

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

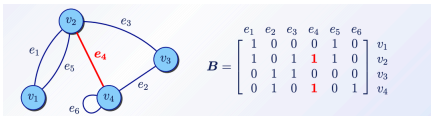
- Graphe partiel**: tous les sommets, sous-ensemble des arêtes
- Sous-graphe**: sous-ensemble des sommets avec leurs arêtes
- Sous-graphe partiel**: sous-ensemble des sommets et des arêtes

Structures dans les graphes

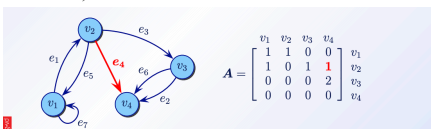
- Chaîne**: succession de sommets et arêtes
- Cycle**: chaîne fermée (même sommet au début et à la fin)
- Chemin**: succession de sommets et arcs
- Circuit**: chemin fermé
- Forêt**: graphe sans cycles
- Arbre**: forêt connexe

Matrices

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

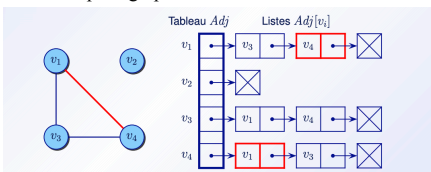


- Matrices d'adjacence**:
 - Matrices carrées $n \times n$ indiquant l'adjacence des sommets
 - Symétriques pour graphes non orientés
 - Permettent de représenter tout type de graphe (avec boucles)

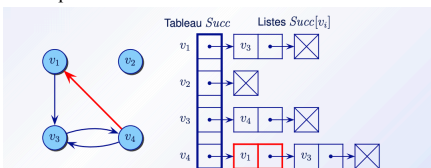


Listes

- 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



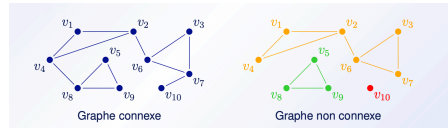
- Tableaux compacts de successeurs**:
 - Composés de deux tableaux (TabSucc et tableau indexé)
 - Optimisation mémoire

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



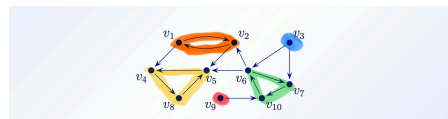
Graphes fortement connexes

- 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



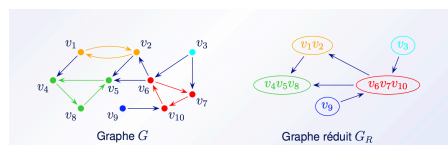
Composantes fortement connexes

- 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

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

Complexité

Exécution

$$O(n + m)$$

Mémoire

$$O(n) +$$

$$O(n + m) \text{ pour stocker le graphe}$$

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

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

Complexité

$$O(n + m)$$

Kosaraju

L'algorithme de Kosaraju est un algorithme permettant de calculer les composantes fortement connexes d'un graphe orienté.

Idée

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

Tarjan

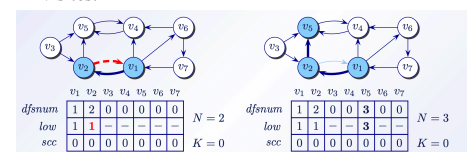
L'algorithme de Tarjan est un algorithme permettant de calculer les composantes fortement connexes d'un graphe orienté en une seule exploration.

Idée

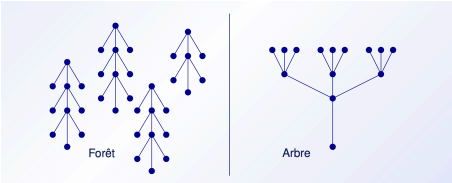
On utilise 3 tableaux :

- 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

- Utiliser trois tableaux : **dfsnum**, **low**, et **scc**.
- Numéroter les sommets au fur et à mesure de leur visite (**dfsnum**).
- Stocker le plus petit numéro atteignable depuis chaque sommet (**low**).
- Utiliser une pile pour suivre les sommets de la composante actuelle.
- Mettre à jour **low** pour chaque sommet et ses voisins.
- Lorsqu'un sommet est la racine d'une composante fortement connexe, attribuer un numéro de composante (**scc**) et retirer les sommets de la pile.
- 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 (sommets 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:

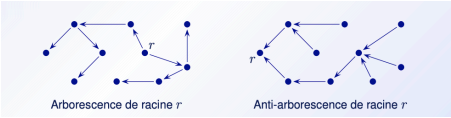
	Tableau	Tas binaire	Tas de Fibonacci
Prim	$O(n^2)$	$O(m \log n)$	$O(m + n \log n)$

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 coût 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 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 Bellman-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 (A_j, p[j]) pour chaque sommet				Valeur de Continuer
			1	2	3	4	
1	Valeurs de départ		(0, -)	(∞, -)	(∞, -)	(∞, -)	faux
		(1,2)		(3,1)			vrai
		(1,3)				(4,1)	
		(2,1)					
		(2,3)			(7,2)		
		(3,4)					
2	Valeurs de départ		(0, -)	(1,4)	(6,4)	(4,1)	faux
		(2,3)			(5,2)		vrai
3	Valeurs de départ		(0, -)	(1,4)	(5,2)	(4,1)	faux
	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:
- Dans l’algorithme de Prim, le coût de connexion du sommet est compté depuis son parent direct
 - Dans l’algorithme de Dijkstra, le coût de connexion du sommet est compté 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 ∞ 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.

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 additionne 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
 - $c'_{ij} = c_{ij} - (\delta_j - \delta_i) = c_{ij} + \delta_i - \delta_j$
 - 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)