Arborescences et racines

GRE

5 - Arborescences et racines

Abstract

Definition

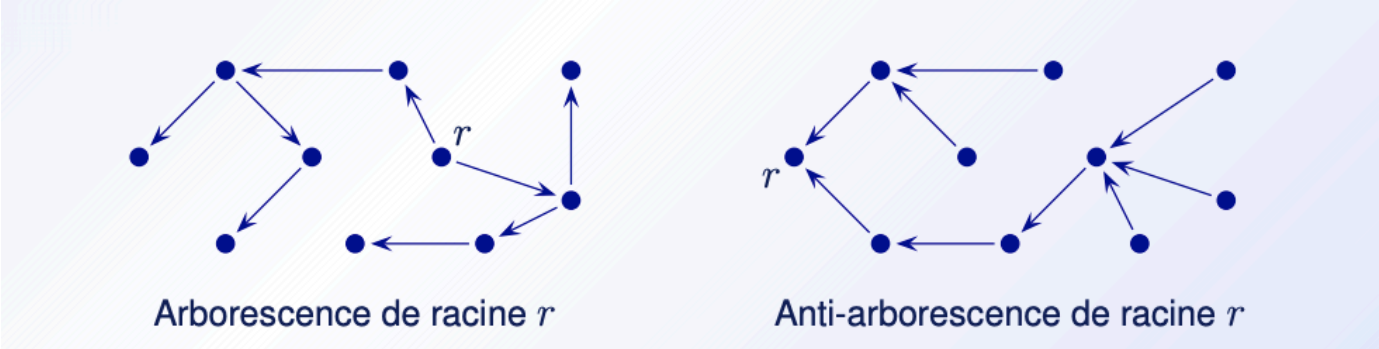
Table des matières

1. Arborescences et racines

1.1. 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.



Note
Dans une arborescence, chaque sommet possède exactement un prédécesseur sauf la raine qui n'en a aucun.
- Tout graphe connexe vérifiant la propriété précédente est une arborescence.
- De même, tout graphe sans circuits vérifiant la propriété précédente est une arborescence.

1.2. 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 êter 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**.

1.3. Arborescence de poids minimum

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

1.4. Chu-Liu

L'algorithem de Chu-Liu est un algorithme de recherche d'une arborescence recouvrante de racine s et de poids minimum.

1.5. Idée

1. Identifier les circuits du graphe
2. Les regroupers 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
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

2. Plus courts chemins dans les réseaux

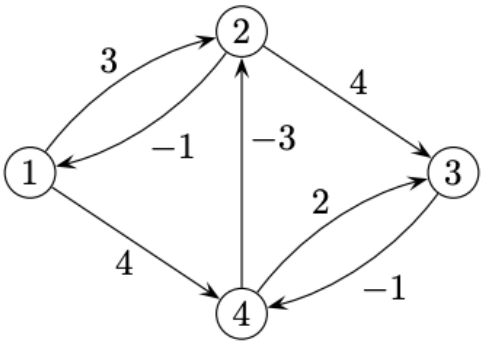
Dans des réseaux, il se peut que l'on trouve des circuits à coût négaitf. Par cela on entend qu'il existe un circuit qui permet de diminuer le coût d'un chemin. On parle aussi de circuit absorbant.S

2.1. 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.

2.1.1. 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,—)	(∞,—)	(∞,—)	(∞,—)	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

2.2. 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**

2.2.1. 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 $O(n)$.

2.3. 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'éxiste entre deux sommets
- P qui contient les prédécesseurs immédiats de chaque sommet

2.3.1. 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 regarde si on peut améliorer le poids d'un sommet en passant par le sommet k

3. On répète cette opération pour tous les sommets

2.3.2. 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.

2.4. Algorithme de Dantzig

L'algorithme de Dantzig résout le même problème que l'algorithme de Floyd-Warshall (calcul des plus courts chemins pour tous les couples de sommets et détection de circuits absorbants) avec une complexité asymptotique identique en O(n³) et un espace mémoire en O(n²). Cependant, il utilise une décomposition différente du problème : à chaque itération k, il calcule les plus courts chemins entre tous les couples de sommets du sous-graphe Gk engendré par les k premiers sommets, en s'appuyant sur les résultats de l'itération précédente. Après n itérations, le sous-graphe Gn correspond au graphe complet de départ, et les valeurs finales représentent les longueurs des plus courts chemins entre tous les couples de sommets du réseau.

2.5. 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.

2.5.1. 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
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

2.5.2. 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: $O(mn + n^2 \log n)$

2.5.3. 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)

Note
Le principe clé de l'algorithme de Johnson est la "repondération" (reweighting) qui transforme un problème avec poids négatifs en un problème équivalent avec uniquement des poids positifs, tout en préservant la structure des plus courts chemins.