## Types de graphes

- **Graphes non orientés**: sommets reliés par des arêtes bidirectionnelles
- **Graphes orientés**: sommets reliés par des arcs unidirectionnels
- **Graphe simple**: sans boucles ni arêtes multiples
- **Graphe vide**: aucune arête/arc
- **Graphe trivial**: un sommet, aucune arête
- **Graphe nul**: plusieurs sommets, aucune arête

## Propriétés des sommets

- **Degré**: nombre d'arêtes/arcs connectés à un sommet
- **Demi-degré extérieur**: nombre d'arcs sortants
- **Demi-degré intérieur**: nombre d'arcs entrants
- **Sommet pendant**: degré égal à 1

## Variantes des graphes

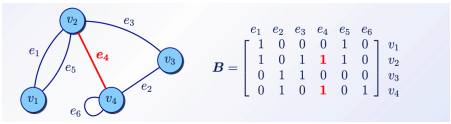
- **Graphe partiel**: tous les sommets, sous-ensemble des arêtes
- **Sous-graphe**: sous-ensemble des sommets avec leurs arêtes
- **Sous-graphe partiel**: sous-ensemble des sommets et des arêtes

## Structures dans les graphes

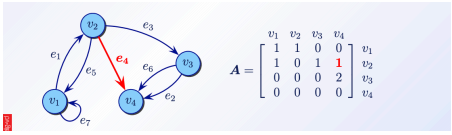
- **Chaîne**: succession de sommets et arêtes
- **Cycle**: chaîne fermée (même sommet au début et à la fin)
- **Chemin**: succession de sommets et arcs
- **Circuit**: chemin fermé
- **Forêt**: graphe sans cycles
- **Arbre**: forêt connexe

## Matrices

- **Matrices d'incidence**:
  - Représentent les relations sommets-arêtes
  - Non adaptées pour les boucles
  - Types différents pour graphes orientés/non orientés

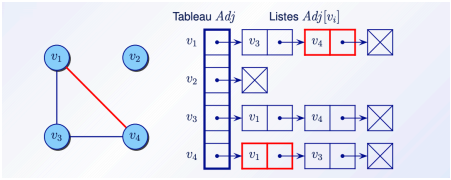



- **Matrices d'adjacence**:
  - Matrices carrées $n \times n$ indiquant l'adjacence des sommets
  - Symétriques pour graphes non orientés
  - Permettent de représenter tout type de graphe (avec boucles)

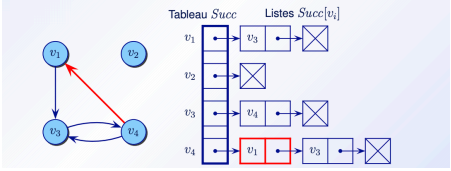



## Listes

- **Tableaux de listes d'adjacence**:
  - Stockent pour chaque sommet sa liste de voisins
  - Utilisés pour graphes non orientés




- **Tableaux de listes des successeurs**:
  - Pour graphes orientés
  - Chaque case contient la liste des successeurs



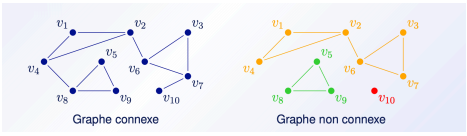

- **Tableaux compacts de successeurs**:
  - Composés de deux tableaux (TabSucc et tableau indexé)
  - Optimisation mémoire

## Comparaison

- **Matrices**: idéales pour graphes denses, test d'adjacence rapide
- **Listes**: recommandées pour graphes peu denses, économie mémoire
- **Structures spécifiques**: nécessaires pour cas particuliers

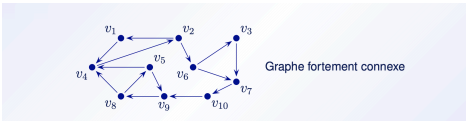
## Graphes connexes

- **Connexité**: existence d'une chaîne entre toute paire de sommets
- **Composante connexe**: sous-ensemble maximal de sommets reliés par des chaînes
- Chaque sommet appartient à une seule composante connexe



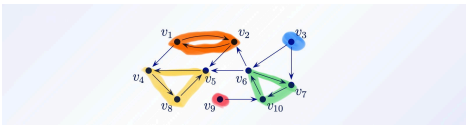

## Graphes fortement connexes

- **Fortement connexe**: existence d'un chemin orienté entre toute paire de sommets
- Critères: au moins 2 sommets, possibilité de circuit passant par tous les sommets



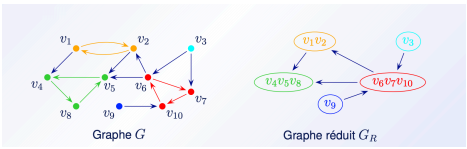

## Composantes fortement connexes

- Sous-ensembles maximaux de sommets avec chemins orientés entre chaque paire
- Un graphe peut contenir plusieurs composantes fortement connexes



## Graphes réduits

- Représentation où chaque composante fortement connexe devient un sommet
- Propriétés: graphe simple, sans circuit
- Un graphe est fortement connexe si son réduit est trivial




## Complexité et notation de Landau

- **Complexité**: ressources nécessaires (temps/espace) à la résolution d'un problème
- **Types**: pire cas, cas moyen, meilleur cas
- **Notations asymptotiques**:
  - O(g): croissance au plus aussi rapide que g
  - Ω(g): croissance au moins aussi rapide que g
  - Θ(g): même ordre de croissance que g

## BFS (exploration en largeur)

L'exploration en largeur est un algorithme permettant de parcourir un graphe.

- utilise une liste FIFO
- permet de calculer les plus courtes chaînes (ou chemin pour les graphes orientés) d'un sommet à tous les autres sommets de sa composante

### Idée

1. On commence par le sommet de départ
2. Tous les voisins directs sont ajoutés à la liste (souvent dans l'ordre alphabétique)
3. On traite successivement les sommets de la liste en y ajoutant à chaque fois les voisins directs non découverts
4. On continue jusqu'à ce que la liste soit vide

## DFS (exploration en profondeur)

L'exploration en profondeur parcourt un graphe en explorant chaque branche au maximum avant de revenir en arrière. Elle est généralement récursive, mais peut utiliser une liste LIFO pour les sommets à explorer dans une version non récursive.

### Idée

1. On commence par le sommet de départ
2. On traite le sommet le plus proche (si graphe pondéré) ou le sommet suivant dans l'ordre alphabétique
3. On continue jusqu'à ce que l'on ne puisse plus avancer
4. On revient en arrière et on traite le sommet suivant
5. On continue jusqu'à ce que la liste soit vide

### Complexité

$$O(n+m)$$

## Kosaraju

Permet de trouver les composantes fortement connexes d'un graphe orienté.

### Idée

1. Construire le graphe transposé (inversion des arcs)
2. Effectuer une exploration en profondeur du graphe transposé
3. Utiliser les dates de fin de traitement de l'expoloration pour ordonner les sommets dans l'ordre décroissant de leur date de fin
4. Effectuer une exploration du graphe initial en utilisant la liste précédente pour le choix des racines de chaque exploration
5. Les sommets de chacune des arborescences construitent lors de ce second parcours définissent les composantes fortement connexes du graphe initial

## Tarjan

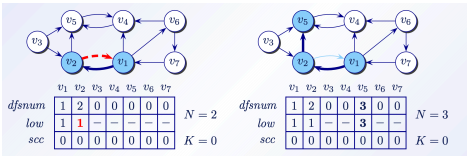
Permet de calculer les composantes fortement connexes d'un graphe orienté.

### Idée

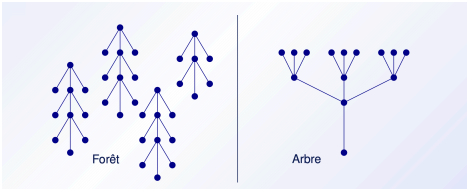
On utilise 3 tableaux :

- *dfsnum* : numéros de découverte des sommets
- *low* : stocke le plus petit numéro que l'on peut atteindre depuis $u$ **pour l'instant**
- *scc* : numéro de la composante fortement connexe du sommet $u$

1. Utiliser trois tableaux : *dfsnum*, *low*, et *scc*.
2. Numéroter les sommets au fur et à mesure de leur visite (*dfsnum*).
3. Stocker le plus petit numéro atteignable depuis chaque sommet (*low*).
4. Utiliser une pile pour suivre les sommets de la composante actuelle.
5. Mettre à jour *low* pour chaque sommet et ses voisins.
6. Lorsqu'un sommet est la racine d'une composante fortement connexe, attribuer un numéro de composante (*scc*) et retirer les sommets de la pile.
7. Répéter jusqu'à ce que tous les sommets soient visités.




## Arbre et forêts




### Propriétés

Les propriétés des arbres et forêts sont les suivantes :

- Un arbre est une forêt connexe.
- Chaque composante connexe d'une forêt est un arbre.

- Forêts et arbres sont des graphes simples ! En effet, toute boucle ou toute paire d'arêtes parallèles crée un cycle simple.
- Dans un arbre (ou une forêt) les sommets pendants (sommet de degré 1) sont souvent appelés des feuilles.
- Tout arbre comptant au moins 2 sommets possède au moins 2 feuilles.

## Arbres et forêts

### Recouvrant

Un arbre recouvrant d'un graphe non orienté connexe est un arbre qui contient tous les sommets du graphe initial. Cela est pareil pour une forêt recouvrante.

- En bleu: un arbre non recouvrant
- En vert: un arbre recouvrant
- En rouge: une forêt recouvrante

### Arbre recouvrant minimal

Un arbre recouvrant minimal d'un graphe non orienté connexe est un arbre recouvrant de poids minimal. L'objectif est de trouver un arbe permettant de relier tous les sommets du graphe avec le plus petit poids possible.

### Algorithme de Kruskal (1956)

#### Application

Pour appliquer cet algorithme, il faut:
- construire un tableau contenant toutes les arêtes du graphe avec leurs poids respectifs
- trier ce tableau par ordre croissant de poids
- sélectionner les plus petites arêtes du tableau pour relier tous les sommets du graphe

#### Complexité

$$O(m \log n + mn) = (mn)$$

### Algorithme de Prim (1957)

#### Application

Pour appliquer cet algorithme, il faut:
- construire un tableau ayant comme colonne
  - le n°d'itération
  - le sommet en cours de traitement
  - tous les sommets du graphe
- on définit des couples (distance, sommet) en commançant par l'itération 0
- on choisit le sommet le plus proche de l'arbre actuel puis mets à jour le tableau

#### Complexité

En fonction de la structure de données utilisée pour représenter le graphe, la complexité de l'algorithme de Prim est:

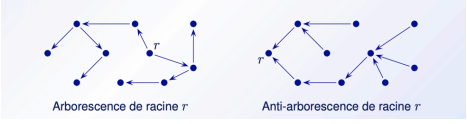
| | **Tableau** | **Tas binaire** | **Tas de Fibonacci** |
|---|---|---|---|
| Prim | $O(n^2)$ | $O(m \log n)$ | $O(m + n \log n)$ |

## Arborescences et racines

### Définition

Une arborescence de racine r est un arbre **orienté** dans lequel il existe un chemin du sommet r vers chacun des autres sommets de l'arbre.

De manière similaire, nous définissons une anti-arborescence de racine r comme un arbre orienté dans lequel il existe un chemin allant de chaque sommet vers le sommet r.




### Arborescence recouvrantes

Une arborescence recouvrante d'un graphe orienté est un arbre recouvrant de $G$ qui est une arborescence.

- Un graphe doit être connexe pour admettre une arborescence recouvrante
- Un graphe $G$ peut posséder des arborescences recouvrantes sans être fortement connexe mais…
- Un graphe $G$ est fortement connexe si et seulement s'il possède des arborescences recouvrantes **quel que soit le sommet racine choisi**.

### Arborescence de poids minimum

Une arborescence de poids minimum est une arborescence recouvrante d'un graphe orienté dont le poids est minimal.

### Chu-Liu

Si anti-arborescence on doit inverser les arcs de l'arborescence recouvrante pour obtenir une anti-arborescence.

#### Idée

1. Identifier les circuits du graphe en prenant pour chaque sommet le plus petit arc entrant
2. Si on trouve un circuit, les regrouper en un seul sommet
3. Tracer les arcs en partant du principe qu'un noeud ne peut avoir qu'un seul arc entrant
   - Pour cela il faut déduire du cout de l'arc entrant dans la composante fortement connexe le poids de l'arc interne à la entrant sur le sommet choisi **uniquement si on entre dans un circuit**
4. Répéter jusqu'à ce qu'il n'y ait plus de circuits
5. Regonfler le graphe au fur et à mesure en gardant uniquement les arcs nécessaires

## Plus courts chemins dans les réseaux

Dans des réseaux, il se peut que l'on trouve des circuits à coût négaitf. Par cela on entend qu'il existe un circuit qui permet de diminuer le coût d'un chemin. On parle aussi de circuit absorbant.S

### Algorithme de Belleman-Ford

L'algorithme de Bellman-Ford est un algorithme de recherche d'un plus court chemin dans un graphe orienté. Il est capable de gérer les circuits absorbants.

#### Idée

1. Initialisation: La distance de la source à elle-même est 0, et la distance vers tous les autres sommets est considérée comme infinie au départ
2. Relaxation des arêtes: Pour chaque arête du graphe, vérifier si on peut améliorer le chemin connu vers sa destination en passant par cette arête
3. Répéter cette vérification (n-1) fois, où n est le nombre de sommets, pour garantir que tous les plus courts chemins sont trouvés
4. Détection des circuits problématiques: Si après toutes ces vérifications on peut encore améliorer un chemin, cela signifie qu'il existe un circuit de poids négatif dans le graphe

| Itér. $k$ | Arc testé | Couples $(\lambda_j, p[j])$ pour chaque sommet 1 | 2 | 3 | 4 | Valeur de *Continuer* |
|---|---|---|---|---|---|---|
| Valeurs de départ | | (0,—) | ($\infty$,—) | ($\infty$,—) | ($\infty$,—) | faux |
| 1 | (1,2) | | (3,1) | | | vrai |
| | (1,4) | | | | (4,1) | |
| | (2,1) | | | | | |
| | (2,3) | | | (7,2) | | |
| | (3,4) | | | | | |
| | (4,2) | | (1,4) | | | |
| | (4,3) | | | (6,4) | | |
| Valeurs de départ | | (0,—) | (1,4) | (6,4) | (4,1) | faux |
| 2 | (2,3) | | | (5,2) | | vrai |
| Valeurs de départ | | (0,—) | (1,4) | (5,2) | (4,1) | faux |
| 3 | Aucune amélioration pendant l'itération | | | | | faux |

### Algorithme de Dijkstra

L'algorithme de Dijkstra se rapproche de l'algorithme de Prim il existe cependant une différence majeure:
- **Prim** coût depuis le parent direct
- **Dijkstra** coût depuis la source en comprenant tous les sommets intermédiaires qui sont déjà dans l'arborescence actuelle

#### Complexité

La similitude entre les algorithmes de Prim et de Dijkstra s'étend également à la complexité des deux algorithmes qui est la même et dépend, rappelons-le, de la structure utilisée pour stocker et gérer la liste L.
- La complexité de l'algorithme de Dijkstra est en $O(n^2)$ si L est gérée à l'aide d'un tableau contenant les priorités λ, les prédécesseurs immédiats p et une marque précisant si un sommet est encore dans L ou non.
- Elle est en $O(m \log n)$ si L est une queue de priorité simple (un tas binaire).
- Elle est en $O(m + n \log n)$ si L est gérée à l'aide d'un tas de Fibonacci.

La complexité spatiale additionnelle est égale à l'espace nécessaire pour stocker la liste L et les différentes marques. Elle est donc en $O(n)$.

### Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall se construit avec deux matrices:
- $W$ qui contient les poids des arcs et $\infty$ si aucune relation n'éxiste entre deux sommets
- $P$ qui contient les prédécesseurs immédiats de chaque sommet

Si une valeur de la diagonale de $W$ est négative alors il y a circuit négatif, on arrête l'itération.

#### Idée

1. Initialisation: On initialise la matrice $W$ avec les poids des arcs et la matrice $P$ avec les prédécesseurs immédiats
2. On fixe la k-ième ligne et colonne de la matrice avec le numéro du sommet puis on aditionne la valeur de ligne et colonne et si elle est plus petite que la valeur dans la matrice, on la mets à jour.
3. On répète cette opération pour tous les sommets

#### Complexité

La complexité de l'algorithme de Floyd-Warshall est en $O(n^3)$, où n est le nombre de sommets du graphe. Cette complexité est due à la nécessité d'examiner chaque paire de sommets pour chaque sommet intermédiaire.

### Algorithme de Johnson

L'algorithme de Johnson permet le calcul de tous les plus courts chemins dans un réseau, même en présence d'arcs de poids négatif, tout en étant plus efficace que les méthodes de Floyd-Warshall ou Dantzig pour les graphes peu denses.

#### Idée

1. Ajouter un sommet auxiliaire 0 relié à tous les autres sommets par des arcs de poids nul
2. Appliquer l'algorithme de Bellman-Ford depuis ce sommet auxiliaire pour:
   - Détecter l'existence d'un circuit à coût négatif (auquel cas l'algorithme s'arrête)
   - Calculer une fonction potentiel δ pour chaque sommet
3. Utiliser cette fonction potentiel pour recalculer les poids des arcs avec une formule qui garantit:
   - Que tous les nouveaux poids sont non négatifs
   - Que les plus courts chemins sont préservés
   - $c'ij = cij - (\delta j - \delta i) = cij + \delta i - \delta j$
   - poids + départ - arrivée
4. Appliquer l'algorithme de Dijkstra sur ce graphe repondéré pour chaque sommet source
5. Corriger les distances obtenues pour retrouver les distances réelles dans le graphe d'origine

#### Complexité

- Construction du réseau auxiliaire: $O(n)$
- Calcul des potentiels avec Bellman-Ford: $O(mn)$
- Calcul des nouveaux poids: $O(m)$
- Calcul des plus courts chemins avec Dijkstra: $O(mn + n^2 \log n)$
- Correction des distances: $O(n^2)$
- Complexité totale: $O(mn + n^2 \log n)$

#### Avantages

- Particulièrement efficace pour les graphes peu denses ($m \in O(n)$) avec une complexité de $O(n^2 \log n)$
- Capable de gérer les arcs de poids négatif (tant qu'il n'y a pas de circuit absorbant)
- Combine les avantages de Bellman-Ford (pour gérer les arcs négatifs) et de Dijkstra (pour l'efficacité)

## Graphes sans circuits et applications

Les graphes sans circuits (**DAG**) sont essentiels en gestion de projets, ordonnancement et compilation.

### Propriétés fondamentales

- Tout graphe fini sans circuits possède au moins un sommet sans prédécesseurs et un sans successeurs

- Tout sous-graphe partiel d'un graphe sans circuits est sans circuits
- Permettent d'introduire une notion de **rang** : $\text{rang}(u) < \text{rang}(v)$

### Tri topologique (Kahn) - $O(n+m)$

- **But** : Ordonner les sommets en respectant les relations d'ordre (indispensable pour ordonnancement)
- **Principe** : Répéter jusqu'à épuisement des sommets
  - Choisir un sommet sans prédécesseurs, le numéroter dans l'ordre croissant, le supprimer du graphe
- **Propriété** : Si le graphe contient un cycle, l'algorithme s'arrête avant d'avoir numéroté tous les sommets

## Plus court/long chemin - Équation de Bellman

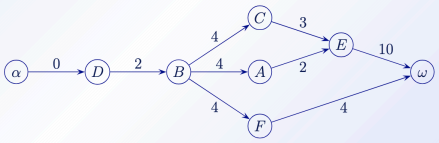
- **Applications** : Ordonnancement projets, optimisation, planification
- **Avantage DAG** : Traitement dans l'ordre topologique, pas d'itérations multiples comme Bellman-Ford classique
- **Plus court chemin** :
  - $\lambda_j = \min_{i \in \text{ Pred}[j]}(\lambda_i + c_{ij})$ avec $\lambda_s = 0$
- **Plus long chemin** :
  - $\lambda_j = \max_{i \in \text{ Pred}[j]}(\lambda_i + c_{ij})$ avec $\lambda_s = 0$
- **Algorithme** : Traiter sommets dans ordre topologique, appliquer équation
- **Complexité** : $O(n+m)$ (une seule passe suffit grâce au DAG)

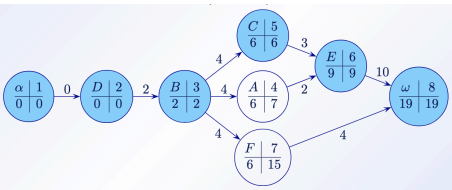
# Graphes potentiels-tâches

## Modélisation de projets

- **Sommets** : tâches du projet
- **Arcs** : contraintes de précédence (i précède j)
- **Poids** : durée $d_i$ de la tâche i
- **Ajouts** : sommet début $a$ et fin $w$ (poids 0)

| Tâche | Description | Durée (sem.) | Antériorités |
|---|---|---|---|
| D | Choix des stations | 2 | — |
| B | Accord administratif | 4 | D |
| C | Commande des décodeurs | 3 | B |
| A | Installation des antennes | 2 | B |
| E | Installation des décodeurs | 10 | C,A |
| F | Modification de la facturation | 4 | B |




### Méthode du chemin critique

- **But** : Identifier les tâches critiques dont tout retard retarde le projet entier
- **Applications** : Gestion de projets, planification industrielle, optimisation

**Phase 1 - Calcul dates au plus tôt** (forward pass) :
- $t_a = 0$ (début projet)
- $t_j = \max_{i \in \text{ Pred}[j]}(t_i + d_i)$ pour chaque tâche j
- Traitement dans l'ordre topologique

**Phase 2 - Calcul dates au plus tard** (backward pass) :
- $T_w = t_w$ (durée minimale projet)
- $T_i = \min_{j \in \text{ Succ}[i]}(T_j) - d_i$ pour chaque tâche i
- Traitement dans l'ordre topologique inverse

**Résultats** :
- **Tâche critique** : $t_i = T_i$ (marge libre nulle)
- **Chemin critique** : Succession de tâches critiques de début à fin
- **Durée projet** : $t_w$ (date plus tôt de fin)
- **Marge libre tâche i** : $T_i - t_i$ (retard possible sans impact)

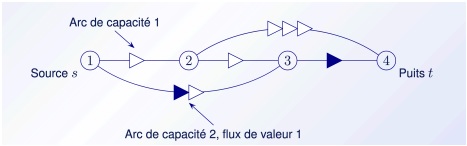
## Composition d'un nœud

| Nom des tâches | Numéros topologiques |
|---|---|
| Date de début au plus tôt | Date de début au plus tard |

# Flots dans un réseau

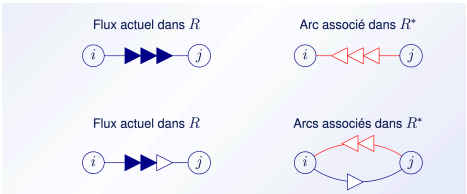
## Concepts fondamentaux

- **Réseau** : $R = (V, E, c, u)$ avec capacités $u_{ij}$ et coûts $c_{ij}$
- **Flot compatible** : Respecte capacités et conservation
- **Loi de conservation** : $\sum_{\text{entrant}} = \sum_{\text{sortant}}$ (sauf source/puits)




## Réseau d'augmentation

**Principe** : Construire graphe permettant d'augmenter le flot
- **Arcs directs** : $(i,j)$ si $x_{ij} < u_{ij}$, capacité résiduelle = $u_{ij} - x_{ij}$
- **Arcs inverses** : $(j,i)$ si $x_{ij} > 0$, capacité = $x_{ij}$ (annuler flot)




## Algorithmes de flot maximum

### Ford-Fulkerson - $O(mf*)$

- **But** : Trouver flot de valeur maximale de source s vers puits t
- **Applications** : Réseau transport, affectation ressources, couplage
- **Principe général** :

1. Partir d'un flot initial (souvent flot nul)
2. Construire réseau d'augmentation du flot actuel
3. Chercher chemin augmentant de s à t (DFS par exemple)
4. Si chemin existe : augmenter flot et retour étape 2
5. Si aucun chemin : flot actuel est optimal

- **Terminaison** : Algorithme se termine quand aucun chemin augmentant
- **Complexité** : $O(mf*)$ où $f*$ = valeur flot maximum (non polynomial)

### Edmonds-Karp - $O(m^2 n)$

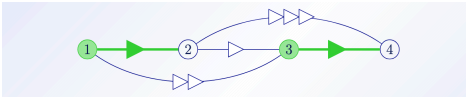
- **Amélioration de Ford-Fulkerson** : Choix du chemin augmentant
- **Stratégie** : Choisir plus court chemin (nombre d'arcs) via BFS
- **Avantages** :
- Complexité polynomiale garantie
- Évite cas pathologiques de Ford-Fulkerson
- Plus efficace en pratique sur graphes denses

## Coupe et théorème max-flow min-cut

**Coupe** $(S,T)$ : Partition de V avec $s \in S, t \in T$
**Capacité coupe** :

$$\sum_{(i,j):i \in S, j \in T} u_{ij}$$

**Théorème Ford-Fulkerson** : Valeur flot max = capacité coupe min




# Flot maximum à coût minimum

### Algorithme de Busacker-Gowen

- **But** : Flot de valeur maximale avec coût total minimal
- **Principe** : À chaque itération, saturer le plus court chemin (coût) dans réseau d'augmentation
- **Problème** : Arcs inverses ont coûts négatifs → impossibilité d'utiliser Dijkstra

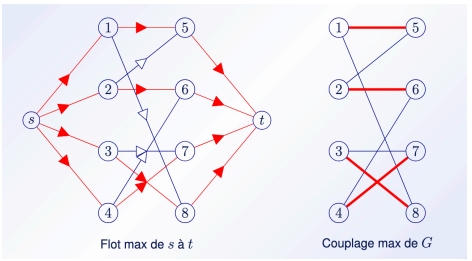
### Fonction de potentiel (Edmonds-Karp)

- **Solution** : Transformer les coûts pour éliminer les valeurs négatives
- **Potentiel** : $\lambda_i$ = distance depuis s dans réseau actuel
- **Coût réduit** : $c'_{ij} = c_{ij} + \lambda_i - \lambda_j$
- **Condition** : Réseau de base sans circuits de coût négatif


# Applications des flots

## Couplage maximum dans un graphe biparti

**Transformation** :
1. Orienter arêtes $A \to B$ (capacité 1)
2. Ajouter source s reliée à A (capacité 1)
3. Ajouter puits t relié depuis B (capacité 1)
4. Flot max = taille couplage max




## Problème d'affectation linéaire

- **Contexte** : n personnes, n tâches, coût $c_{ij}$ pour personne i sur tâche j
- **Objectif** : Affecter chaque personne à une tâche (coût minimum)
- **Méthode** : Couplage parfait de coût minimum → flot max-coût min

## Problème de transbordement

**Modélisation** : Réseau $R = (V, E, c, u)$
- **Sources** : offre $b_i < 0$
- **Puits** : demande $b_i > 0$
- **Transit** : $b_i = 0$

**Équation conservation** :

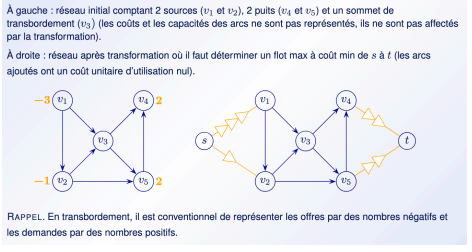
$$\sum_{j \in \text{ Pred}(i)} x_{ji} - \sum_{j \in \text{ Succ}(i)} x_{ij} = b_i$$

**Condition équilibre** :

$$\sum_{i \in V} b_i = 0$$

**Transformation en flot max-coût min** :
1. Source artificielle s → sources (coût 0, capacité = |offre|)
2. Puits → puits artificiel t (coût 0, capacité = demande)

**Cas particuliers** :
- **Transport** : graphe biparti complet (sources vers puits)
- **Affectation** : transport avec offres = demandes = 1

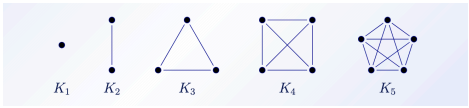



# Autres types de graphes

**Graphe complet** $K_n$ : Graphe simple où toute paire sommets distincts reliée
- Nombre arêtes : $\binom{n}{2} = \frac{n(n-1)}{2}$
- Tous sommets ont degré $n-1$

- Exemple : $K_4$ a 6 arêtes, $K_5$ a 10 arêtes

**Graphe complémentaire** $\overline{G}$ de $G = (V, E)$ :
- Mêmes sommets que G
- Arêtes = toutes arêtes possibles non présentes dans G
- $\overline{E} = \{\{u, v\} \mid \{u, v\} \notin E, u \neq v \text{ et } u, v \in V\}$
- Propriété : $G$ et $\overline{G}$ forment partition complète des arêtes




## Tournois

- **Définition** : Graphe orienté simple où chaque paire sommets reliée par exactement un arc
- **Construction** : Orientation complète d'un graphe complet
- **Propriétés fondamentales** :
- Graphe sous-jacent = graphe complet $K_n$
- Nombre total d'arcs = $\binom{n}{2}$
- Au plus 1 sommet sans prédécesseurs (source)
- Au plus 1 sommet sans successeurs (puits)

**Caractérisation acyclique** : Tournoi sans circuits $\iff$ matrice adjacence définit ordre strict total **Applications** : Modélisation compétitions, classements, votes

## Graphes bipartis

- **Définition** : Graphe $G = (V, E)$ avec $V = A \cup B$ (A,B disjoints)

tel que toute arête relie sommet de A à sommet de B
- **Notation** : $G = (A, B, E)$ ou $G = (A \cup B, E)$

**Théorème caractérisation** : Graphe biparti $\iff$ ne contient aucun cycle de longueur impaire

**Graphes bipartis complets** $K_{r,s}$ :
- r sommets dans A, s sommets dans B
- Toute paire (a∈A, b∈B) reliée par arête
- Nombre arêtes = $r \times s$
- Applications : modélisation relations complètes entre deux ensembles

## Recouvrements et transversaux

- **Recouvrement** : Sous-ensemble $R \subseteq E$ d'arêtes tel que chaque sommet du graphe est extrémité d'au moins une arête de R
- **Problème du recouvrement minimum** : Trouver recouvrement R de cardinal minimal
- **Transversal** : Sous-ensemble $T \subseteq V$ de sommets tel que chaque arête du graphe est incidente avec au moins un sommet de T
- **Problème du transversal minimum** : Trouver transversal T de cardinal minimal

**Complexité algorithmique** :
- Recouvrement minimum : **Polynomial** (problème "facile")
- Transversal minimum : **NP-difficile** (problème "difficile")

## Couplages et chaînes augmentantes

**Couplage** : Ensemble M ⊆ E d'arêtes sans extrémités communes
- **Couplage parfait** : Sature tous les sommets du graphe
- **Couplage maximum** : Cardinal maximal parmi tous couplages possibles
- **Couplage maximal** : Ne peut être étendu (≠ maximum)
- **Sommet saturé** : Incident à arête du couplage

**Chaînes alternées** (relativement à couplage M) : Chaîne dont arêtes alternent : dans M, hors M, dans M, hors M, …

**Chaînes augmentantes** (relativement à M) : Chaîne alternée avec extrémités NON saturées par M **Propriété clé** : Permet augmenter taille couplage de 1

**Théorème de Berge (1957) -** Condition optimalité : Couplage M maximum $\iff$ graphe ne contient aucune chaîne augmentante relative à M

**Démonstration** :
- **($\Longrightarrow$)** Si C chaîne augmentante, alors $M' = M \triangle C$ (différence symétrique) est couplage de cardinal $|M| + 1$
- **($\Longleftarrow$)** Si M non maximum, ∃ couplage M' avec |M'| > |M|. Dans $M \triangle M'$, il existe une chaîne alternée avec plus d'arêtes de M' que de M → chaîne augmentante

**Algorithme général de recherche couplage maximum** :
1. Partir d'un couplage M (souvent vide)
2. Tant qu'il existe une chaîne augmentante C : a) Trouver chaîne augmentante C b) Remplacer M par $M \triangle C$ (différence symétrique)
3. M est maximum

**Cas graphes bipartis -** Construction graphe orienté :
- Arêtes de M : orientées de B vers A
- Arêtes hors M : orientées de A vers B
- Explorer depuis sommets non saturés de A vers sommets non saturés de B
- Complexité : $O(mn)$ (Hopcroft-Karp : $O(m\sqrt{n})$)

**Applications algorithmes** : Base algorithmes hongrois, Blossom


# Graphes planaires

## Définitions et formule d'Euler

- **Planaire** : Représentable sur le plan sans croisements d'arêtes
- **Faces** : Régions délimitées par les arêtes (incluant face extérieure)
- **Formule d'Euler** : $n - m + f = 2$ (graphe connexe planaire)


Quatre représentations du graphe complet $K_4$. Seules les trois dernières sont des graphes planaires topologiques.

## Bornes et non-planarité

- **Inégalité générale** (graphes simples connexes, $n \geq 3$) :

$m \leq 3n - 6$
- **Formule Euler** : $f = 2 - n + m$
- **Substitution** : $3(2 - n + m) \leq 2m \rightarrow m \leq 3n - 6$

# Graphes Eulériens

**Définitions:**
- Chaîne/cycle eulérien passe 1 fois par chaque arête.
- Chemin/circuit eulérien passe 1 fois par chaque arc.
- Graphe eulérien possède cycle/circuit eulérien.

**Observation:** Cycle arrive/repart même nb fois → tous sommets degré **pair** si eulérien.

## Caractérisations

**Non orienté**
- Graphe $G$ connexe eulérien $\iff$ tous sommets degré pair.
- Exactement 2 sommets degré impair → chaîne eulérienne (extrémités=sommets impairs).

**Orienté**
- Graphe $G$ connexe eulérien $\iff \deg_+(v) = \deg_-(v)$ ∀v.
- Égalité sauf 2 sommets (écart=1) → chemin eulérien (extrémités=sommets différents).
- Si $\deg_+(v) = \deg_-(v)$ ∀v graphe connexe → fortement connexe car eulérien.

## Construction circuit eulérien

1. Choisir sommet r, construire anti-arborescence recouvrante d'anti-racine r
2. Circuit itératif depuis r: choisir arc anti-arborescence uniquement si dernier arc non utilisé quittant sommet (≠r)

## Postier chinois

Graphe $G = (V, E)$ connexe, $c : E \to R_+$. Tournée passant ≥1 fois par chaque arête, longueur minimale.

**Note**
Tous sommets degré pair → cycle eulérien = solution optimale. Sinon → dédoubler certaines arêtes, minimiser somme longueurs dédoublées.

# Graphes Hamiltoniens

**Définitions:** Chaîne/chemin hamiltonien passe 1 fois par chaque sommet. Cycle/circuit hamiltonien idem fermé. Graphe hamiltonien possède cycle/circuit hamiltonien.

Chaîne hamiltonienne = chaîne élémentaire longueur n-1 (n sommets).

**Note**
Recherche chemin hamiltonien: problèmes ordonnancement n tâches avec contraintes antériorité (pas exécution simultanée).

## Voyageur de commerce (TSP)

Plus court circuit hamiltonien dans graphe (généralement complet) pondéré. Optimisation combinatoire.

## Contraintes graphes potentiels-tâches