Chapitre 2 : Problèmes d'ACM et de cheminement

1. Rappel : Arbre, forêt et arborescence

Définition 1. On appelle **arbre** un graphe (non orienté) connexe sans cycle.

Théorème Les propriétés suivantes sont équivalentes :

- G est connexe sans cycle. (G est un arbre.)
- G est sans cycle comportant n−1 arêtes.
- G est sans cycle, et si l'on rajoute une arête alors on obtient un cycle et un seul.
- G est connexe, et la suppression d'une arête fait apparaître 2 composantes connexes.
- Il existe une chaîne et une seule entre toute paire de sommets de G.

Def 2. un graphe partiel H=(X,W) est un **co-arbre** de G s'il ne contient pas de cocycles de G=(X,U), mais contient un cocycle si l'on ajoute n'importe quel arc u ∈ U\W.

Déf 3. On appelle **forêt** un graphe sans cycle (pas nécessairement connexe) dont chaque composante connexe est un arbre.

Def 4. Une **co-forêt** est un graphe partiel de G qui a un co-arbre de G pour chaque composante connexe de G.

Def 5. On appelle **arbre couvrant de G** tout graphe partiel de G définissant un arbre connectant tous les sommets de G.

Def 6. Soit G = (S, A) un graphe non orienté.

Un graphe est dit **quasi-fortement connexe** si à toute paire de sommets (x,y) on peut associer un sommet z tel qu'il existe un chemin de z vers x et un chemin de z vers y. (x, y et z ne sont pas nécessairement distincts.)

Remarque Un graphe fortement connexe est quasi-fortement connexe.

Déf 7. On appelle **racine** d'un graphe orienté un sommet R (s'il existe) tel que pour tout sommet x de G il existe un chemin allant de R vers x.

Proposition Un graphe est quasi-fortement connexe si et seulement si il possède une racine.

Déf 8. On appelle **arborescence** un graphe orienté G vérifiant l'une des propriétés suivantes :

- G est quasi-fortement connexe et sans circuit
- G est quasi-fortement connexe et possède n−1 arêtes
- G est quasi-fortement connexe et cesse de l'être si on supprime un arc quelconque

Propriété Un graphe orienté G = (S, A) admet au moins une arborescence (couvrante) comme graphe partiel si et seulement si G est quasi-fortement connexe.

2. Le problème de l'arbre couvrant minimal (ACM)

Soit G = (S,A,V) un graphe non orienté, connexe et valué. V = {v(i,j)/v(i,j) = coût de l'arête (i,j)} Le problème de **l'arbre couvrant minimal** de G consiste à trouver un arbre couvrant de G dont le coût total des arêtes est minimal. Si G n'est pas connexe, on peut calculer une forêt couvrante minimale.

5.1. Méthode de Kruskal

Principe

Soit G = (S,A,V), S = {1,2,...,n}

- Trier les arêtes par ordre croissant de coût.
- Construire une forêt composée de n sommets (initialisation).
- À chaque itération, on rajoute à cette forêt la plus petite arête ne créant pas de cycle avec celles déjà choisies.
- On arrête les itérations lorsque l'arbre contient (n−1) arêtes.

Algorithme

```
1: procedure Kruskal
2:   ∀i ∈ E, cc(i) ← i
3:   Trier les arêtes
4:   T ←∅
5:   pour k = 1 à n−1 faire
6:       Choisir une arête (x,y)
7:       si cc(x) 6= cc(y) alors
8:           T = T ∪{(x,y)}
9:           pour tout sommet i faire
10:                si cc(i) = cc(x) alors
11:                    cc(i) ← cc(y)
```

12: **fin si**
13: **fin pour**
14: **fin si**
15: **fin pour**
16: **fin procedure**

5.2. Méthode de Prim

Principe Soit T l'arbre en cours de construction. À chaque itération, on rajoute à T un sommet et une arête. Soit R l'ensemble des sommets pas encore dans T. à chaque sommet x de R, on associe le sommet y de T dont la distance à x est minimale. Le coût v(x,y) est appelé distance de x à T, notée d(x,T). On choisit de faire entrer dans T le sommet x de R dont la distance d(x,T) à T est minimale.

Algorithme

fonction $T = \mathsf{prim}(G)$
  $poids =$ poids total de l'arbre couvrant (initialisé à 0)
  marquer le sommet 1
  tant que il reste des sommets non-marqués **faire**
    $\{x; y\} =$ arête de coût minimal joignant un sommet marqué x et un sommet non-marqué y
    marquer le sommet y et ajouter l'arête $\{x; y\}$ à T
    $poids = poids + W(x, y)$
  fin faire

1. Problème du plus court chemin

Le problème d'optimisation suivant est celui du chemin optimal. Ici nous avons besoin d'étudier une notion plus générale de longueur qui puisse être appliquée à un graphe valué.

Définition 1.1. (Longueur et distance) Dans un graphe orienté valué G = (S,A,f) on appellera longueur d'un chemin C = (x_0,x_1,…,x_{p-1},x_p) relativement à *f* la valeur

$$\text{Longueur}_f(C) = \sum_{i=0}^{p-1} f(x_i, x_{i+1})$$

on appellera distance de x à y par rapport à *f* la longueur (relativement à *f*) du plus court chemin de x à y

Distmin(x,y) = min C=(x,…,y)

Longueurf(C), et Distmax(x,y) = max C=(x,…,y)
Longueurf(C)

Proposition 1.1. (existence du chemin optimal) Dans un graphe orienté valué G = (S,A,*f*) il existe un plus court (resp. long) chemin entre tout couple de sommets si et seulement si il n'existe pas de circuit de longueur négative (resp. positive) relativement à *f*.

Définition 1.2. (algorithme de Bellman-Ford-Kalaba) Soit un graphe orienté valué G = (S,A,f), d'ordre n et de taille m, et x un sommet de G. L'algorithme de Bellman calcule deux matrices de taille 1 × n

• Dist matrice des distances telle que Dist(y) = distance optimale de x à y

• Pred matrice des prédécesseurs telle que Pred(y) = prédécesseur de y dans le chemin optimal depuis x Pour le plus court chemin l'algorithme s'écrit :

```
fonction [Dist, Pred] = BELLMAN(G, s)
  Initialisation : n = nombre de sommets de G
  Pred = tableau des prédécesseurs initialisé à 0
  Dist = tableau des distances initialisé à +∞ (sauf Dist(s) = 0)
  W = matrice des poids des arcs (∞ si l'arc n'existe pas)
  Traitement : k = 1
  tant que k ≤ n et il y a eu des modifications à l'étape précédente faire
        pour tout sommet x faire
              pour tout y successeur de x faire
                    si Dist(x) + W(x, y) < Dist(y)
                      alors modifier Dist(y) et Pred(y) = x
                    fin
              fin faire
              k = k + 1
        fin faire
```

L'algorithme de Bellman-Ford-Kalaba reste encore coûteux et complexe. Dans de nombreux cas on peut simplifier la recherche d'un chemin optimal à condition que le graphe possède certaines propriétés. Le premier exemple d'une telle situation est l'algorithme de Dijkstra, que l'on peut utiliser pour la recherche de chemins minimaux dans un graphe à valuations positives.

Définition 1.3. (algorithme de Dijkstra-Moore) Soit un graphe orienté valué G = (S,A,f), d'ordre n et de taille m, et x un sommet de G. L'algorithme de Dijkstra calcule deux matrices de taille 1 × n

• Dist matrice des distances telle que Dist(y) = distance optimale de x à y

• Pred matrice des prédécesseurs telle que Pred(y) = prédécesseur de y dans le chemin optimal depuis x Pour le plus court chemin l'algorithme s'écrit :

```
fonction [Dist, Pred] = DIJKSTRA(G, s)
  Initialisation :
  n = nombre de sommets de G
  Pred = tableau des prédécesseurs initialisé à 0
  Dist = tableau des distances initialisé à +∞ (sauf Dist(s) = 0)
  W = matrice des poids des arcs (∞ si l'arc n'existe pas)
  C = {1; 2; ...; n} (liste des sommets restant à traiter)
  D = ∅ (liste des sommets déjà traités)
  Traitement :
  tant que C ≠ ∅ faire
        x = sommet de C le plus proche de s
        retirer x de C et le mettre dans D
        pour tout sommet y ∈ C faire
              si Dist(x) + W(x, y) < Dist(y)
                 alors modifier Dist(y) et Pred(y) = x
              fin
          fin faire
        fin faire
```

L'algorithme de Dijkstra a un temps d'exécution assez rapide (~ n^2) mais a deux défauts:

• il ne s'applique qu'aux graphes à valuations positives

• il ne marche que pour trouver les plus courts chemins

Définition 1.4. (algorithme de Bellman simplifié) Soit un graphe orienté valué G = (S,A,f), d'ordre n et de taille m, et x un sommet de G. L'algorithme de Bellman simplifié calcule deux matrices de taille 1 × n

• Dist matrice des distances telle que Dist(y) = distance optimale de x à y

• Pred matrice des prédécesseurs telle que Pred(y) = prédécesseur de y dans le chemin optimal depuis x Pour le plus court chemin l'algorithme s'écrit :

```
fonction [Dist, Pred] = BELLMAN_SIMPLE(G, s)
  Initialisation :
  Pred = tableau des prédécesseurs initialisé à 0
  Dist = tableau des distances initialisé à +∞ (sauf Dist(s) = 0)
  W = matrice des poids des arcs (∞ si l'arc n'existe pas)
  Faire la décomposition en niveau de G
  Traitement :
  pour tout k = niveau(s) + 1 jusqu'à niveau maximum faire
        pour tout sommet x du niveau k faire
              pour tout sommet y prédécesseur de x faire
                    si Dist(y) + W(y, x) < Dist(x)
                      alors modifier Dist(x) et Pred(x) = y
                    fin
              fin faire
        fin faire
  fin faire
```

l'algorithme de Bellman simplifié a un temps d'exécution assez rapide (~ n^2) mais ne s'applique qu'aux graphes décomposable en niveaux (donc sans circuits).