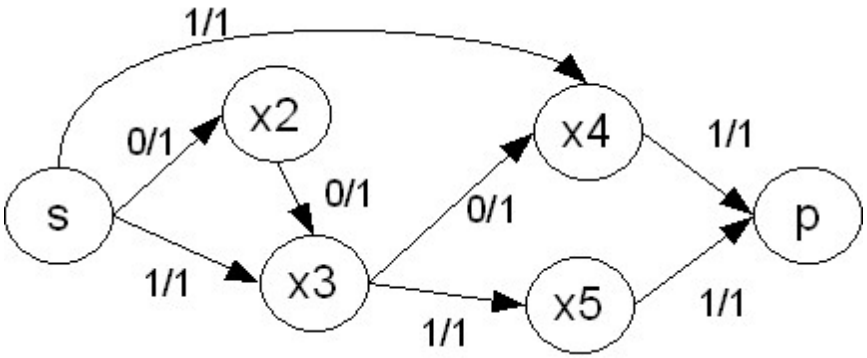
3.10. Applications des flots – Chemins disjoints (2)

- **Exemple – chemins arcs-disjoints**

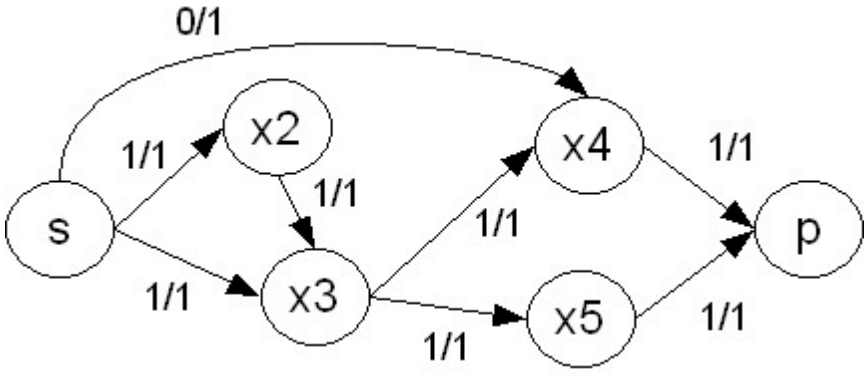




- Résolution :



ou

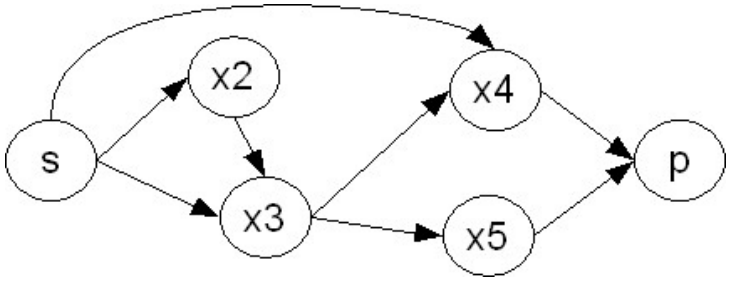


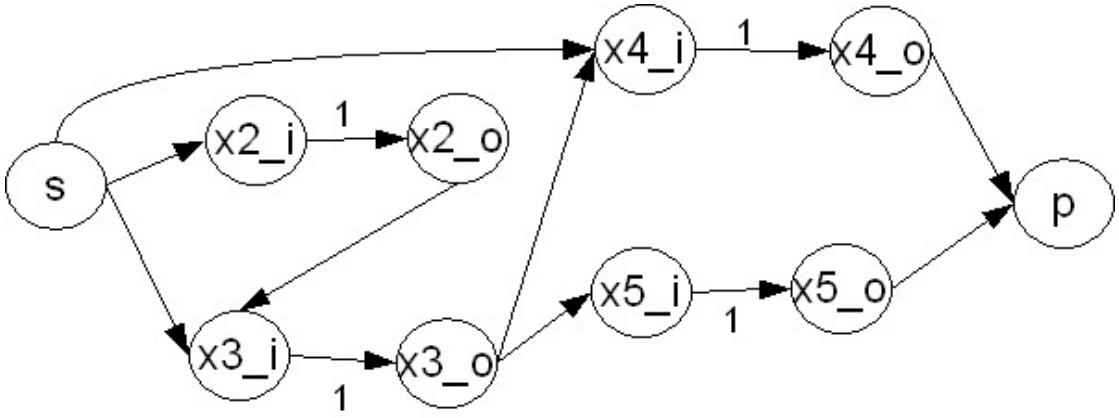
3.10. Applications des flots – Chemins disjoints (3)

- **Recherche de chemins sommets-disjoints**
  - Soit un graphe G(X, A)
    - On détermine le graphe de flot
      - Dédoubler les sommets :
        - Sommet x ➔ x_in et x_out
      - Arcs
        - 1 arc de x_in vers x_out : capacité 1
        - Si arc de x vers y dans le graphe initial : placer un arc de x_out vers y_in de capacité 1
      - Le débit total obtenu en cherchant le flot maximal = le nombre de chemins disjoints

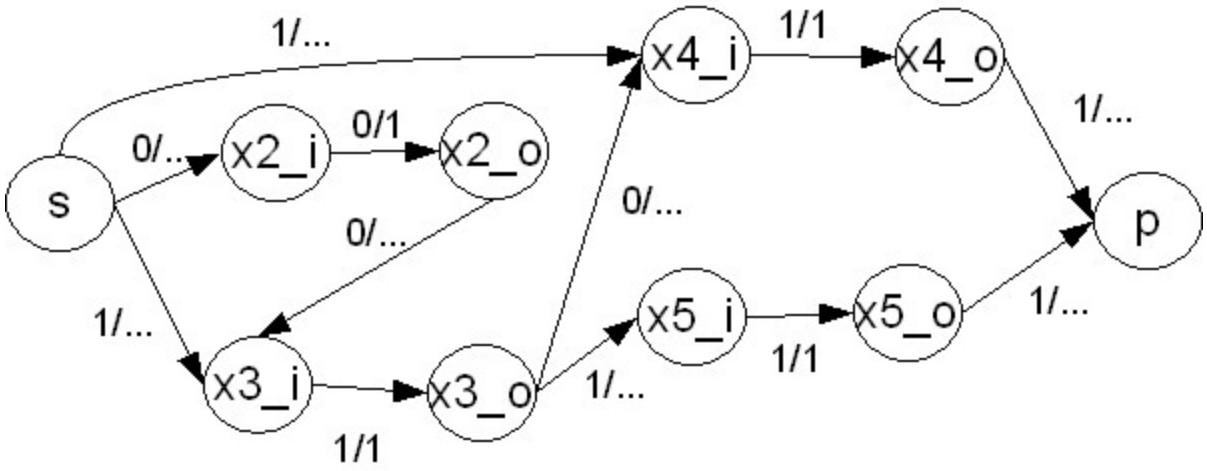
3.10. Applications des flots – Chemins disjoints (4)

- **Exemple**





- Résolution

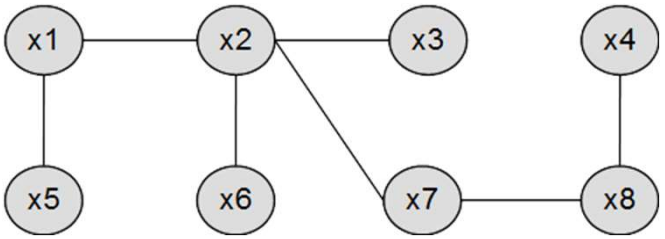


Plan

1. **Introduction**
2. **Parcours de Graphe**
3. **Optimisation et Graphes**
   - Plus courts chemins
   - Problèmes de flots
   - Arbre couvrant de cout minimal

3.11– Arbres Couvrant

- Rappel définition d'un arbre
  - Le graphe **non orienté** $G(X, A)$ **est un arbre**
    - G est connexe et sans cycle
    - G est (sans cycle)/(connexe) et comporte $(|X| - 1)$ arêtes
    - G est sans cycle et en ajoutant une arête, on crée un cycle élémentaire
    - G est connexe et en supprimant une arête, le graphe n'est plus connexe
    - Pour toute paire de sommets (x_i, x_j), il existe exactement une chaine les reliant

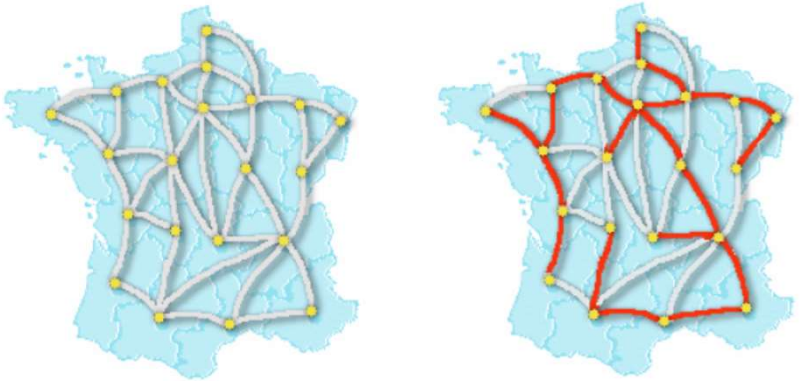


    - **Feuilles** : sommets de degré 1
    - **Nœuds internes** : sommet de degré > 1

3.11– Arbres Couvrant

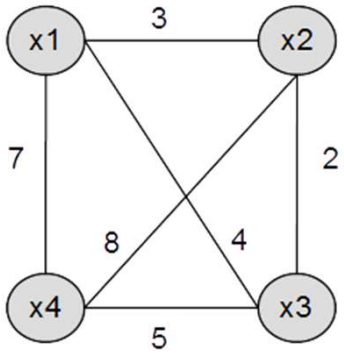
- Soit un graphe non orienté valué $\boldsymbol{G(X, A, C)}$
  - Un arbre couvrant de G est un graphe partiel $G'(X, A', C')$
    - Tous les sommets sont couverts par les arêtes sélectionnées
  - Arbre couvrant de cout minimal
    - La somme des couts des arêtes est minimale
  - Tout graphe connexe admet un arbre couvrant

  - **Exemple** :
    - Sélectionner les connexions à établir telles que toutes les villes soient connectées et le cout de connexion soit minimal

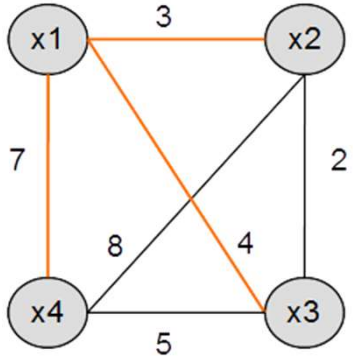


3.11– Arbres Couvrant

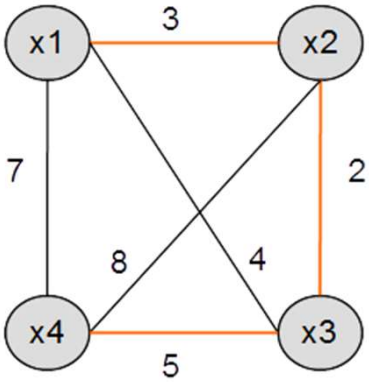
- Arbre couvrant de cout minimal et arbre des plus courts chemins
  - En général l'arbre couvrant de cout minimal est différent de l'arbre des plus courts chemins
  - **Exemple**



- Arbre des plus courts chemins
  - Cout = 14



- Arbre couvrant de cout minimal
  - Cout = 10



3.11– Algorithmes gloutons

- Deux algorithmes pour déterminer arbre couvrant de cout minimal
  - **Prim** :
    - arbre = graphe connexe avec un nombre minimal d'arêtes
    - Maintenir un graphe connexe à chaque itération en ajoutant une arêtes pour connecter la partie connexe aux sommets non encore couverts
  - **Kruskal** :
    - arbre = graphe acyclique avec un nombre maximum d'arêtes
    - Maintenir un graphe acyclique à chaque itération en ajoutant une arête ne créant pas de cycle et couvrant 1 ou 2 nouveaux sommets
  - **Algorithmes dits « gloutons »**
    - Construction progressive de la solution optimale
      - A chaque étape on ajoute une arête qui sera dans la solution optimale
      - Pas de remise en cause de la sélection

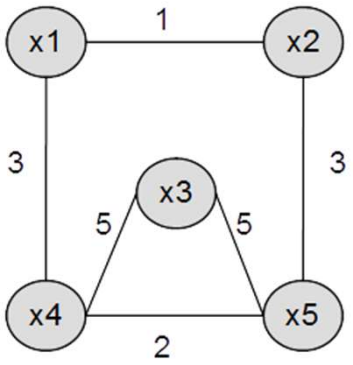
3.11– Algorithme de Prim

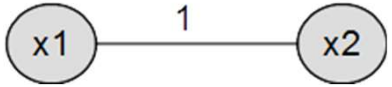
- Principe
  - Soit un graphe initial $G(X, A)$
  - Maintenir un graphe partiel connexe en ajoutant une arête connectant un nouveau sommet
  - Initialisation :
    - Un sommet ➔ graphe partiel connexe $G'(X', A')$ tel que
      - $X' = \{x_i\}$ et $A' = \emptyset$; Cout = 0
  - Itérations : on a un graphe partiel connexe $G'(X', A')$
    - Sélectionner l'arête $a = (x_j, x_k)$ de cout minimal telle que $x_j \in X'$ et $x_k \notin X'$
    - Nouveau graphe partiel connexe $G'\left(X' \cap \{x_k\}, A' \cap \{(x_j, x_k)\}\right)$
    - Cout = Cout actuel + cout(a)
  - Arrêt : tous les sommets sont connectés : $X' = X$

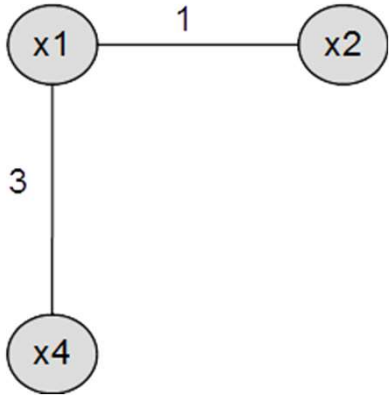
- Remarque :
  - choix du sommet initial : pas d'impact sur la valeur de l'arbre couvrant

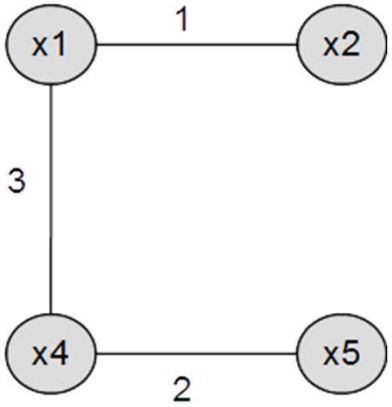
3.11– Algorithme de Prim

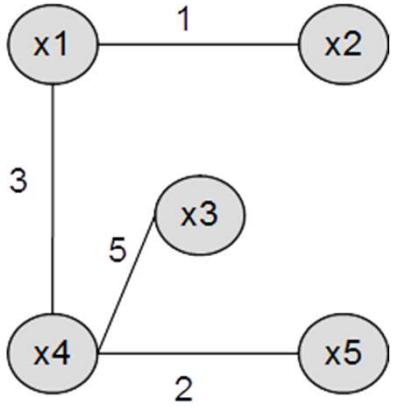
- Exemple











3.11– Algorithme de Prim

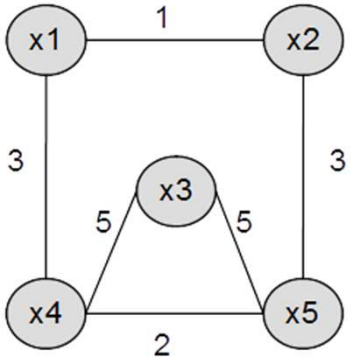
- Convergence
  - On détermine un arbre couvrant si le graphe initial est connexe ➔ arrêt lorsque tous les sommets sont « marqués »
- Optimalité
  - Admise
- Complexité
  - Nombre d'itérations : $n - 1$ + initialisation
  - A chaque étape : chercher l'arête minimale permettant d'ajouter un sommet parmi les arêtes « sortantes »
    - Structure pour mémoriser les voisins de chaque sommet dans l'ordre des couts
      - Sélectionner la meilleure arête ➔ nouveau sommet
      - Mise à jour des sommets voisins à celui sélectionné

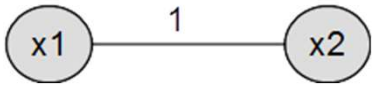
3.11– Algorithme de Kruskal

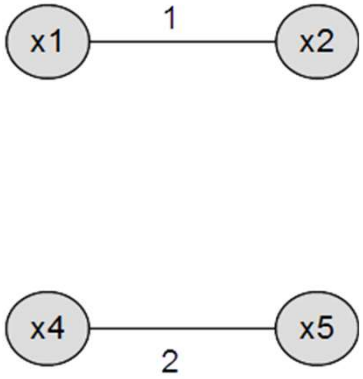
- Principe
  - Soit un graphe initial $G(X, A)$
  - Maintenir un graphe acyclique à chaque itération en ajoutant une arête ne créant pas de cycle
    - Graphe partiel = Forêt
  - Initialisation :
    - Trier les arêtes par cout croissant
    - $G'(X', A')$ tel que $X' = \emptyset$ et $A' = \emptyset$; Cout = 0
  - Itérations :
    - Sélectionner l'arête $a = (x_i, x_j)$ de cout minimal telle que G' acyclique
    - Nouveau graphe partiel connexe $G'\left(X' \cap \{x_i, x_j\}, A' \cap \{(x_i, x_j)\}\right)$
    - Cout = Cout actuel + cout(a)
  - Arrêt : sélection de $n - 1$ arêtes

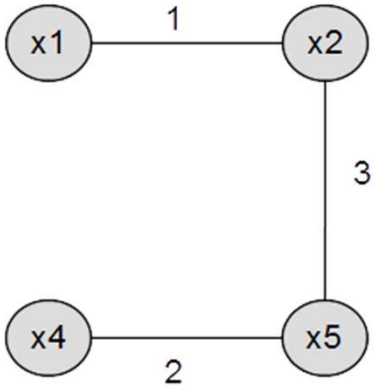
3.11– Algorithme de Kruskal

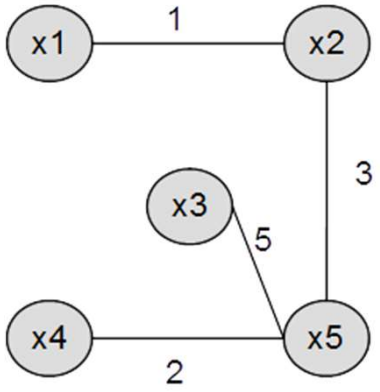
- Exemple











3.11– Algorithme de Kruskal

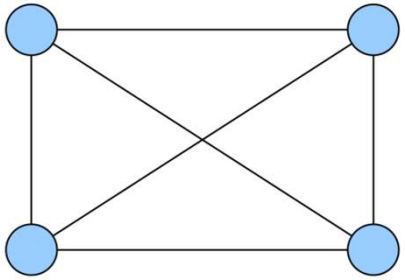
- Convergence
  - On détermine un arbre couvrant si le graphe initial est connexe ➔ arrêt le nombre requis d'arêtes est sélectionné
- Optimalité
  - Admise
- Complexité
  - Nombre d'itérations : $n-1$ + initialisation tri des arêtes : $O(m.\log(m))$
  - A chaque étape : chercher l'arête minimale ne créant pas de cycle
    - Maintenir les composantes connexes de la forêt en construction
    - Si l'arête sélectionné relie 2 sommets de la même composante elle crée un cycle
    - Fusionner les composantes connexes reliées par l'ajout d'une arête

Plan

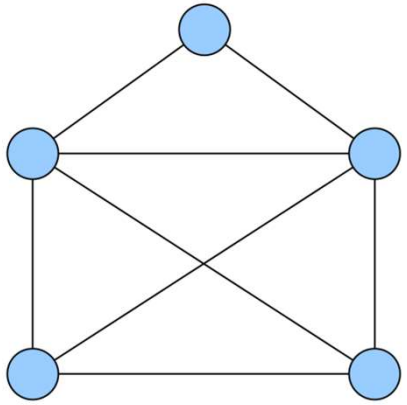
1. **Introduction**
2. **Parcours de Graphe**
3. **Optimisation et Graphes**
   - Plus courts chemins
   - Problèmes de flots
   - Arbre couvrant de cout minimal
   - Parcours Eulériens / Hamiltoniens / Voyageur de Commerce

3.12- Parcours Eulériens

- Parcours Eulériens
  - Passe une fois et une seule par chaque arête du graphe (arc)
    - Pas de notion d'optimisation
  - **Conditions d'existence** (cas non orienté)
    - Le graphe est connexe
    - 0 ou 2 sommets de degré impair
      - Si 0 sommet de degré impair : existence d'un cycle eulérien
      - Si 2 sommets x et y de degré impair : existence d'une chaine eulérienne entre eux



Non (tous les sommets de degré impair)



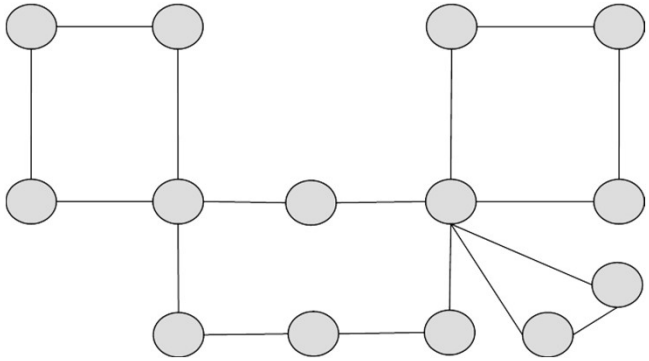
Oui – Chaine eulérienne
(2 sommets de degré impair)

3.12–Algorithme

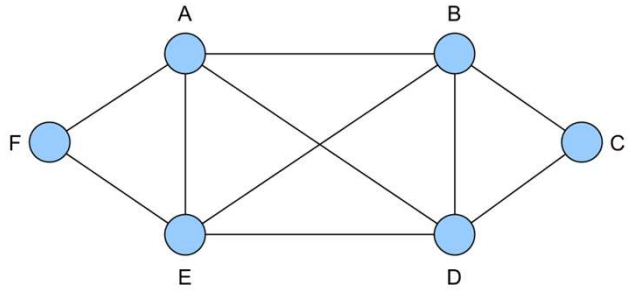
- Adaptation parcours
  - Choisir un sommet (aléatoire si tous degré pair) ou sommet degré impair
  - Construire un cycle à partir de ce sommet en toutes les arêtes du sommet
  - Eulerien ← cycle
  - Tant qu'il existe un sommet avec des arêtes non couvertes
    - Construction d'un cycle à partir de ce sommet
    - Insertion du cycle dans le trajet eulérien en cours

- **Complexité** : parcours (marquer les arêtes visitées)

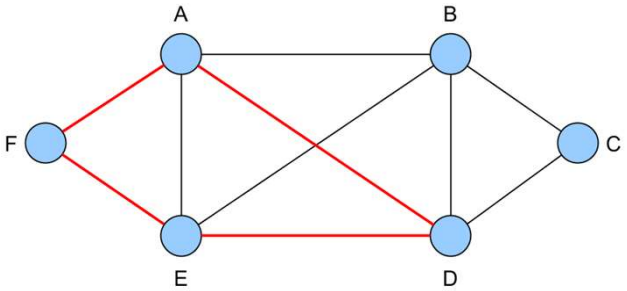
- **Exemple**
  - Tous les degrés sont pairs
  - Il existe un cycle eulérien

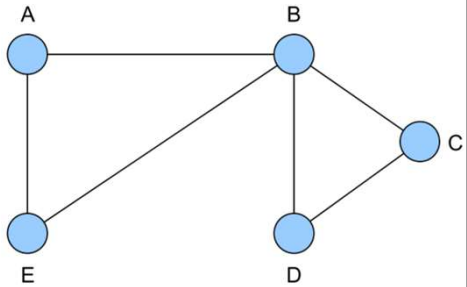


3.12–Déroulement algorithme

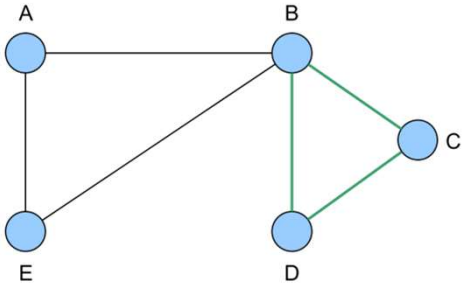


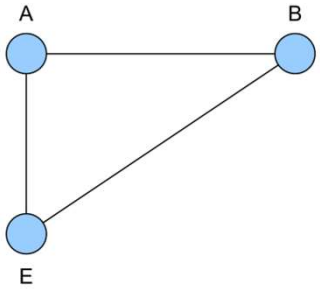
1. Construire un cycle : ADEFA et le « retirer »



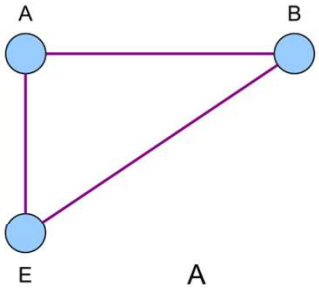


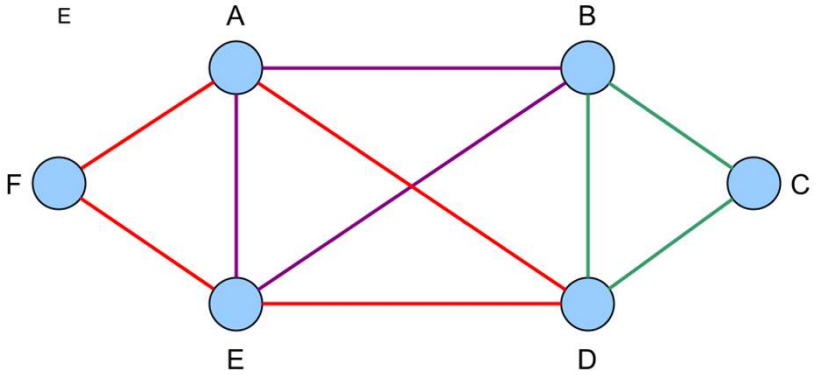
2. Construire un cycle : DCBD et le « retirer





3. Construire un cycle : EABE





Recombiner les cycles
(A, D, C, B, D, E, A, B, E, F, A)

- Retours sur les ponts de Koenigsberg

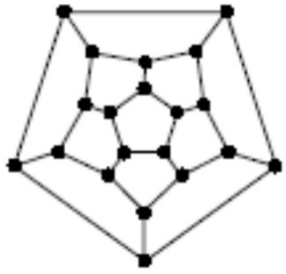
  - 4 sommets de degrés impairs ➔ pas de parcours eulérien

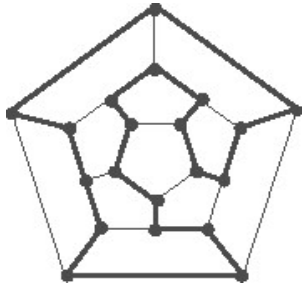
- **Variante** : Parcours chinois
  - Un parcours eulérien n'existe pas forcément … Relacher la contrainte d'unicité
  - Passe au minimum une fois par chaque arête / arc
- **Exemple** : distribution de courriers, déneigement de voies

3.12– Parcours hamiltoniens

- Parcours hamiltoniens (graphes non orientés)
  - Passe une fois et une seule par chaque sommet du graphe
  - Un graphe est hamiltonien ssi il contient un cycle hamiltonien
  - Déterminer si un graphe est hamiltonien est NP-Complet

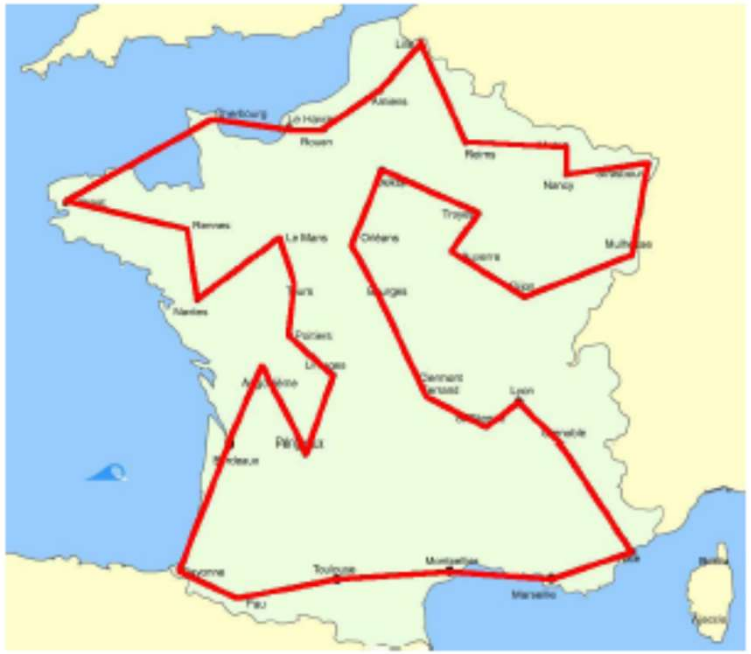
  - Conditions suffisantes d'existence
    - Si Graphe complet alors possible un cycle hamiltonien
    - Si possède un sommet de degré 2 alors ils sont dans le cycle hamiltonien
  - **Exemple** :





3.12– Voyageur de Commerce

- Problème
  - Graphe pondéré orienté ou non
  - Cout d'un cycle = somme des couts des arêtes/arcs
  - Déterminer un cycle (circuit) hamiltonien de cout minimal



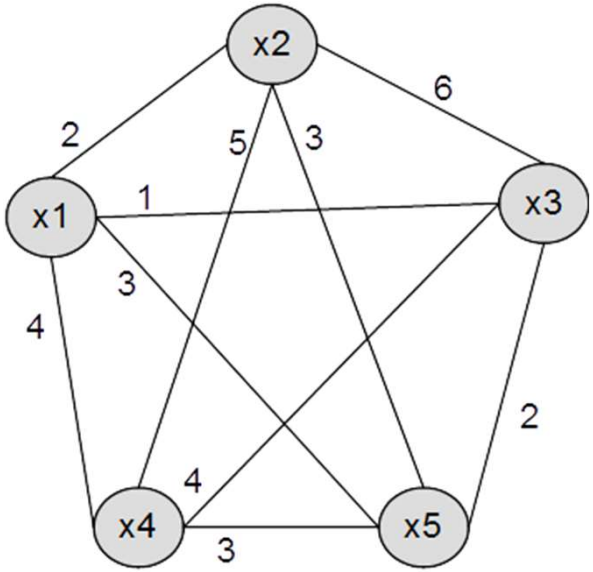
3.12– Voyageur de Commerce

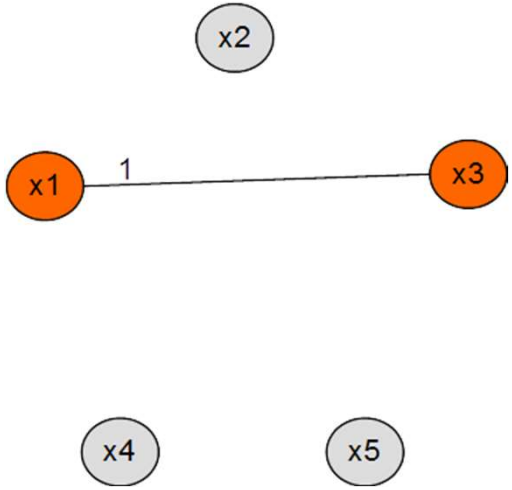
- Voyageur de Commerce : NP-Complet
  - Méthodes exactes
  - Méthodes approchées
    - Heuristique

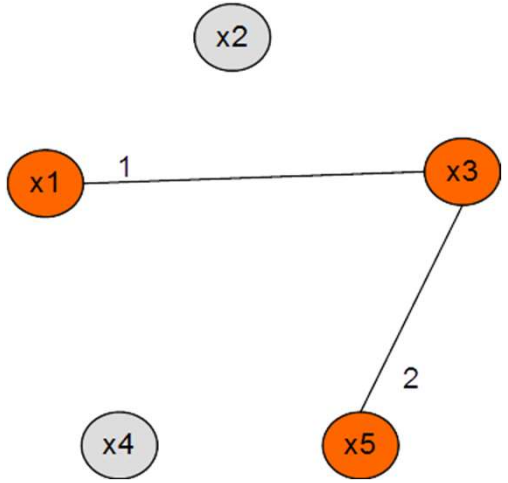
  - **Exemple** : **Heuristique (algorithme glouton)**
    - Sélectionner un sommet origine (ou sommet fixé) : x_i
    - Connecter ce sommet avec son plus proche voisin : x_k
    - Poursuivre à partir de x_k en suivant le même principe et sans repasser par un sommet déjà visité
    - Relier le dernier sommet au sommet origine

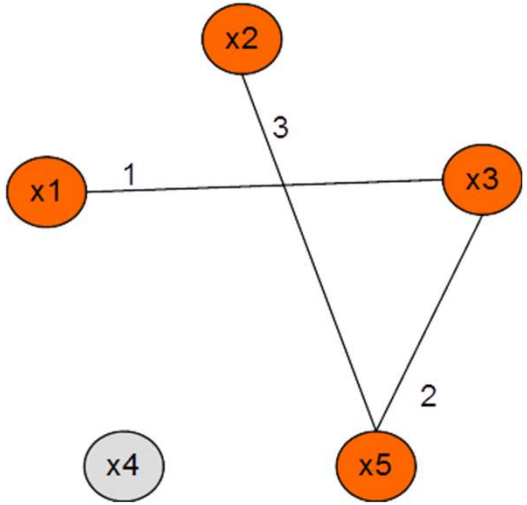
  - **Remarque** :
    - Si le graphe n'est pas complet ….

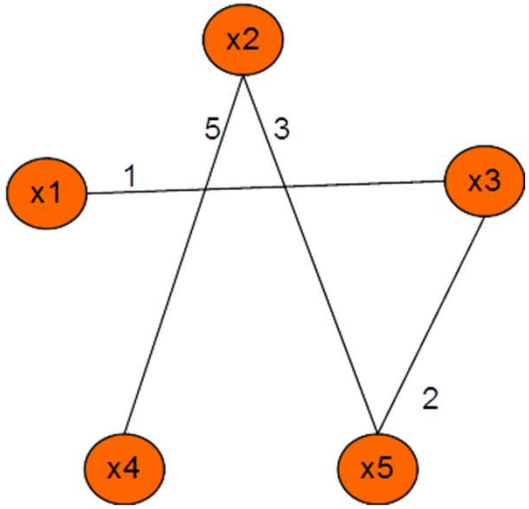
3.12– Déroulement Heuristique

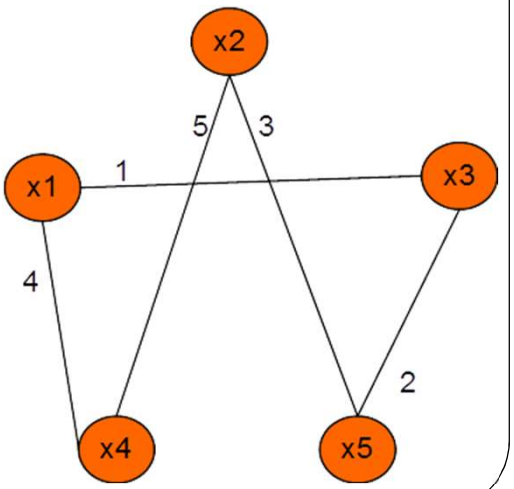






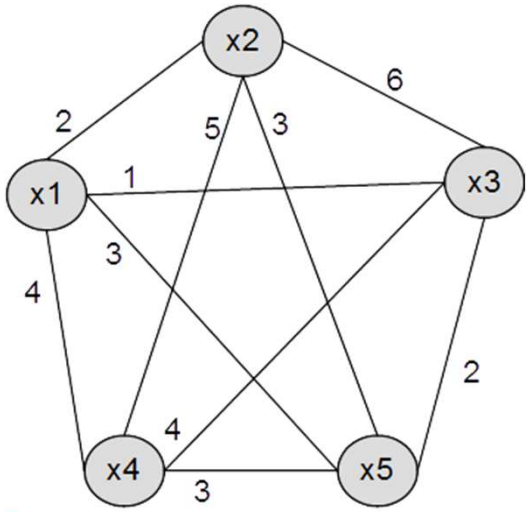


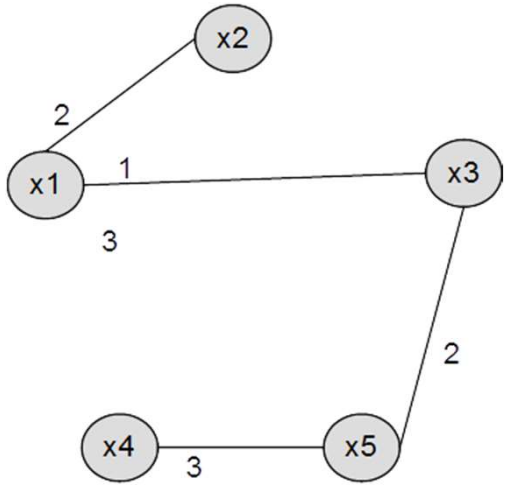


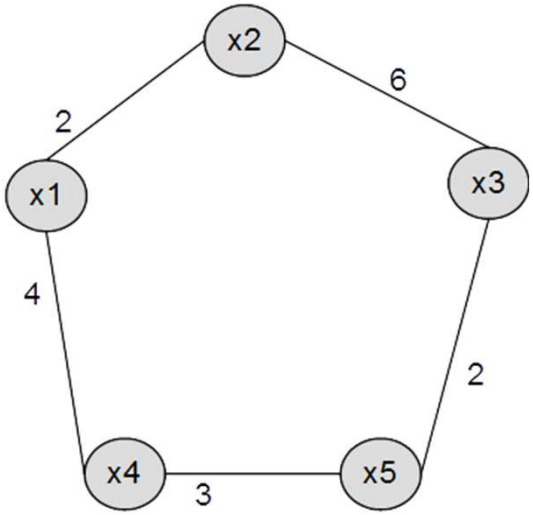


3.12– Voyageur de Commerce

- Autre heuristique
  - Déterminer un arbre couvrant de cout minimal du graphe initial
  - Parcourir en profondeur l'arbre couvrant et numéroter les sommets dans l'ordre d'entrée dans la pile
  - Créer une chaine en suivant la numérotation
  - Relier au sommet de départ







- Numérotation : x_1, x_2, x_3, x_5, x_4

Plan

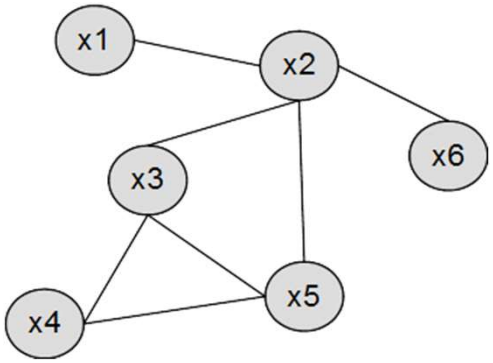
1. **Introduction**
2. **Parcours de Graphe**
3. **Optimisation et Graphes**
   - Plus courts chemins
   - Problèmes de flots
   - Arbres couvrants de cout minimal
   - Parcours Eulériens / Hamiltoniens / Voyageur de Commerce
   - Divers :
     - Stable Maximal / Maximal Independent Set
     - Transerval minimal / Minimal Vertex Cover
     - Clique Maximale / Maximal Clique
     - Coloration des sommets

3.13– Stable / Independent Set

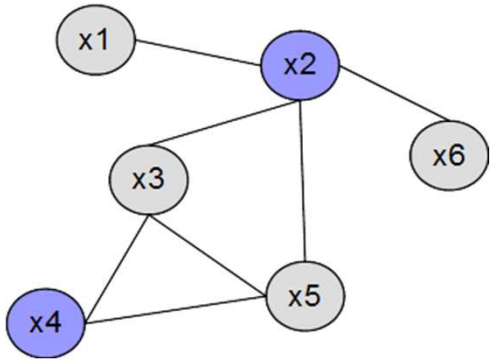
- Soit un graphe non orienté $G(X, A)$
  - **Stable / Independent Set**
    - Ensemble de sommets S non adjacents deux à deux
    - Cardinalité : nombre de sommets ➔ Stable maximal

NP-Complet /Difficile

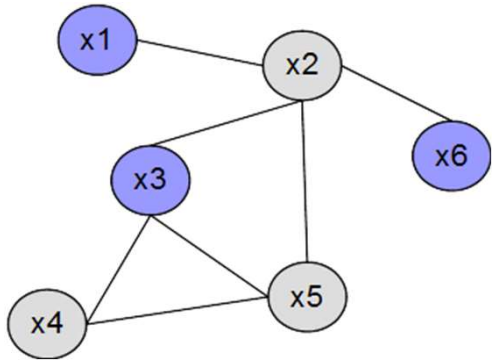
- **Exemple** :



o



stable cardinalité 2



stable cardinalité 3

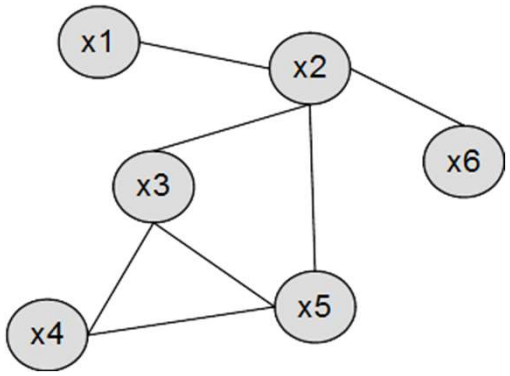
- Application : Graphe d'incompatibilité (parallélisation d'activités)

3.13– Transversal / Vertex Cover

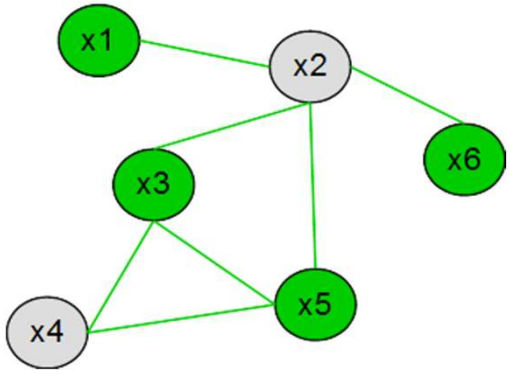
- Soit un graphe non orienté $G(X, A)$
  - **Transversal / Couverture par les sommets / Vertex Cover**
    - Ensemble de sommets T tel que toute arête à un sommet dans T
    - Cardinalité : nombre de sommets ➔ Minimal Vertex Cover

NP-Complet /Difficile

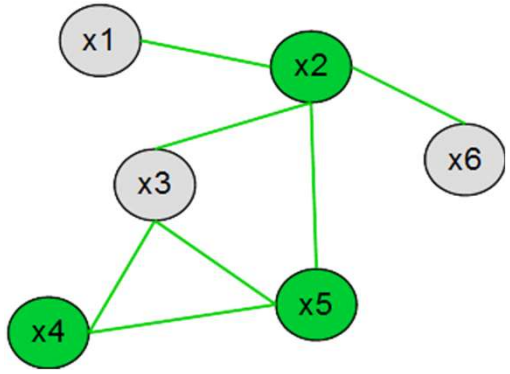
- **Exemple** :



o



Vertex cover taille 4

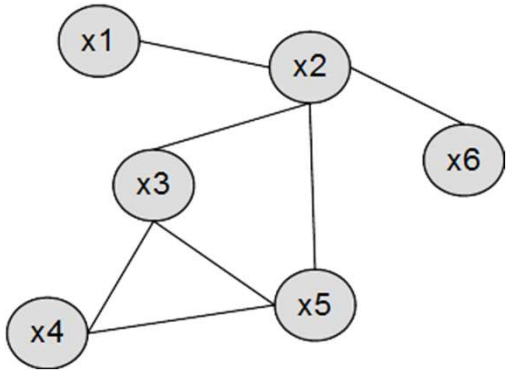


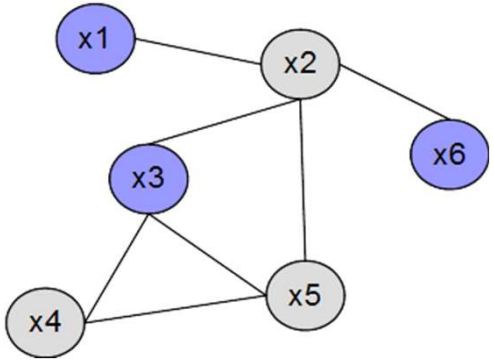
Vertex cover taille 3

- Application : surveillance dans un bâtiment (min nb caméras)

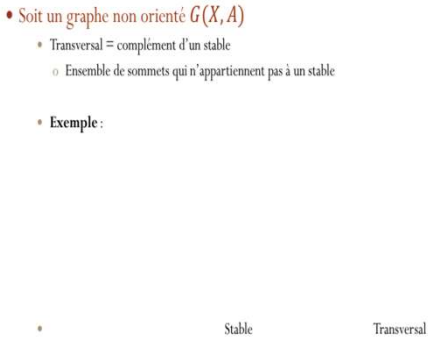
3.13– Stable et Transversal

- Soit un graphe non orienté $G(X, A)$
  - Transversal = complément d'un stable
    - Ensemble de sommets qui n'appartiennent pas à un stable
  - **Exemple** :





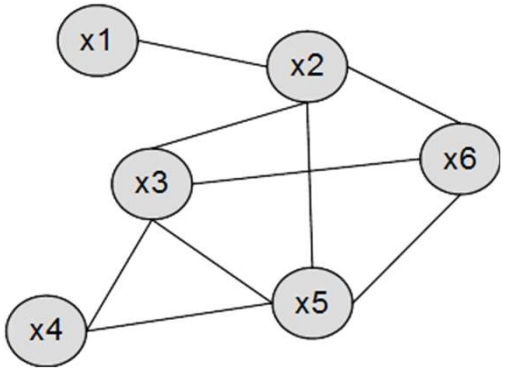
Stable

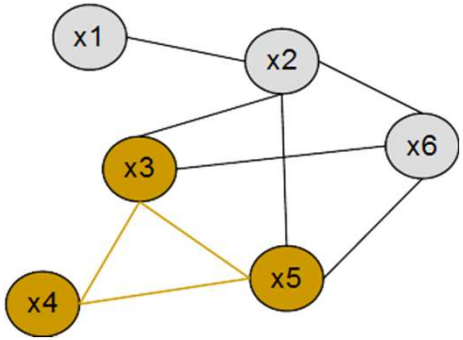


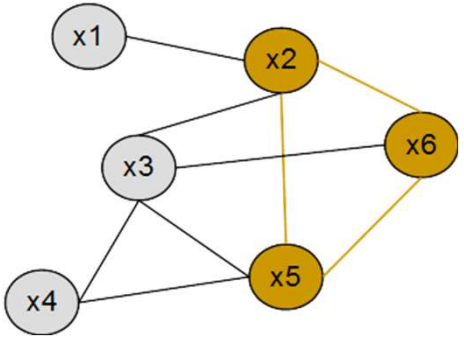
Transversal

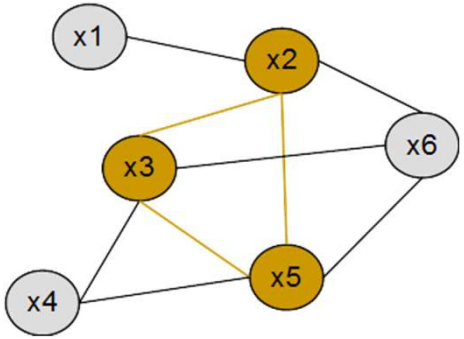
3.13– Clique

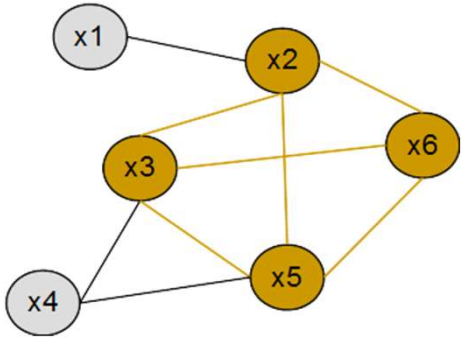
- Soit un graphe non orienté $G(X, A)$
  - **Clique**
    - ensemble de sommets C engendrant un sous-graphe complet
    - Cardinalité : nombre de sommets ➔ Maximal Clique **NP-Complet /Difficile**
  - **Exemple** :





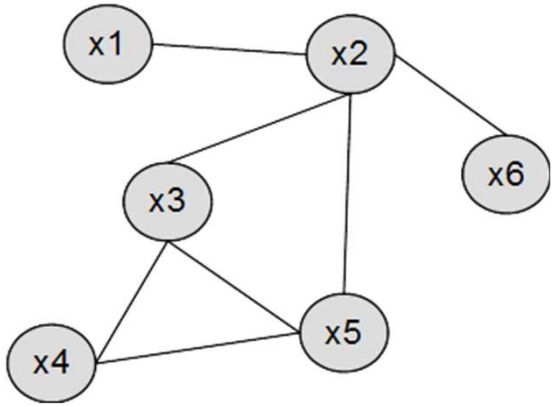


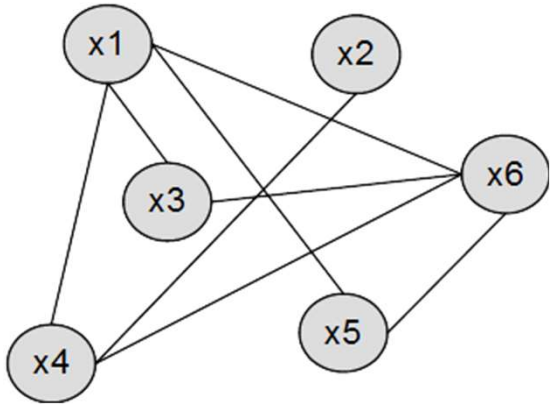


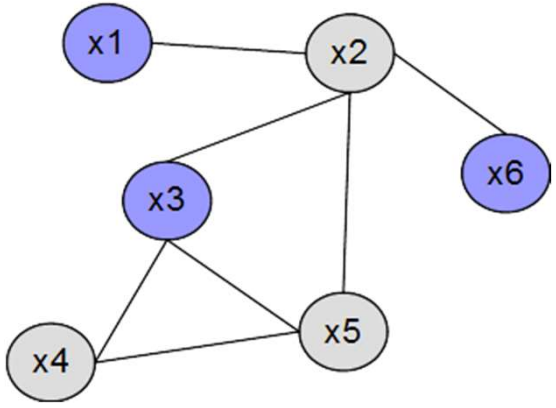


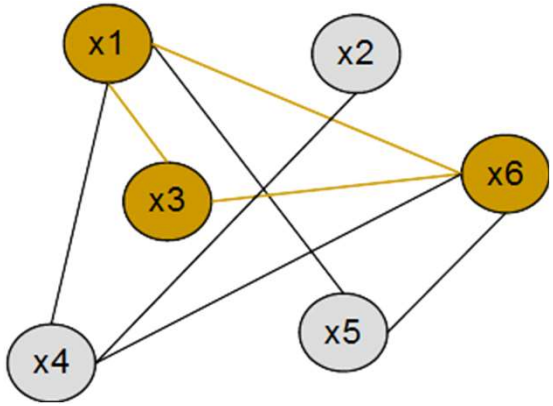
3.13– Stable et Clique

- **Stable maximal = Clique maximal dans le graphe complémentaire**



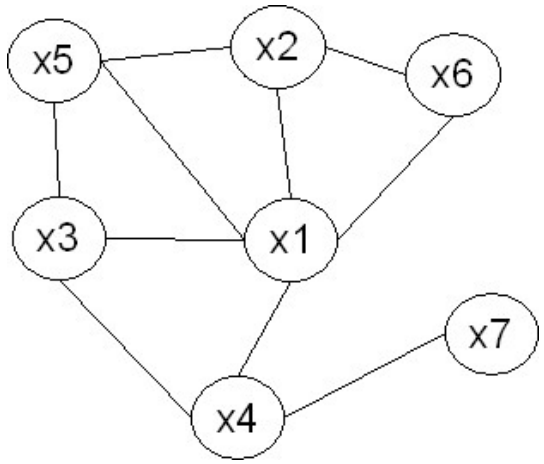


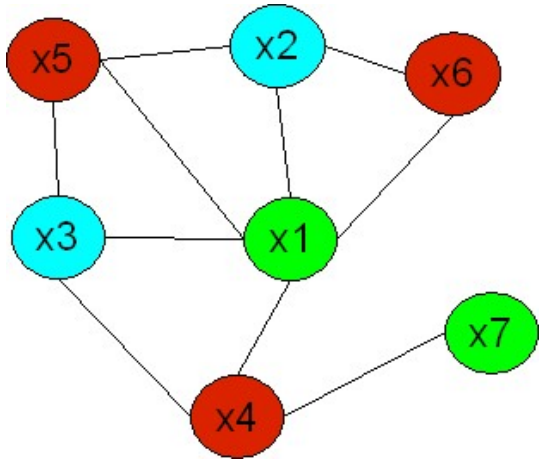




3.13– Coloration des sommets

- Soit un graphe non orienté $G(X, A)$
  - Déterminer une coloration des sommets telle que deux sommets adjacents n'aient pas la même couleur





- Nombre de couleurs utilisées : $\chi(G)$: nombre chromatique
- Minimiser le nombre de couleurs

NP-Complet /Difficile

3.13– Coloration des sommets

- Coloration, stable et clique
  - Stable = ensemble de sommets pouvant prendre la même couleur
    - Coloration = partition du graphe en stables
  - Clique = ensemble de sommets de couleur différentes
- Bornes
  - Supérieure : basée sur le degré max : $\chi(G) \leq \Delta + 1$
  - Inférieure :
    - basée sur taille clique maximale : $C_{max} \leq \chi(G)$
    - Basée sur taille stable maximal : $n \leq \chi(G) \times S_{max}$
  - **Applications** :
    - Emplois du temps :
      - sommets = cours; couleurs = créneaux, arêtes = incompatibilités
    - Allocation de fréquences
      - Sommets = antennes; couleurs = fréquences; arêtes = interférences

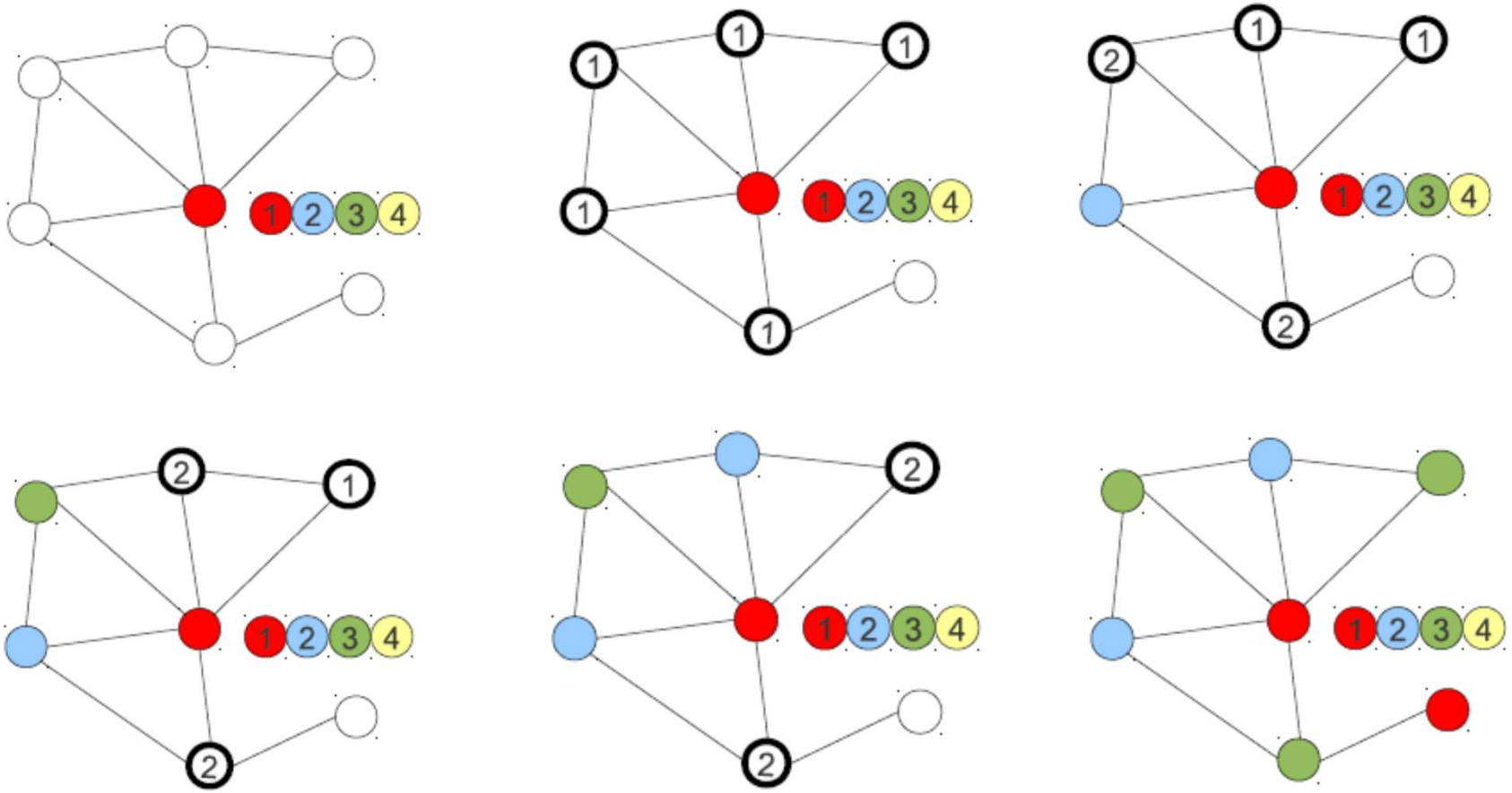
3.13– Coloration des sommets

- Heuristique
  - Ordre sur les sommets / Ensemble de couleurs
  - Allouer les couleurs en prenant les sommets dans l'ordre

  - **Exemple heuristique dynamique**
    - Ordre des sommets = nombre de voisins colorés décroissants
    - Prendre les couleurs dans un ordre donné fixe
  - Pour le premier sommet : allouer la première couleur possible
    - Ré-ordonner les sommets restant en fonction du nb de voisins colorés

3.13– Coloration des sommets

- **Exemple heuristique dynamique**



Conclusion

- **Outils de modélisation de problèmes**
  - Représentation informatique
- **Quelques méthodes de résolution**
  - Propriétés dans des graphes / Décision / Optimisation

- **Des problèmes de complexité polynomiale**
  - Parcours / Plus courts Chemins / Flot Max / Arbres Couvrants / Parcours Eulérien
- **Des problèmes NP-Complet**
  - Circuit hamiltonien, Voyageur de Commerce, Max Independent Set, Min Vertex Cover, Max Clique, Min color, ….
  - Heuristiques / Algorithmes gloutons

Conclusion

- **Fin du cours de Graphes mais …**
  - La suite :
    - BE Graphes (calcul d'itinéraires) : 3MIC
    - TP Programmation Fonctionnelle : 4IR-Info (Flot Max)
    - Cours d'IA : 4IR-Info
      - Jeux combinatoires
      - Métaheuristiques
    - Optimisation Combinatoire, Fouille de Données, Représentation de Connaissances, …
    - ……