ALGORITHMES DES GRAPHES

INTRODUCTION

La théorie des graphes et l'algorithmique qui lui est liés est un des outils les plus utilisés pour la modélisation et la résolution des problèmes dans bon nombres de domaines allant de la science fondamentale aux applications technologiques concrètes.

La mise en applications dans le domaine des graphes va de la résolution des problèmes combinatoire aux problèmes concurrentiels passant par le routage du Traffic dans les réseaux de télécommunications et les réseaux d'ordinateurs. La notion des graphes sont régit par un mode de fonctionnement, un ensemble de concepts et propriétés propre à elle et vu sur un angle de structure de données un ensemble d'algorithmes.

Présentés ainsi il nous incombera dans le cadre de notre travail de présenter de manière globale la notion de graphe, de faire une étude sur les algorithmes à elle appliqués.

PARTIE I : ALGORITHMES DES GRAPHES

CHAPITRE I : NOTION DE GRAPHE

I. Définition

1. Type de Graphe

Il existe principalement deux types de graphes :

- ✓ Graphe Orienté
- ✓ Graphe non Orienté

a. Graphe Orienté

- Un graphe orienté $G = (X, S)$ est représenté par la donnée :
  - D'un ensemble de sommet ou nœuds X
  - D'un ensemble ordonné U de couples de sommets appelés arcs
- Le nombre de sommet d'un graphe est l'ordre du graphe
- Si $\mathrm{U} = (\mathrm{a}, \mathrm{b})$est un arc de G alors :
  - a est l'extrémité initiale de U
  - b est l'extrémité terminale de U
  - a et b sont adjacents
- Les arcs ont un sens. L'arc $U = (a, b)$ va de a vers b
- Ils peuvent être munit d'un coût, d'une capacité, etc...
- on note $\omega(i)$: l'ensemble des arcs ayant i comme extrémité
- on note $\omega + (i)$: l'ensemble des arcs ayant i comme extrémité initiale = ensemble des arcs sortant de i
- on note $\omega - (i)$: l'ensemble des arcs ayant i comme extrémité terminale $=$ ensemble des arcs entrant dans i
- $\Gamma(i)$ ensembles des successeurs de i
- $d + (i)$: degré sortant de i (nombre de successeurs)
- $d - (i)$: degré entrant dans i (nombre des sommets pour lesquels i est un successeur)

b. Graphe Non-Orienté

- Un graphe $G = (X, S)$ est représenté par :
  - D'un ensemble de sommet ou nœuds X
  - D'un ensemble ordonné U de couples de sommets appelés arêtes
- Les arêtes ne sont pas orientées
- Deux sommets sont voisins s'ils sont reliés par un arc ou une arête

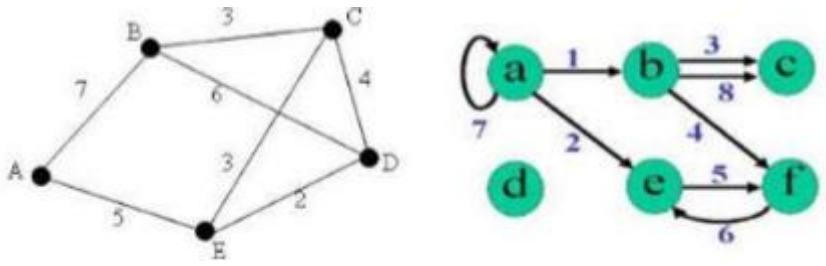


Figure 1:Graphe non Orienté et Graphe Orienté

II. Vocabulaire sur les graphes

Propriétés	Signification
Arc	couple (x, y) dans un graphe orienté
Arête	nom d'un arc, dans un graphe non orienté
Boucle	arc reliant un sommet à lui-même
Chaine	Nom d'un chemin dans un graphe non orienté ; séquence d'arcs avec une extrémité commune dans un graphe orienté.
Chemin	suite d'arcs connexes reliant un sommet à un autre. Par exemple $(a;b)$ $(b;c)$ $(c;d)$ $(d;b)$ $(b;e)$ est un chemin reliant a à e ; on le note(a,b,c,d,b,e). Un chemin est une chaîne, la réciproque étant fausse
Chemin eulérien	désigne un chemin simple passant une fois et une seule par toutes les arêtes du graphe ; il n'existe pas toujours…
Chemin Hamiltonien	désigne un chemin simple qui passe une fois et une seule par chaque sommet
Circuit	chemin dont l'origine et l'extrémité sont identiques
Connexité	Un graphe est connexe s'il existe toujours une chaîne, ou un chemin, entre deux sommets quelconques. Par exemple le plan d'une ville doit être connexe.
Graphe complet	un graphe est complet si quels que soient deux sommets distincts, il existe un arc (ou une arête) les reliant dans un sens ou dans l'autre
Cycle	nom d'un circuit dans un graphe non orienté ; dans un graphe orienté, un cycle est une chaîne dont l'extrémité initiale coïncide avec l'extrémité finale

Degré d'un sommet	nombre d'arête issues d'un sommet dans un graphe non orienté ; nombre d'arcs arrivant ou partant d'un sommet dans un arc orienté ; on peut vérifier facilement que la somme des degrés de tous les sommets, est donc le double du nombre des arêtes (puisque chacune est comptée deux fois).
Diamètre	le diamètre d'un graphe est la plus grande chaîne (chemin) de toutes reliant deux sommets quelconques du graphe
Distance	la distance entre deux sommets d'un graphe est la plus petite longueur des chaînes, ou des chemins, reliant ces deux sommets.
Graphe simple	désigne un graphe non orienté n'ayant pas de boucle ni plus d'une arête reliant deux sommets. Sur le dessin, les liens entre les sommets sont des segments, et on ne parle alors plus d'arcs mais d'arêtes ; tout graphe orienté peut donc être transformé en graphe simple, en remplaçant les arcs par des arêtes
Longueur d'un chemin ou d'une chaine	nombre d'arcs du chemin (ou d'arêtes de la chaîne)
Ordre d'un graphe	nombre de sommets du graphe
Rang	le rang d'un sommet est la plus grande longueur des arcs se terminant à ce somme
Sous graphe	le graphe G' est un sous graphe de G si l'ensemble des sommets de G' est inclus dans l'ensemble des sommets de G, et si l'ensemble des arcs de G' est égal au sous ensemble des arcs de G reliant entre eux tous les sommets de G' ; on a donc retiré de G certains sommets, et tous les arcs adjacents à ces sommets
Stable	soit un graphe $G = (E; R)$, et f un sous-ensemble de sommets. On dit que f est un sous ensemble stable de E s'il n'existe aucun arc du graphe reliant deux sommets de f.
Prédécesseur	Dans l'arc(x, y), x est prédécesseur de y
Successeur	dans l'arc(x, y), y est successeur de x
Graphe partiel	Soi $G = (V, E)$ un graphe. Le graphe $G = (V, E')$ est un graphe partiel de G, si E' est inclus dans E. Autrement dit, on obtient G' en enlevant une ou plusieurs arêtes graphe G.
Un sommet pendant	est un sommet de degré 1.

Un pont	est une arête telle que sa suppression déconnecte le graphe en question.
Demi-degré intérieur (degré entrant) d'un sommet x ,) $(xd-)$,	est le nombre d'arcs entrant$x$,
Demi-degré extérieur (degré sortant) d'un sommet x ,) $(xd+)$,	est le nombre d'arcs sortant de x.
Un graphe valué (pondéré)	est un graphe où les arcs (arêtes) possèdent un poids.

Tableau 1: Tableau récapitulatifs du vocabulaire sur les Graphes

III. Connexité et forte-connexité

Dans un graphe, on note également la notion de **connexité** et **de forte connexité**.

On dit de deux **sommets** x et y qu'ils sont **connexes** s'il existe une chaine entre ces deux sommets ou si $x = y$.

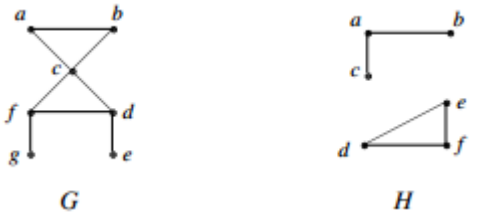


Figure 2 : graphe connexes et graphe non connexes

On appelle **composante connexe**, un ensemble de sommets qui ont la relation de connexité deux à deux.

Un graphe $G = (S, A)$ est dit **graphe connexe** si tous ses sommets ont deux à deux la relation de connexité ou s'il possède une seule composante connexe.

On dit de deux **sommets** x et y qu'ils sont **fortement connexes** s'il existe un chemin de x vers y et vis-versa ou si $x = y$.

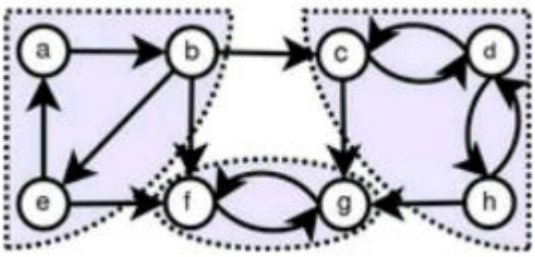


Figure 3 : Graphe fortement connexe

On appelle **composante fortement connexe**, un ensemble de sommets qui ont la relation de forte connexité.

Un graphe $G = (S, A)$ est dit **graphe fortement connexe** si tous ses sommets ont deux a deux la relation de forte connexité ou s'il possède une seule composante fortement connexe.

IV. Arbres, Arborescence et Arbre couvrant

Les arbres et les arborescences sont des graphes particuliers très souvent utilisés en informatique pour représenter des données, entre autres.

1. Arbre, Forêt et Arborescence

Etant donné un graphe non orienté comportant n sommets, les propriétés suivantes sont équivalentes pour caractériser :

- ✓ Un **arbre** :
  - ❖ G est connexe et sans cycle,
  - ❖ G est sans cycle et possède $n - 1$ arêtes,
  - ❖ G est connexe et admet $n - 1$ arêtes,
  - ❖ G est sans cycle, et en ajoutant une arête, on crée un et un seul cycle élémentaire,
  - ❖ G est connexe, et en supprimant une arête quelconque, il n'est plus connexe,
  - ❖ Il existe une chaine et une seule entre 2 sommets quelconques de G.

- ✓ Une **forêt** est un graphe dont chaque composante connexe est un arbre.

- ✓ Une **arborescence** est un graphe orienté sans circuit admettant une racine $s_0 \in S$ telle que, pour tout autre sommet $s_i \in S$, il existe un chemin unique allant de s_0 vers s_i.

- ✓ Un **arbre couvrant** de G est un graphe partiel de G sans cycle

V. Représentation des graphes

1. Liste des successions

1	2, 3, 4, 5
2	3
3	4
4	-
5	1, 4

Successeurs

1	5
2	1
3	1, 2
4	1, 3, 5
5	1

Prédécesseurs

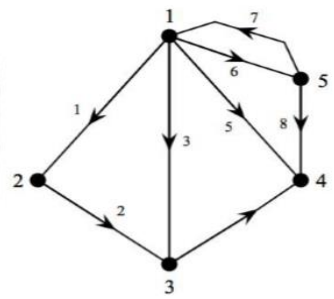


Figure 4: Liste des successeurs

2. Matrice adjacence

Soit le graphe suivant $G1$:

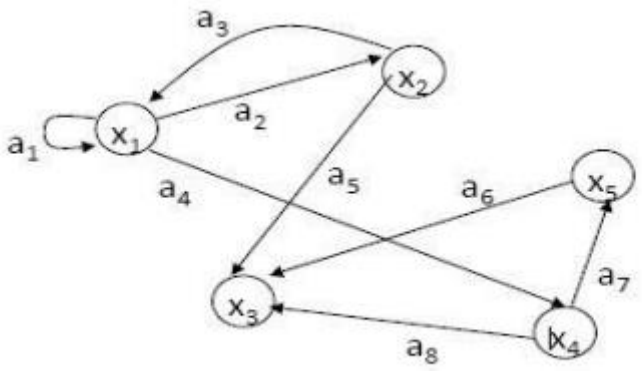


Figure 5: Graphe Exemple G1

Considérons une matrice carrée d'ordre n : les cases sont associées aux couples(x_i, x_j),. Elles sont marquées 1 si (x_i, x_j) est un arc qui existe et 0 si (x_i, x_j) est un arc qui n'existe pas. Dans le cas du graphe dessiné plus haut, voici la matrice booléenne résultante :

	X_1	X_2	X_3	X_4	X_5
X_1	1	1	0	1	0
X_2	1	0	1	0	0
X_3	0	0	0	0	0
X_4	0	0	1	0	1
X_5	0	0	1	0	0

Tableau 2: Tableau Matrice d'adjacence de G1

Lecture : ligne vers colonne = dictionnaire des suivants ; et colonne vers ligne= dictionnaire des précédents.

Si les arcs sont valués, on remplace le 1 par la valeur numérique associée à l'arc correspondant ; la matrice n'est plus booléenne.

3. Représentation par la matrice des arcs

Ce type de représentation sera très utilisé dans la recherche de chemins entre plusieurs sommets. Elle est très facile à mettre en œuvre une fois qu'on a déjà le dessin du graphe connu.

Ici, on trace une matrice qui est pareille à celle de la matrice booléenne ; s'il existe un arc $X_i \rightarrow X_j$ dans le graphe, on entre dans la case de la matrice (i, j) la valeur X_iX_j et s'il n'existe rien, on entre rien.

	X_1	X_2	X_3	X_4	X_5
X_1	X_1X_1	X_1X_2		X_1X_4	
X_2	X_2X_1		X_2X_3		
X_3					
X_4			X_4X_3		X_4X_5
X_5			X_5X_3		

Tableau 3: Tableau Matrice des Arcs

CHAPITRE II : QUELQUES ALGORITHME SUR LES GRAPHES

I. Algorithmes de parcours d'un graphe

1. Parcours en largeur

L'**algorithme de parcours en largeur** (ou BFS, pour *Breadth First Search* en anglais) permet le parcours d'un graphe ou d'un arbre de la manière suivante : on commence par explorer un nœud source, puis ses successeurs, puis les successeurs non explorés des successeurs, etc. L'algorithme de parcours en largeur permet de calculer les distances de tous les nœuds depuis un nœud source dans un graphe non pondéré (orienté ou non orienté). Il peut aussi servir à déterminer si un graphe non orienté est connexe.

a. Principe

Cet algorithme diffère de l'algorithme de parcours en profondeur par le fait que, à partir d'un nœud source S, il liste d'abord les voisins de S pour ensuite les explorer un par un. Ce mode de fonctionnement utilise donc une file dans laquelle il prend le premier sommet et place en dernier ses voisins non encore explorés.

Les nœuds déjà visités sont marqués afin d'éviter qu'un même nœud soit exploré plusieurs fois. Dans le cas particulier d'un arbre, le marquage n'est pas nécessaire.

Étapes de l'algorithme :

1. Mettre le nœud source dans la file.
2. Retirer le nœud du début de la file pour le traiter.
3. Mettre tous les voisins non explorés dans la file (à la fin).
4. Si la file n'est pas vide reprendre à l'étape 2.

Note : l'utilisation d'une pile au lieu d'une file transforme l'algorithme du parcours en largeur en l'algorithme de parcours en profondeur.

b. Exemple

Sur le graphe suivant, cet algorithme va alors fonctionner ainsi :

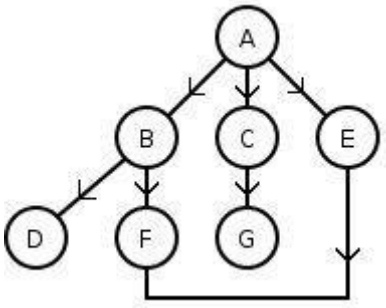


Figure 6:Graphe Parcours en largeur

Il explore dans l'ordre les sommets A, B, C, E, D, F, G, contrairement à l'algorithme de parcours en profondeur qui cherche dans cet ordre : A, B, D, F, C, G, E.

c. *Algorithme*

```
1  Fonction BFS(g, s_0)
       Entrée       : Un graphe g et un sommet s_0 de g
       Postcondition : Retourne une arborescence π d'un parcours en largeur de g à partir de s_0
       Déclaration  : Une file (FIFO) f initialisée à vide
2      pour chaque sommet s_i de g faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5      Ajouter s_0 dans la file f et colorier s_0 en gris
6      tant que la file f n'est pas vide faire
7          Soit s_FirstOut le sommet le plus ancien dans f
8          pour tout sommet s_i ∈ succ(s_FirstOut) faire
9              si s_i est blanc alors
10                 Ajouter s_i dans la file f et colorier s_i en gris
11                 π[s_i] ← s_FirstOut
12         Enlever s_FirstOut de f et colorier s_FirstOut en noir
13     retourne π
```

Figure 7: Algorithme BFS

d. *Complexité*

Soient n et p le nombre de sommets et arcs de G, respectivement. Chaque sommet accessible depuis So est mis au plus une fois dans la file F. À chaque passage dans la boucle (ligne 6 à 12), il y'a exactement un sommet qui est enlevé de la fille cette boucle sera donc exécutée au plus n fois. A chaque fois qu'un sommet est enlever de la file la boucle (ligne 8 à 11) parcourt tous les successeurs du sommet enlevé, de sorte que les lignes 9 à 11 seront exécutées au plus une fois par arcs. Par conséquent, la complexité de l'algorithme est en $\boldsymbol{O(n+p)}$

e. Application

Le parcours en largeur explore tous les sommets accessibles depuis le sommet source. On peut utiliser cet algorithme pour calculer les composantes connexes d'un graphe non orienté avec une complexité linéaire en la taille du graphe.

De plus, lors de ce parcours, les sommets sont explorés par distance croissante au sommet source. Grâce à cette propriété, on peut utiliser l'algorithme pour résoudre le problème de cheminement suivant : calculer des plus courts chemins entre le sommet source et tous les sommets du graphe. L'algorithme de Dijkstra peut être vu comme une généralisation du parcours en largeur avec des arcs pondérés positivement.

Un raffinement appelé LexBFS permet de reconnaître rapidement certaines classes de graphes.

4. Parcours en profondeur

L'algorithme de parcours en profondeur (ou parcours en profondeur, ou DFS, pour Depth-First Search) est un algorithme de parcours d'arbre, et plus généralement de parcours de graphe. Il se décrit naturellement de manière récursive. Son application la plus simple consiste à déterminer s'il existe un chemin d'un sommet à un autre.

a. Principe

L'exploration d'un parcours en profondeur depuis un sommet S fonctionne comme suit. Il poursuit alors un chemin dans le graphe jusqu'à un cul-de-sac ou alors jusqu'à atteindre un sommet déjà visité. Il revient alors sur le dernier sommet où on pouvait suivre un autre chemin puis explore un autre chemin. L'exploration s'arrête quand tous les sommets depuis S ont été visités. Bref, l'exploration progresse à partir d'un sommet S en s'appelant récursivement pour chaque sommet voisin de S.

Le nom d'algorithme en profondeur est dû au fait que, contrairement à l'algorithme de parcours en largeur, il explore en fait « à fond » les chemins un par un : pour chaque sommet, il marque le sommet actuel, et il prend le premier sommet voisin jusqu'à ce qu'un sommet n'ait plus de voisins (ou que tous ses voisins soient marqués), et revient alors au sommet père.

Si G n'était pas un arbre, l'algorithme pourrait a priori tourner indéfiniment si on continuait l'exploration depuis un sommet déjà visité. Pour éviter cela, on marque les sommets que l'on visite, de façon à ne pas les explorer à nouveau.

Dans le cas d'un arbre, le parcours en profondeur est utilisé pour caractériser l'arbre.

b. *Algorithme*

```
1  Fonction DFS(g, s_0)
       Entrée          : Un graphe g et un sommet s_0 de g
       Postcondition   : Retourne une arborescence π d'un parcours en profondeur de g à partir de s_0
       Déclaration     : Une pile (LIFO) p initialisée à vide
2      pour tout sommet s_i ∈ S faire
3          π[s_i] ← null
4          Colorier s_i en blanc
5      Empiler s_0 dans p et colorier s_0 en gris
6      tant que la pile p n'est pas vide faire
7          Soit s_i le dernier sommet entré dans p (au sommet de p)
8          si ∃s_j ∈ succ(s_i) tel que s_j soit blanc alors
9              Empiler s_j dans p et colorier s_j en gris
10             π[s_j] ← s_i
11         sinon
12             Dépiler s_i de p et colorier s_i en noir
13     retourne π
```

Figure 8 : Algorithme DFS

c. *Algorithme Récursif*

```
1  Procédure DFSrec(g, s_0)
       Entrée          : Un graphe g et un sommet s_0 de g
       Précondition    : s_0 est blanc
2      début
3          Colorier s_0 en gris
4          pour tout s_j ∈ succ(s_0) faire
5              si s_j est blanc alors
6                  π[s_j] ← s_0
7                  DFSrec(g, s_j)
8          Colorier s_0 en noir
```

Figure 9: Algorithme Récursif DFS

d. *Complexité*

Chaque sommet accessible depuis So est mis, puis enlevé, exactement une fois dans la pile, comme dans BFS, et à chaque passage dans la boucle lignes 6 à 12, soit un sommet est empilé, soit un sommet est dépilé. Par conséquent, l'algorithme passera au plus $2n$ fois dans la boucle lignes 6 à 12. À chaque passage, il faut parcourir la liste des successeurs de Si pour

chercher un successeur non visité. Si nous utilisons un itérateur qui mémorise pour chaque sommet de la pile le dernier successeur de ce sommet qui était non visité, alors la complexité de DFS est en $O(n+p)$.

e. Applications

Comme les autres algorithmes de parcours de graphe, l'algorithme de parcours en profondeur trouve l'ensemble des sommets accessibles depuis un sommet donné s, c'est-à-dire ceux vers lesquels il existe un chemin partant de s. Il s'agit précisément des sommets marqués par l'algorithme. Ceci s'applique à un graphe orienté ou non orienté. Sur un graphenon orienté, on peut utiliser cette propriété pour le calcul des composantes connexes.

Dans le cas d'un graphe orienté acyclique, le parcours en profondeur permet de calculer untri topologique des sommets.

L'algorithme de Kosaraju effectue un double parcours en profondeur pour calculer les composantes fortement connexes d'un graphe orienté quelconque.

II. Recherche des plus courts chemins

1. Définition des problèmes des plus courts chemins à origine unique

Etant donné un graphe orienté $G = (S, A)$ ne comportant pas de circuit absorbant, une fonction cout : A $\rightarrow$ R et un sommet origine $S_0 \in S$, il s'agit de calculer pour chaque sommet $S_j \in$ S le coût $\delta(S_0, S_j)$ du plus court chemin de S_0 à S_j.

2. Quelques variantes du problème

- *Pour calculer le plus court chemin allant d'un sommet So vers un autre sommet si (la destination est unique), il suffit d'utiliser la résolution du problème précédent, qui calcule tous les plus courts chemins partant de So. Dans ce cas, il est également possible d'utiliser l'algorithme A^* que vous étudierez dans le cours d'Intelligence artificielle. Cet algorithme est généralement plus efficace dès lors qu'il est possible de calculer efficacement une borne minimale de la longueur d'un plus court chemin entre deux points.*

- *Pour calculer tous les plus courts chemins entre tous les couples de sommets possibles, nous pourrions utiliser la résolution du problème à origine unique en résolvant le problème pour chaque sommet du graphe, ce qui rajoute un facteur n à la complexité de l'algorithme. Toutefois, il existe un algorithme plus efficace dans ce cas : l'algorithme de Floyd-Warshall, qui utilise un principe de programmation dynamique pour éviter de recalculer plusieurs fois des chemins.*
- *Pour calculer non pas des plus courts chemins, mais des plus longs chemins, ou encore si la fonction définissant le coût d'un chemin est différente (s'il s'agit, par exemple, d'un produit, un max ou un min des coûts des arcs du chemin), il faudra adapter les algorithmes que nous allons étudier.*

5. Algorithme de DIJSKTRA

a. *Principe*

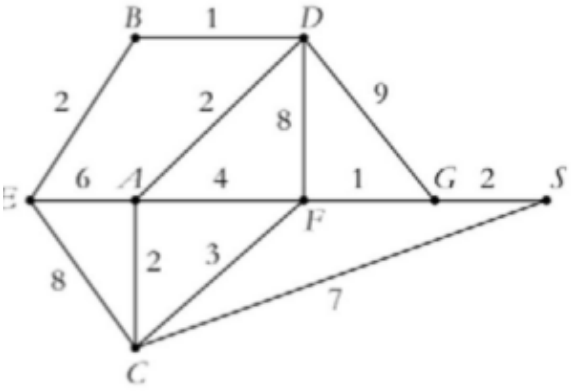
En théorie des graphes, l'algorithme de DIJSKTRA sert à résoudre le problème du plus court chemin. Il permet, par exemple, de déterminer un plus court chemin pour se rendre d'une ville à une autre connaissant le réseau routier d'une région.

L'algorithme (de DIJSKTRA) présenté ici permet, s'il est suivi pas à pas, de n'oublier aucun cas.

- Placer tous les sommets du graphe dans la première ligne d'un tableau. Sur la deuxième ligne, écrire le coefficient 0 sous le point de départ et le coefficient ∞ sous les autres sommets.
- Repérer le sommet X de coefficient minimal ; commencer une nouvelle ligne et rayer toutes les cases vides sous X.
- Pour chaque sommet Y adjacent à X, calculer la somme p du coefficient de X et du poids de l'arête reliant X à Y. Si p est strictement inférieur au coefficient de Y, inscrire pX dans la case correspondante de la colonne Y ; sinon, inscrire le coefficient de Y.
- Compléter la ligne par des coefficients de la ligne précédente.
- S'il reste des sommets non sélectionnés, retournez à l'étape ii. ; sinon, passer à l'étape
- La longueur minimale est le nombre lu sur la dernière ligne du tableau.

La plus courte chaîne ne passe pas toujours par tous les sommets !

Un livreur prépare sa tournée. Il doit visiter un certain nombre de ses clients nommés A, B, C, D, F et G en partant de E pour arriver en S. Les liaisons possibles sont représentées sur le graphe suivant pondéré par les durées en minutes des trajets. On cherche le trajet à emprunter pour minimiser la durée totale du trajet de E à S.



E	A	B	C	D	F	G	S
0	∞	∞	∞	∞	∞	∞	∞
	6E	2E	8E	∞	∞	∞	∞
	6E	2E	8E	∞	∞	∞	∞
	5D		8E		11D	12D	∞
			7A		9A	12D	∞
					9A	12D	∞
							∞
							12G

La durée minimum du trajet EBDAFGS est de 12 minutes.

Figure 10:Exemple DIJSKTRA

b. *Algorithme*

1 **Fonction** *Dijkstra*$(g, cout, s_0)$
  Entrée : Un graphe $g = (S, A)$, une fonction $cout : A \to \mathbb{R}$ et un sommet de départ $s_0 \in S$
  Précondition : Pour tout arc $(s_i, s_j) \in A, cout(s_i, s_j) \geq 0$
  Postcondition : Retourne une arborescence π des plus courts chemins partant de s_0 et un tableau d tel que $d[s_i] = \delta(s_0, s_i)$
2 **pour** *chaque sommet* $s_i \in S$ **faire**
3 $d[s_i] \leftarrow +\infty$
4 $\pi[s_i] \leftarrow null$
5 Colorier s_i en blanc
6 $d[s_0] \leftarrow 0$
7 Colorier s_0 en gris
8 **tant que** *il existe un sommet gris* **faire**
9 Soit s_i le sommet gris tel que $d[s_i]$ soit minimal
10 **pour** *tout sommet* $s_j \in succ(s_i)$ **faire**
11 **si** s_j *est blanc ou gris* **alors**
12 relâcher$((s_i, s_j), \pi, d)$
13 **si** s_j *est blanc* **alors** Colorier s_j en gris;
14 Colorier s_i en noir
15 **retourne** π *et* d

Figure 11:Algorithme de DIJSKTRA

c. *Complexité*

Soient n et p le nombre de sommets et arcs du graphe, respectivement. À chaque passage dans la boucle lignes 8 à 14, exactement un sommet est colorié en noir, et ne pourra plus jamais être recoloré en noir puisque seuls les sommets gris sont coloriés en noir. L'algorithme passera donc au plus n fois dans la boucle (il peut passer moins de n fois si certains sommets ne sont pas accessibles depuis S_0). À chaque passage dans la boucle, il faut chercher le sommet gris ayant la plus petite valeur de d, puis relâcher tous les arcs partant de ce sommet et arrivant sur un sommet non noir. Si la recherche du sommet ayant la plus petite valeur de d est faite linéairement, alors la complexité de DIJSKTRA est $O(n^2)$.

6. Recherche du plus court chemin dans un DAG

Rappelons qu'un DAG est un graphe orienté sans circuit. Cette propriété peut être exploitée pour calculer les plus courts chemins en ne relâchant chaque arc qu'une seule fois. Contrairement à l'algorithme de DIJSKTRA, nous n'avons plus de précondition sur les coûts des arcs qui pourront être positifs ou négatifs. Comme pour DIJSKTRA, l'idée est de relâcher les arcs partant d'un sommet s_i dès lors que $d[s_i] = \delta(s_0; s_i)$. Pour cela, il suffit de ne relâcher les arcs partant d'un sommet que si tous les arcs se trouvant sur un chemin entre s_0 et s_i ont déjà été relâchés. Cet algorithme suppose que le sommet de départ s_0 ne possède pas de prédécesseur. En effet, si s_0 possède des prédécesseurs, alors il n'existe pas de chemin allant de s_0 jusqu'à ces prédécesseurs (car le graphe n'a pas de circuit) de sorte que cela n'a pas de sens de considérer ces sommets au moment de calculer les plus courts chemins partant de s_0. Cet algorithme suppose également que le sommet s_0 est le seul sommet ne possédant pas de prédécesseur, pour les mêmes raisons (s'il existe un autre sommet que s_0 n'ayant pas de prédécesseur, alors ce sommet n'est pas accessible depuis s_0).

d. Algorithme

1 **Fonction** *topoDAG*$(g, cout, s_0)$
  Entrée : Un graphe $g = (S, A)$, une fonction $cout : A \to \mathbb{R}$ et un sommet de départ $s_0 \in S$
  Précondition : g est un DAG et s_0 est le seul sommet de g ne possédant pas de prédécesseur
  Postcondition : Retourne une arborescence π des plus courts chemins partant de s_0 et un tableau d tel que $d[s_i] = \delta(s_0, s_i)$
2 **pour** *chaque sommet* $s_i \in S$ **faire**
3 $d[s_i] \leftarrow +\infty$
4 $\pi[s_i] \leftarrow null$
5 $d[s_0] \leftarrow 0$
6 Trier topologiquement les sommets de g à l'aide d'un parcours en profondeur
7 **pour** *chaque sommet* s_i *pris selon l'ordre topologique* **faire**
8 **pour** *chaque sommet* $s_j \in succ(s_i)$ **faire** relâcher$((s_i, s_j), \pi, d)$;
9 **retourne** π *et* d

Figure 12: Algorithme TOPODAG

e. *Complexité*

Soient n et p le nombre de sommets et arcs du graphe, respectivement .La complexité du tri topologique est en $O(n+p)$. L'algorithme passera exactement n fois dans la boucle lignes 7 à 8, et chaque arc sera relâché très exactement une fois. Donc, la complexité de l'algorithme 12 est en $O(n+p)$.

7. Algorithme de BELLMAN-FORD

a. *Principe*

L'algorithme précédemment étudier ne peut être utilisé que si le graphe ne comporte pas de circuit. Quand le graphe comporte des circuits, nous pouvons utiliser l'algorithme de DIJSKTRA, mais il faut dans ce cas que la fonction coût soit telle que l'ajout d'un arc à la fin d'un chemin ne puisse pas dégrader le coût du chemin. Considérons par exemple le graphe suivant :

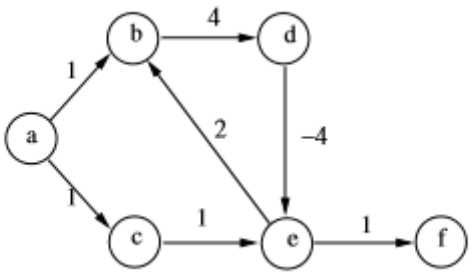


Figure 13:Graphe Exemple

Les coûts de certains arcs sont négatifs. Si le coût d'un chemin est défini par la somme des coûts de ses arcs, alors l'ajout d'un arc de coût négatif à la fin d'un chemin diminue le coût du chemin. Si nous exécutons l'algorithme de DIJSKTRA sur ce graphe, à partir du sommet a, l'algorithme va trouver le chemin $< \mathrm{a; c; e; f} >$ pour aller de a à f (car il relâchera les arcs partant de e avant de relâcher les arcs partant de d), alors qu'il existe un chemin plus court ($< \mathrm{a, b, d, e, f} >$). Nous ne pouvons pas non plus appliquer TopoDAG car le graphe comporte un circuit. L'algorithme de Bellman-Ford permet de trouver les plus courts chemins à origine unique dans le cas où le graphe contient des arcs dont le coût est négatif, sous réserve que le graphe ne contienne pas de circuit absorbant (dans ce cas, l'algorithme de Bellman-Ford va détecter l'existence de circuits absorbants). L'algorithme fonctionne selon le même principe que les deux algorithmes précédents, en grignotant les bornes d par des relâchements d'arcs.

Cependant, chaque arc va être relâché plusieurs fois : à chaque itération, tous les arcs sont relâchés.

b. Algorithme

Fonction *Bellman-Ford*$(g, cout, s_0)$
  Entrée : Un graphe $g = (S, A)$, une fonction $cout : A \to \mathbb{R}$ et un sommet de départ $s_0 \in S$
  Postcondition : Retourne une arborescence π des plus courts chemins partant de s_0 et un tableau d tel que $d[s_i] = \delta(s_0, s_i)$. Affiche un message si g contient un circuit absorbant
  pour *chaque sommet* $s_i \in S$ **faire**
    $d[s_i] \leftarrow +\infty$
    $\pi[s_i] \leftarrow null$
  $d[s_0] \leftarrow 0$
  pour k *variant de* 1 *à* $|S| - 1$ **faire**
    pour *chaque arc* $(s_i, s_j) \in A$ **faire** relâcher$((s_i, s_j), \pi, d)$;
  si $\exists$ *un arc* $(s_i, s_j) \in A$ *tel que* $d[s_j] > d[s_i] + cout(s_i, s_j)$ **alors**
    afficher("Le graphe contient un circuit absorbant")
  retourne π *et* d

Figure 14: Algorithme Bellman-Ford

c. Complexité

Si le graphe comporte n sommets et p arcs, chaque arc sera relâché n fois, de sorte que l'algorithme effectuera np appels à la procédure de relâchement. Par conséquent, la complexité est en$O(np)$.

III. Arbres couvrant minimaux

Un arbre couvrant minimal (Minimal Spanning Tree / MST) d'un graphe non orienté $G = (S, A)$ est un graphe partiel $G_0 = (S, A_0)$ de G tel que G_0 est un arbre couvrant (autrement dit, G_0 est connexe et sans cycle), et la somme des coûts des arêtes de A_0 est minimale.

1. Principe Générique

Dans cette partie, nous allons étudier deux algorithmes permettant de calculer des MST. Les deux algorithmes fonctionnent selon un principe glouton décrit dans générique : l'idée est de sélectionner, à chaque itération, une arête de coût minimal traversant une coupure. Une coupure d'un graphe $G = (S, A)$ est une partition de l'ensemble des sommets en deux parties $(P, S \backslash P)$. Une arête (s_i, s_j) traverse une coupure $(P, S \backslash P)$. si chaque extrémité de l'arête

appartient à une partie différente, i.e., $s_i \in P$ et $s_j \in S \backslash P$, ou $s_j \in$ P et $s_i \in$ S \P. Une coupure respecte un ensemble d'arêtes E si aucune arête de E n'est traversée par la coupure.

Considérons par exemple le graphe suivant :

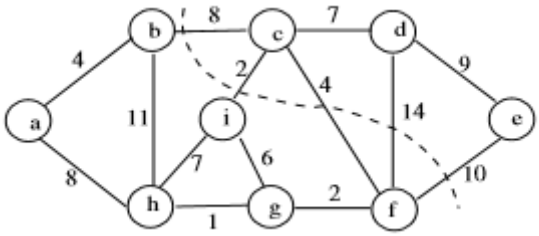


Figure 15: Graphe Illustration MST

La coupure $(\{a, b, f, g, h, i\}, \{c, d, e\})$ est représentée en pointillés et traverse les arêtes{b, c}, {i, c}, {c, f}, {d, f}et{e, f}. L'arête de coût minimal traversant cette coupure est{i, c}.

d. *Algorithme du principe de glouton Générique pour calculer un MST*

1 **Fonction** *MSTgénérique*$(g, cout)$
  Entrée : Un graphe $g = (S, A)$ et une fonction $cout : A \to \mathbb{R}$
  Postcondition : Retourne un ensemble d'arêtes $E \subseteq A$ tel que (S, E) est un MST de g
2 $E \leftarrow \emptyset$
3 **tant que** $| E | < | S | - 1$ **faire**
4 Soit $(P, S \setminus P)$ une coupure (quelconque) qui respecte E
5 Ajouter dans E une arête de coût minimal traversant la coupure $(P, S \setminus P)$
6 **retourne** E

Figure 16: Algorithme Générique MST

2. Algorithme de KRUSKAL

a. *Principe*

L'algorithme de KRUSKAL trouve une arête de poids le plus faible possible qui relie deux arbres de la forêt. Il utilise l'algorithme générique décrit plus haut car il trouve un arbre couvrant minimum pour un graphe pondéré connecté en ajoutant des arcs de coût croissants à chaque étape. Cela signifie qu'il trouve un sous-ensemble des arêtes qui forme un arbre qui inclut chaque sommet, où le poids total de toutes les arêtes de l'arbre est minimisé.

b. Algorithme

1 **Fonction** *Kruskal*$(g, cout)$
  Entrée : Un graphe $g = (S, A)$ et une fonction $cout : A \to \mathbb{R}$
  Postcondition : Retourne un ensemble d'arêtes $E \subseteq A$ tel que (S, E) est un MST de g
  Déclaration : Une arborescence π
  Un vecteur p tel que $p[r_i]$ est une borne supérieure de la profondeur de l'arbre de racine r_i
2 **pour** *chaque sommet* $s_i \in S$ **faire**
3 $\pi[s_i] \leftarrow null$
4 $p[s_i] \leftarrow 0$
5 $E \leftarrow \emptyset$
6 Trier les arêtes de A par ordre de coût croissant
7 **pour** *chaque arête* $\{s_i, s_j\}$ *prise par ordre de coût croissant* **et tant que** $|E| < |S| - 1$ **faire**
8 $r_i \leftarrow$racine(π, s_i)
9 $r_j \leftarrow$racine(π, s_j)
10 **si** $r_i \neq r_j$ **alors**
11 Ajouter (s_i, s_j) dans E
12 **si** $p[r_i] < p[r_j]$ **alors** $\pi[r_i] \leftarrow r_j$;
13 **sinon si** $p[r_j] < p[r_i]$ **alors** $\pi[r_j] \leftarrow r_i$;
14 **sinon** $\pi[r_j] \leftarrow r_i$; $p[r_i] \leftarrow p[r_i] + 1$;
15 **retourne** E
16 **Fonction** *racine*(π, s)
  Entrée/Sortie : Une forêt π
  Entrée : Un sommet s
  Postcondition : Retourne la racine r de l'arborescence contenant s et la met à jour de sorte que tous les sommets se trouvant entre s et r soient directement rattachés sous r
17 **si** $\pi[s] = null$ **alors retourne** s;
18 $\pi[s] \leftarrow$ racine$(\pi, \pi[s])$
19 **retourne** $\pi[s]$

Figure 17: Algorithme de KRUSKAL

c. Complexité

Soient n et p le nombre de sommets et arêtes, respectivement. Le tri des arêtes peut être fait en $O(plogp)$, à l'aide d'un **quicksort**, **mergesort** ou **heapsort**, par exemple. Si les coûts ont des valeurs entières comprises dans un intervalle [min, max], il est possible d'utiliser un tri par dénombrement en $O(p + k)$ où $k = max - min1$. La boucle lignes 7 à 14 est exécutée p fois dans le pire des cas. À chaque fois, il faut remonter des sommets s_i et s_j jusqu'aux racines des arbres correspondants. En rattachant directement les sommets visités sous la racine (dans la fonction racine), et en rattachant l'arbre le moins profond sous l'arbre le plus profond, cette opération peut être faite en$O(logp)$ (voir le livre de Cormen, Leiserson et Rivest pour plus de détails sur la gestion de π). Par conséquent, la complexité de l'algorithme de KRUSKAL est $O(plogp)$.

d. Exemple

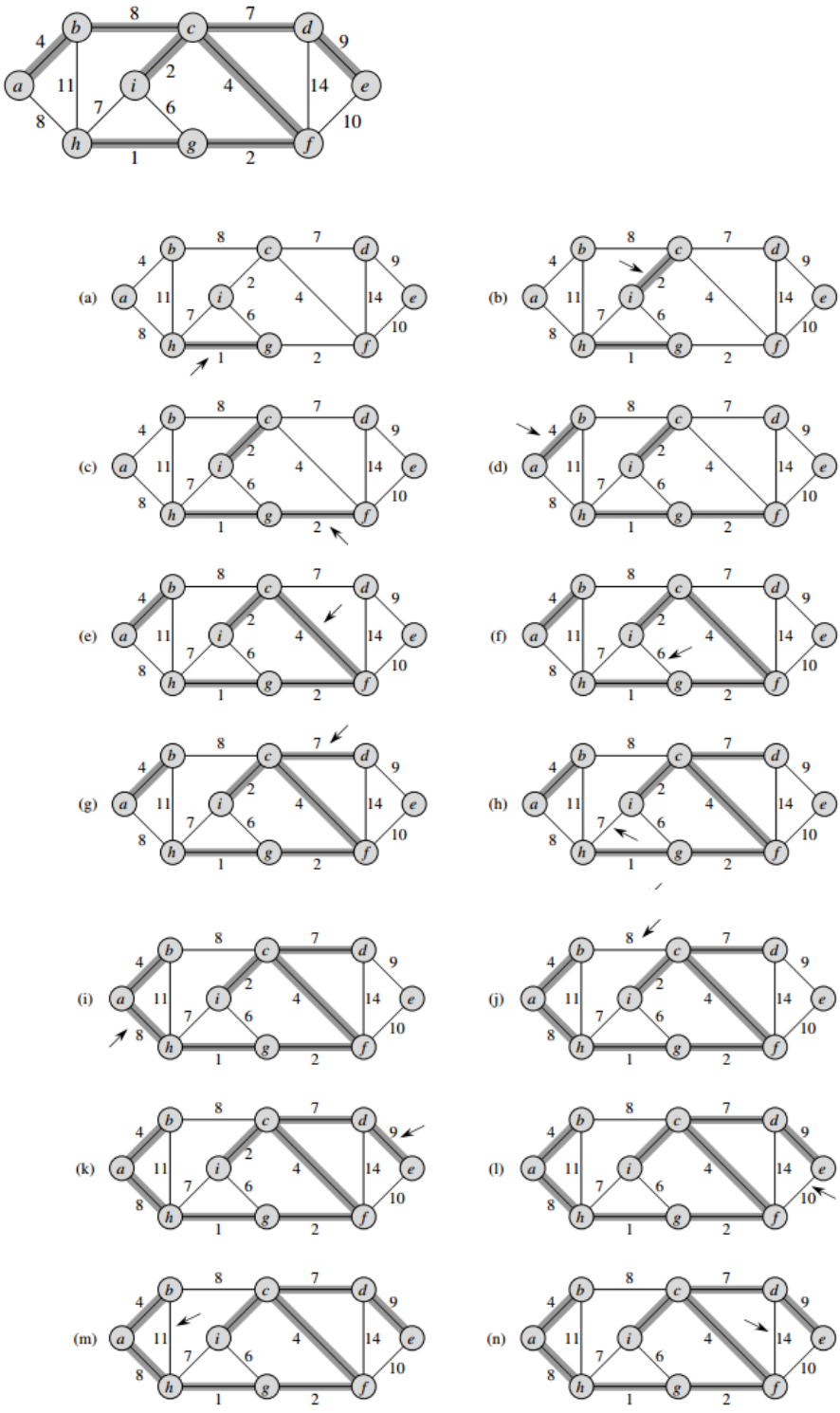


Figure 18: Illustration KRUSKAL

3. Algorithme de PRIM

a. *Principe*

Il ressemble à l'algorithme de DIJSKTRA de recherche des plus courts chemins d'un graphe. Le sommet de départ de l'algorithme sera choisi arbitrairement, L'algorithme consiste à faire croître un arbre depuis un sommet. On commence avec un seul sommet puis à chaque étape, on ajoute une arête de poids minimum ayant exactement une extrémité dans l'arbre en cours de construction. En effet, si ses deux extrémités appartenaient déjà à l'arbre, l'ajout de cette arête créerait un deuxième chemin entre les deux sommets dans l'arbre en cours de construction et le résultat contiendrait un cycle.

Pour implémenter efficacement l'algorithme de PRIM, l'important est de faciliter la sélection de la nouvelle arête à ajouter à l'arbre constitué des arêtes de A'. Dans le pseudo code qui suit, le graphe connexe G et la racine s de l'arbre couvrant minimum à construire sont les entrées de l'algorithme. Pendant l'exécution, tous les sommets qui n'appartiennent pas à l'arbre se trouvent dans une file de priorités min F basée sur un champ clé. Pour chaque sommet v, clé[v] est le poids minimal d'une arête reliant v à un sommet de l'arbre ; par convention, W[v]=∞ si une telle arête n'existe pas. Le champ pred[v] désigne le parent de v dans l'arbre. Pendant le déroulement de l'algorithme, l'ensemble A' de MST-GÉNÉRIQUE est conservé implicitement sous la forme

$E = \{(v, p[v]) : v \in S - \{r\} - F\}$.Quand l'algorithme se termine, la file de priorités min F est vide ; l'arbre couvrant minimum de G est donc $E = \{(v, p[v]) : v \in S - \{s\}\}$.

b. *Algorithme*

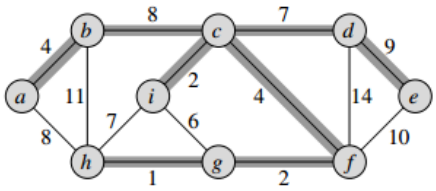
1 **Fonction** *Prim*$(g, cout)$
  Entrée : Un graphe $g = (S, A)$ et une fonction $cout : A \to \mathbb{R}$
  Postcondition : Retourne un ensemble d'arêtes $E \subseteq A$ tel que (S, E) est un MST de g
2 Soit s_0 un sommet de S choisi arbitrairement
3 $P \leftarrow \{s_0\}$
4 $E \leftarrow \emptyset$
5 **pour** *chaque sommet* $s_i \in S$ **faire**
6 **si** $s_i \in adj(s_0)$ **alors** $\pi[s_i] \leftarrow s_0 ; c[s_i] \leftarrow cout(s_0, s_i)$;
7 **sinon** $\pi[s_i] \leftarrow null ; c[s_i] \leftarrow \infty$;
8 **tant que** $P \neq S$ **faire**
9 Ajouter dans P le sommet $s_i \in S \setminus P$ ayant la plus petite valeur de c
10 Ajouter $\{s_i, \pi[s_i]\}$ à E
11 **pour** *chaque sommet* $s_j \in adj(s_i)$ **faire**
12 **si** $s_j \notin P$ **et** $cout(s_i, s_j) < c[s_j]$ **alors**
13 $\pi[s_j] \leftarrow s_i$
14 $c[s_j] \leftarrow cout(s_i, s_j)$
15 **retourne** E

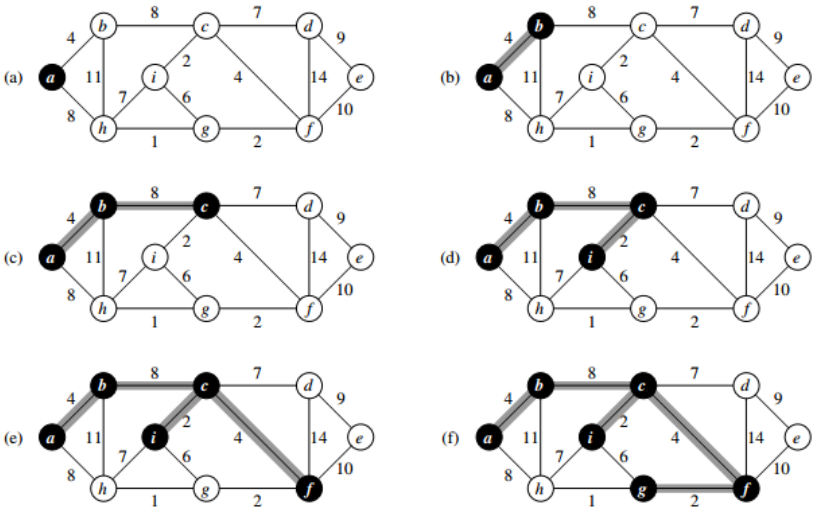
Figure 19: Algorithme de PRIM

c. *Complexité*

Soient n et p le nombre de sommets et arêtes, respectivement. L'algorithme passe $n-1$ fois dans la boucle lignes 8 à 14 (initialement $P = \{s_0\}$, un sommet est ajouté à P à chaque passage, et l'itération s'arrête lorsque $P = S$). À chaque passage, il faut chercher le sommet de $S\backslash P$ ayant la plus petite valeur de c puis parcourir toutes les arêtes adjacentes à ce sommet. Si les sommets de S \P sont mémorisés dans un tableau ou une liste, la complexité est O(n2). Cette complexité peut être améliorée en utilisant une file de priorité, implémentée par un tas binaire, pour gérer l'ensemble $S \backslash P$. Dans ce cas, la recherche du plus petit élément de S \P est faite en temps constant. En revanche, à chaque fois qu'une valeur du tableau c est diminuée, la mise à jour du tas est en $O(logn)$. Comme il y a au plus p mises à jour de c (une par arête), la complexité de PRIM devient $O(plogn)$.

d. *Exemple*





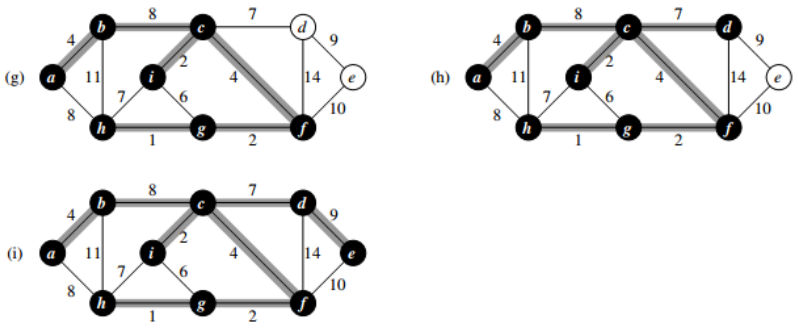


Figure 20: Illustration PRIM

e. Application des MST

Les arbres couvrant minimaux ont des applications directes dans la conception des réseaux, y compris les réseaux informatiques, les réseaux de télécommunications, réseaux de transport, les réseaux d'approvisionnement en eau et les réseaux électriques (qu'ils ont d'abord été inventées pour, comme mentionné ci - dessus). Ils sont invoqués comme sous-programmes dans les algorithmes pour d'autres problèmes, y compris l' algorithme de Christofides pour l'approximation du problème du voyageur de commerce , approchant le problème de coupe minimale multi-terminal (qui est équivalent dans le cas à un seul terminal au débit maximum problème),et l'approximation de l' appariement parfait pondéré au coût minimum .

D'autres applications pratiques basées sur des arbres couvrants minimaux incluent :

- Analyse de cluster: clustering de points dans le plan, single-linkage clustering (une méthode de clustering hiérarchique), graph-theoretic clustering, et clustering de données d'expression génique.
- Construire des arbres pour la diffusion dans les réseaux informatiques.
- Enregistrement d'images et segmentation – voir segmentation minimale basée sur l'arbre couvrant.
- Extraction de caractéristiques curvilignes en vision par ordinateur.
- Reconnaissance de l'écriture manuscrite d'expressions mathématiques.
- Conception de circuits : mise en œuvre de multiplications constantes multiples efficaces, telles qu'utilisées dans les filtres à réponse impulsionnelle finie.
- Observabilité topologies dans les réseaux électriques.

IV. Les Flots dans les graphes

On considère un graphe orienté G tel que :

- Il existe un ensemble disjoint de sommets S *et* P, respectivement appelés source et puit.
- Chaque arête e est value par une capacité $c(e)$ correspondant au flux maximal pouvant traverser sur cette arête.

1. Problème

Quel est le flux maximal pouvant aller des sources vers les puits sans pertes intermédiaire ?

Dans le cas d'un graphe non-dirigé on remplace chaque arête par deux arête dirigées opposées de mêmes capacité que l'arête initiale.

2. Optimisation de flot

La valeur $|\varphi|$ d'un flot φ est definie comme la somme des flots sortants de la source s (ce qui est toujours égal à la somme des flots entrants de la destination t)

3. Recherche d'un flot Maximal : Algorithme de FORD-FULKERSON

a. Algorithme générique

Soit un réseau R, un flot compatible $\emptyset$ dans R et le graphe d'écart $R^e(\emptyset)$. Pour déterminer un flot maximum, l'algorithme générique consiste, à chaque itération, à chercher un chemin μ allant de s à t dans $R^e(\emptyset)$. Si un tel chemin existe, alors, on augmente le flot $\emptyset$ de la quantité $\delta = min_{u\epsilon\mu} c'_u$. Sinon l'algorithme termine et $\emptyset$ est le flot de la valeur maximale.

$\phi \leftarrow 0$
Tant que ($R^e(\phi)$ contient un chemin de s à t) **faire**
    Identifier un chemin μ de s à t
    $\delta = min_{u\in\mu} c'_u$
    Augmenter de δ unités le flot ϕ sur R
    Mettre à jour $R^e(\phi)$
Fait

Figure 21:Algorithme Générique Ford-Fulkerson

Cet algorithme ne précise pas de quelle façon déterminer un chemin μ de s à t. Dans la section suivante, nous présenterons un algorithme de marquage qui sans passer par le graphe d'écart, permet d'exhiber un chemin augmentant dans $R^e(\emptyset)$ (ou de façon équivalente une chaîne augmente dans R) en travaillant directement sur R.

b. *Algorithme de Marquage de FORD-FULKERSON*

$marque(t) \leftarrow 1$
Tant que $(marque(t) \neq 0)$ **faire**
  $marque(i) \leftarrow 0 \quad \forall i \in N$
  $pred(i) \leftarrow i \quad \forall i \in N$
  $marque(s) \leftarrow +\infty$ et $LISTE \leftarrow \{s\}$
  Tant que $(LISTE \neq \emptyset$ et $marque(t) = 0)$ **faire**
    Sélectionner $i \in LISTE$ et faire $LISTE \leftarrow LISTE \backslash \{i\}$
    Pour chaque $u = (i, j) \in \omega^+(i)$ **faire**
      Si $(marque(j) = 0$ et $c_u > \phi_u)$ **Alors**
        $marque(j) \leftarrow c_u - \phi_u$
        $pred(j) \leftarrow i$
        $LISTE \leftarrow LISTE \cup \{j\}$
      Fin Si
    Fin Pour
    Pour chaque $u = (j, i) \in \omega^-(i)$ **faire**
      Si $(marque(j) = 0$ et $\phi_u > 0)$ **Alors**
        $marque(j) \leftarrow -\phi_u$ (marquage de type -)
        $pred(j) \leftarrow i$
        $LISTE \leftarrow LISTE \cup \{j\}$
      Fin Si
    Fin Pour
  Fait
  Si $marque(t) \neq 0$ alors **augmenter**
Fait

augmenter
Identifier la chaîne μ à l'aide de $pred(i)$: $\mu = (s = i_0, i_1, \ldots, i_k = t)$
$\delta \leftarrow min_{i \in \mu} |marque(i)|$
$\phi_{s,i_0} \leftarrow \phi_{s,i_0} + \delta$
Pour j **de** 1 **à** k **faire**
  Si $(marque(i_j) < 0)$ **Alors**
    $\phi_{i_{j-1},i_j} \leftarrow \phi_{i_{j-1},i_j} - \delta$
  Sinon
    $\phi_{i_{j-1},i_j} \leftarrow \phi_{i_{j-1},i_j} + \delta$
  Fin Si
Fin Pour
$\phi_{i_k,t} \leftarrow \phi_{i_k,t} + \delta$

Figure 22: Algorithme Ford-Fulkerson

c. *Complexité*

Chaque itération comporte $O(m)$ opérations élémentaires car la méthode de marquage examine chaque arc et chaque sommet au plus une fois. En conséquence, la complexité est en $O(m)$ fois le nombre d'augmentation. Si chaque capacité est entière et bornée par U, la capacité d'une coupe $(s-t)$est au plus nU. Et la complexité totale est en $O(nmU)$. En conséquence, si par exemple $U = 2^n$, la complexité de l'algorithme est en $O(nm2^n)$, c'est-à-dire exponentielle sur le nombre de nœuds du réseau.

Chapitre III : Quelques Problèmes NP-Difficiles sur les Graphes

I. Recherche de cliques

Etant donné un graphe non orienté $G = (S, A)$, une clique est un sous ensemble de sommets $S' \subseteq S$ qui sont tous connectés 2 à 2 par des arêtes de sorte que :

$$\forall (i,j) \epsilon\, S' \times S', i \neq j \;\Rightarrow \{i;j\} \epsilon A$$

Autrement dit, une clique est un sous-graphe complet.

Le problème de la clique est spécifié de la façon suivante :

- ✓ Entrée : un graphe non orienté $G = (S, A)$ et un entier positif k
- ✓ Question : G contient-il une clique de k-sommets ?

a. *Complexité du problème de la clique*

Théorème : Le problème de la clique est $NP - complet$

Démonstration : Le problème de la clique appartient à la classe NP car nous pouvons vérifier en temps polynomial qu'un sous-ensemble $S' \subseteq S$ de k sommets est bien une clique. Il suffit pour cela de vérifier que pour toute paire de sommets $\{s_i, s_j\} \subseteq S'$, il existe une arête (s_i, s_j) dans A.

II. Coloriage de graphes

Le coloriage de graphe non orienté G consiste à attribuer une couleur à chaque sommet de G, de telle sorte qu'une même couleur ne soit pas attribuée à deux sommets adjacents (reliés par une arête). Plus précisément, étant donné un ensemble C de couleurs, le coloriage d'un graphe non orienté $G = (S, A)$ est une fonction $c: S \longrightarrow C$ telle que $\forall (s_i, s_j) \epsilon\, A, c(s_i) \neq c(s_j)$. Le nombre chromatique de G noté $X(G)$, est la cardinalité du plus petit ensemble de couleurs C permettant de colorier G.

Le problème de k coloriage consiste à déterminer s'il est possible de colorier un graphe avec au plus k couleurs ou, autrement dit si $X(g) \leq k$. Ce problème est spécifié de la façon suivante :

- ✓ Entrée : un graphe non orienté $G = (S, A)$ et un entier k positif
- ✓ Question : G peut-il être colorié avec k couleurs ?

Ce problème est $NP - Complet$

a. Relation entre coloriage et clique

Pour tout graphe G, $X(G)$ est supérieur ou égal au nombre de sommet de la clique maximum de G.

En effet tous les sommets d'une clique sont connectés deux a deux de sorte qu'ils doivent avoir tous des couleurs différentes. Par conséquent, pour toute clique c, $X(G)$ est supérieur ou égal au nombre de sommets de c.

b. Théorème des quatre couleurs

Un graphe planaire est un graphe qui peut être dessiné sur un plan de telle sorte qu'aucune arête ne croise une autre arête (en dehors des extrémités des arêtes). Francis Guthrie à conjecturer en 1852 que le nombre chromatique d'un graphe est inférieur ou égal à quatre (4). Cette conjecture a été démontrer en 1976.

Une conséquence immédiate de cette propriété est qu'il est toujours possible de colorier les pays d'une carte géographique avec seulement quatre (4) couleurs de telle sorte que deux pays voisins soient coloriés avec des couleurs différentes.

c. Algorithme de BRELAZ

L'algorithme de BRELAZ également appelé DSATUR, est un algorithme glouton qui permet de calculer une bonne supérieure $borneX$ de $x(G)$

1 **Fonction** *brélaz*(g)
  Entrée : Un graphe $g = (S, A)$
  Postcondition : retourne une borne supérieure de $\chi(g)$
2 $borne\chi \leftarrow 0$
3 $N \leftarrow S$
4 **pour** *chaque sommet* $s_i \in S$ **faire**
5 $couleurs Voisines[s_i] \leftarrow \emptyset$
6 $nb VoisinsNonColories[s_i] \leftarrow |adj(s_i)|$
7 **tant que** $N \neq \emptyset$ **faire**
8 Soit X l'ensemble des sommets $s_j \in N$ tels que $|couleurs Voisines[s_i]|$ soit maximal
9 Choisir un sommet s_i de X tel que $nb VoisinsNonColories[s_i]$ soit maximal
10 Enlever s_i de N
11 **si** $|couleurs Voisines[s_i]| = borne\chi$ **alors** $borne\chi \leftarrow borne\chi + 1$;
12 Soit k la plus petite valeur comprise entre 1 et $borne\chi$ telle que $k \notin couleurs Voisines[s_i]$
13 $couleur[s_i] \leftarrow k$
14 **pour** *chaque sommet* $s_j \in adj(s_i) \cap N$ **faire**
15 $nb VoisinsNonColories[s_j] \leftarrow nb VoisinsNonColories[s_j] - 1$
16 $couleurs Voisines[s_j] \leftarrow couleurs Voisines[s_j] \cup \{couleur[s_i]\}$
17 **retourne** $borne\chi$

Figure 23: Algorithme de BRELAZ

d. *Complexité de l'algorithme de BRELAZ*

Soit $n = |S|$ et $p = |A|$. L'algorithme passe n fois dans la boucle ligne 7 à 16. A chaque passage dans la boucle, l'algorithme parcourt la liste des voisins du sommet colorié s_i. La complexité de l'algorithme est $O(n + p)$ (en utilisant un tableau de booléens permettant de déterminer en temps constant si un sommet voisin d'un sommet donné utilise déjà une couleur donnée).

III.Le voyageur de commerce

Un cycle hamiltonien d'un graphe non orienté est un cycle passant par chacun d ses sommets exactement une fois.

Considérons maintenant un graphe non orienté $G = (S, A)$ muni d'une fonction $c: A \rightarrow \mathbb{R}^+$ associant un coût à chacune de ses arêtes, et définissons le coût d'un cycle par la somme des coûts de ses arêtes. Le problème du voyageur de commerce consiste à rechercher dans G un cycle hamiltonien de coût minimal. Quand le graphe est orienté, de sorte que le coût de l'arc (i, j) peut être different du coût de l'arc (j, i), le problème est appelé : **le voyageur de commerce asymétrique.**

Théorème : le problème de décision associé au voyageur de commerce, visant si un graphe donnée contient un cycle hamiltonien de coût inférieur ou égal à une borne k donné, est $NP - complet$.

Démonstration : Le problème appartient à la classe NP car nous pouvons vérifier en temps polynomial qu'un cycle donné est hamiltonien et est de coût inférieur ou égal à k.

Pour montrer qu'il est $NP - Complet$, nous allons décrire une procedure permettant de transformer une instance du problème de recherche de cycle hamiltonien dans un graphe G en une instance du problème du voyageur de commerce. Nous définissons pour cela la fonction coût telle que toutes les arêtes de G ont un même coût égal à 1. Nous pouvons facilement vérifier qu'il existe une solution au problème du voyageur de commerce pour $k = |S|$ si et seulement si G admet un cycle hamiltonien.

Conclusion

Parvenu au terme de notre travail sur les algorithmes de graphe ; il a été question pour nous de présenter d'une part les généralités sur les graphes, par la suite de faire une étude un peu plus détaillée de quelques algorithmes appliqués aux graphes a l'instar des algorithmes de parcours, les algorithmes de recherche des plus cours chemins, la notion d'arbre couvant minimum et les algorithmes a elle appliqué et les algorithmes des flots maximum.il ressort donc de cette étude que les graphes permette de résoudre de nombreux problème dans différents domaines notamment le routage du trafic dans les réseaux d'ordinateurs et de télécommunications, l'affectation des ressources et bien d'autres encore. Une étude plus approfondie des graphes nous permet de relever que certain problème des graphes fait partie de la classe des problèmes dit NP-complet.

Partie II : TRAVAUX DIRIGES