

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

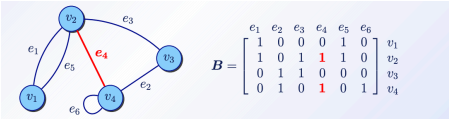
- **Graphe partiel**: tous les sommets, sous-ensemble des arêtes
- **Sous-graphe**: sous-ensemble des sommets avec leurs arêtes
- **Sous-graphe partiel**: sous-ensemble des sommets et des arêtes

Structures dans les graphes

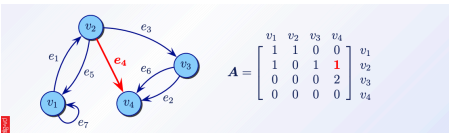
- **Chaîne**: succession de sommets et arêtes
- **Cycle**: chaîne fermée (même sommet au début et à la fin)
- **Chemin**: succession de sommets et arcs
- **Circuit**: chemin fermé
- **Forêt**: graphe sans cycles
- **Arbre**: forêt connexe

Matrices

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

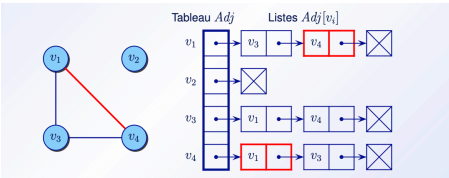


- **Matrices d'adjacence**:
 - Matrices carrées $n \times n$ indiquant l'adjacence des sommets
 - Symétriques pour graphes non orientés
 - Permettent de représenter tout type de graphe (avec boucles)

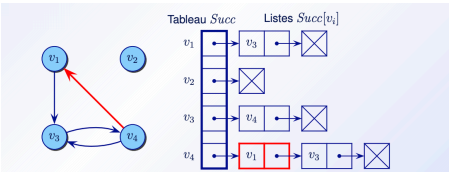


Listes

- **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



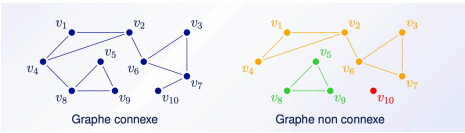
- **Tableaux compacts de successeurs**:
 - Composés de deux tableaux (TabSucc et tableau indexé)
 - Optimisation mémoire

Comparaison

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

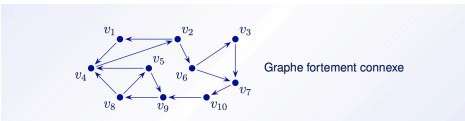
Graphes connexes

- **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



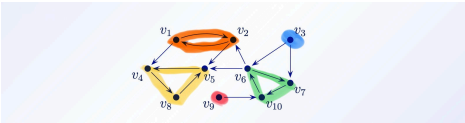
Graphes fortement connexes

- **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



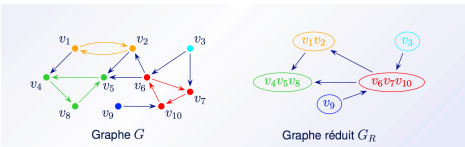
Composantes fortement connexes

- 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
 - $\Omega(g)$: croissance au moins aussi rapide que g
 - $\Theta(g)$: 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

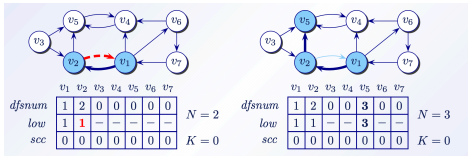
1. Construire le graphe transposé (inversion des arcs)
2. Effectuer une exploration en profondeur du graphe transposé
3. Utiliser les dates de fin de traitement de l'exploration pour ordonner les sommets dans l'ordre décroissant de leur date de fin
4. Effectuer une exploration du graphe initial en utilisant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

Tarjan

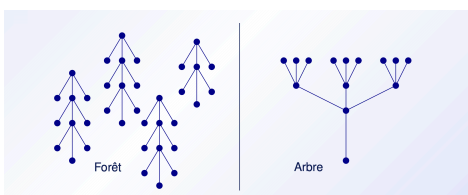
Permet de calculer les composantes fortement connexes d'un graphe orienté.

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. Numérotter 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 arbre 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 commençant 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	$O(n^2)$	$O(m \log n)$	$O(m + n \log n)$

Arbrescences 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.



Arbrescences 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.

Idee

- Identifier les circuits du graphe en prenant pour chaque sommet le plus petit arc entrant
- Si on trouve un circuit, les regrouper en un seul sommet
- 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**
- Répéter jusqu'à ce qu'il n'y ait plus de circuits
- 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 négatif. 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.

Idee

- Initialisation:** distance vers elle-même est 0, distance vers autres = infinie au départ
- Relaxation des arêtes:** pour chaque arête du graphe, vérifier si on peut améliorer le chemin connu vers sa destination en passant par cette arête
 - Répéter cette vérification (n-1) fois, où n est le nombre de sommets, pour garantir que tous les plus courts chemins sont trouvés
- 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				Valeur de Continuer
		1	2	3	4	
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)			
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 Belleman-Ford-Yens

Au lieu de traiter les arcs nous traitons les sommets. L'algorithme s'arrête lors-ce que le **File** est vide ou que l'on a fait n itérations.

Idee

- On part du sommet 0 (la source) distance vers 1 est 0, vers les autres sommets infinie
- On utilise 0 comme sentinelle pour ajouter les prochains sommets à traiter après la sentinelle
- On traite les sommets un par un en mettant à jour la distance vers les sommets voisins jusqu'à atteindre la sentinelle, itération + 1, on remet la sentinelle à 0 et on résume le traitement de l'itération avant de passer à l'itération suivante

Itér. k	Sommet traité	Couples (λj, p[j]) pour chaque sommet						File Q après trait.
		1	2	3	4	5	6	
0	—	(0, -)	(∞, -)	(∞, -)	(∞, -)	(∞, -)	(∞, -)	(1,0)
1	0		(2,1)		(1,1)			(0,2,4)
	0	(0, -)	(2,1)	(∞, -)	(1,1)	(∞, -)	(∞, -)	(2,4,0)

2	2			(6,2)			(4,0,3)
	4		(0,4)		(5,4)		(0,3,2,5)
	0	(0, -)	(0,4)	(6,2)	(1,1)	(5,4)	(∞, -)
3	3						(3,2,5,0)
	2			(4,2)			(2,5,0)
	5						(5,0,3)
4	0	(0, -)	(0,4)	(4,2)	(1,1)	(5,4)	(∞, -)
	3					(3,3)	(0,5)
	0	(0, -)	(0,4)	(4,2)	(1,1)	(3,3)	(∞, -)
5	5						(0)
	0	(0, -)	(0,4)	(4,2)	(1,1)	(3,3)	(∞, -)

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 complexité de l'algorithme de Dijkstra est en $O(n^2)$ si L est gérée à l'aide de tableaux.
- 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.

Algorithme de Floyd-Warshall

L'algorithme de Floyd-Warshall se construit avec deux matrices:

- W qui contient les poids des arcs et ∞ si aucune relation n'existe 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.

Idee

- Initialisation: On initialise la matrice W avec les poids des arcs et la matrice P avec les prédécesseurs immédiats
- Elle est en O(m log n) si L est une queue de priorité simple (un tas binaire).
- On répète cette opération pour tous les sommets

Complexité

La complexité de l'algorithme de Floyd-Warshall est en $O(n^3)$.

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.

Idee

- Ajouter un sommet auxiliaire 0 relié à tous les autres sommets par des arcs de poids nul
- 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
- 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
 - $c'_{ij} = c_{ij} - (\delta_j - \delta_i) = c_{ij} + \delta_i - \delta_j$
 - poids + départ - arrivée
- Appliquer l'algorithme de Dijkstra sur ce graphe repondéré pour chaque sommet source
- Corriger les distances obtenues pour retrouver les distances réelles dans le graphe d'origine

Complexité

- Complexité totale: $O(mn + n^2 \log n)$

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

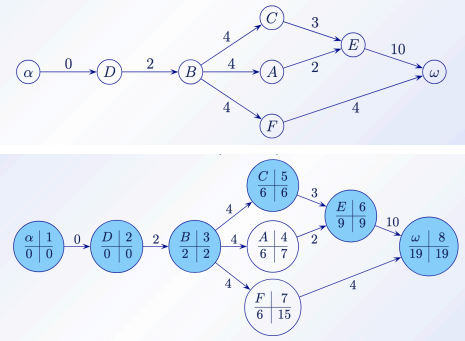
- **Applications** : Ordonnancement projets, optimisation, planification
- **Avantage DAG** : Traitement dans l'ordre topologique, pas d'itérations multiples comme Bellman-Ford classique
- **Plus court chemin** :
 - $\lambda_j = \min_{i \in \text{Pred}[j]} (\lambda_i + c_{ij})$ avec $\lambda_s = 0$
- **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 précedence (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

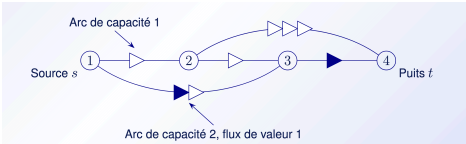
Nom des tâches	Numéros topologiques
----------------	----------------------

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 - $O(m^2n)$

- **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

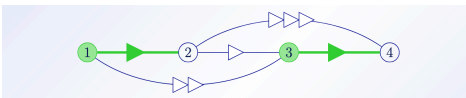
Coupe et théorème max-flow min-cut

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

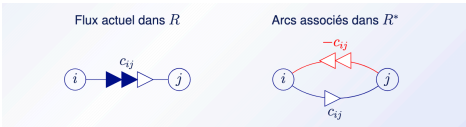
Algorithme de Busacker-Gowen

- **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** : λ_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

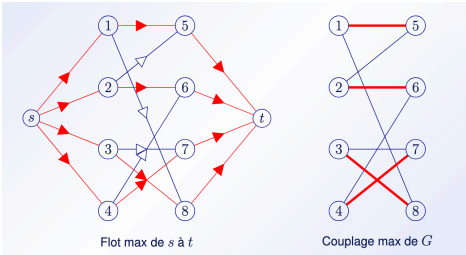


Applications des flots

Couplage maximum dans un graphe biparti

Transformation :

1. Orienter arêtes $A \rightarrow B$ (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

À gauche : réseau initial comptant 2 sources (v_1 et v_2), 2 puits (v_4 et v_5) et un sommet de transbordement (v_3) (les coûts et les capacités des arcs ne sont pas représentés, ils ne sont pas affectés par la transformation).

À droite : réseau après transformation où il faut déterminer un flot max à coût min de s à t (les arcs ajoutés ont un coût unitaire d'utilisation nul).



RAPPEL : En transbordement, il est conventionnel de représenter les offres par des nombres négatifs et les demandes par des nombres positifs.

Autres types de graphes

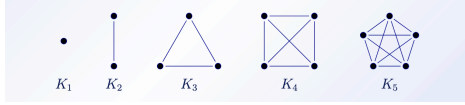
Graphe complet K_n : Graphe simple où toute paire

- 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 \bar{G} de $G = (V, E)$:

- Mêmes sommets que G
- Arêtes = toutes arêtes possibles non présentes dans G
- $\bar{E} = \{\{u, v\} \mid \{u, v\} \notin E, u \neq v \text{ et } u, v \in V\}$
- Propriété : G et \bar{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 \Leftrightarrow 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 \Leftrightarrow ne contient aucun cycle de longueur impaire

Graphes bipartis complets $K_{r,s}$:

- r sommets dans A , s sommets dans B
- Toute paire ($a \in A$, $b \in B$) 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 $M \subseteq E$ 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 (\neq 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 \Leftrightarrow graphe ne contient aucune chaîne augmentante relative à M

Démonstration :

- (\Rightarrow) Si C chaîne augmentante, alors $M' = M \Delta C$ (différence symétrique) est couplage de cardinal $|M| + 1$
- (\Leftarrow) Si M non maximum, \exists couplage M' avec $|M'| > |M|$. Dans $M \Delta M'$, il existe une chaîne alternée avec plus d'arêtes de M' que de $M \rightarrow$ chaîne augmentante

Algorithme général de recherche couplage maximum :

- Partir d'un couplage M (souvent vide)
- Tant qu'il existe une chaîne augmentante C : a) Trouver chaîne augmentante C b) Remplacer M par $M \Delta C$ (différence symétrique)
- 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

- Graphe planaire** : Graphe admettant une représentation sur le plan où les arêtes ne se coupent pas (sauf aux extrémités).
- Faces** : Régions connexes créées par le découpage du plan par les arêtes d'un graphe planaire.

Formule d'Euler

Pour tout graphe planaire connexe :

$$n - m + f = 2$$

n = nombre de sommets, m = nombre d'arêtes, f = nombre de faces

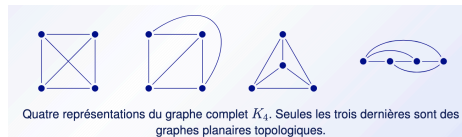
Graphes simples connexes

Pour $n \geq 3$ sommets :

$$m \leq 3n - 6$$

Graphes bipartis connexes

Pour $n \geq 4$ sommets :

$$m \leq 2n - 4$$


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

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 \rightarrow tous sommets degré **pair** si eulérien.

Caractérisations

Non orienté

- Graphe G connexe eulérien \Leftrightarrow tous sommets degré pair.
- Exactement 2 sommets degré impair \rightarrow chaîne eulérienne (extrémités=sommets impairs).

Orienté

- Graphe G connexe eulérien $\Leftrightarrow \deg_+(v) = \deg_-(v) \forall v$.
- Égalité sauf 2 sommets (écart=1) \rightarrow chemin eulérien (extrémités=sommets différents).
- Si $\deg_+(v) = \deg_-(v) \forall v$ graphe connexe \rightarrow fortement connexe car eulérien.

Construction circuit eulérien

- Choisir sommet r , construire anti-arborescence recouvrante d'anti-racine r
- Circuit itératif depuis r : choisir arc anti-arborescence uniquement si dernier arc non utilisé quittant sommet ($\neq r$)

Postier chinois

Graphe $G = (V, E)$ connexe, $c : E \rightarrow \mathbb{R}_+$. Tournée passant ≥ 1 fois par chaque arête, longueur minimale.

Note

Tous sommets degré pair \rightarrow cycle eulérien = solution optimale. Sinon \rightarrow dédoubler certaines arêtes, minimiser somme longueurs dédoublées.

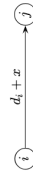
Graphes Hamiltoniens

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

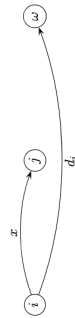
- Graphes K_n est hamiltonien pour $n \geq 3$.
- Graphes bipartis complets $K_{r,s}$ est hamiltonien si $\min(r, s) \geq 2$ et $r = s$

Contraintes graphes potentiels-tâches

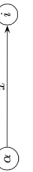
La contrainte : « la tâche j peut débuter au plus tôt x jours après la fin de la tâche i » se modélise par un arc (i, j) de poids $c_{ij} = d_i + x$:



La contrainte : « la tâche j peut commencer au plus tôt x jours après le début de la tâche i » se modélise par un arc (i, j) de poids $c_{ij} = x$ ainsi qu'un arc (i, ω) de poids $c_{i\omega} = d_i$ (afin d'éviter que la fin du projet ne puisse débuter avant la fin de i) :

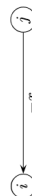


La contrainte : « la tâche i débute au plus tôt x jours après le commencement des travaux » se modélise par un arc (α, i) de poids $c_{\alpha i} = x$:



La contrainte : « la tâche j doit commencer si tôt la tâche i terminée » se modélise par un arc (i, j) de poids $c_{ij} = d_i$ et un arc (j, i) de poids $c_{ji} = -d_i$ car on a

$$t_j \leq t_i + x \iff t_i - t_j \geq -x.$$



$$\begin{cases} t_i - t_j \geq -d_i \\ t_j - t_i \geq d_i \end{cases}$$

$$\begin{cases} t_j \leq t_i + d_i \\ t_j \geq t_i - d_i \end{cases}$$



La contrainte : « la tâche j doit commencer si tôt la tâche i terminée » se modélise par un arc (i, j) de poids $c_{ij} = d_i$ et un arc (j, i) de poids $c_{ji} = -d_i$ car on a

$$t_j = t_i + d_i$$