Problèmes de cheminement dans les Graphes

1. Graphes sans circuits

1.1 Propositions

Un graphe orienté $G=(X, U)$ est dit sans circuits (S.C.) si et seulement si toute composante fortement connexe est réduite à un sommet.
Un graphe orienté $G=(X, U)$ est dit sans circuits (S.C.) si et seulement si il est isomorphe à son graphe réduit.
Un graphe orienté $G=(X, U)$ est dit sans circuits (S.C.) si et seulement si tout chemin dans G est élémentaire.

1.2 Source et Puits

Un sommet s est appelé source si et seulement si $d^-(s)=0$.
Un sommet p est appelé puits si et seulement si $d^+(p)=0$.
Tout graphe sans circuits possède une source et un puits.
Dans un graphe sans circuits $G=(X, U)$ $\forall x \in X$, l'extrémité initiale ($\boldsymbol{s} \in X$) d'un plus long chemin vers x est une source.
Dans un graphe sans circuit $G=(X, U)$ $\forall x \in X$, l'extrémité terminale ($\boldsymbol{p} \in X$) d'un plus long chemin commençant en x est un puits.

2. Partition en niveaux d'un graphe sans circuits

2.1 Niveau d'un sommet

Soit $G=(X, U)$, un graphe orienté sans circuits. A tout $x \in X$, on associe un entier :
$v(x)$: **niveau** du sommet x et représente la longueur maximale d'un chemin élémentaire se terminant à x.
On affecte par convention à une source s la valeur $v(s)=0$.
Soit $G=(X,U)$ un graphe orienté, et soit $\lambda(G)$ la longueur du plus long chemin élémentaire dans G,
$\lambda(G)=\underset{x\in X}{Max}\ \{v(x)\}$.

2.2 Partition en niveaux :

Soit $G=(X,U)$ est un graphe sans circuits. L'ensemble des sommets X peut être partitionné au maximum en $\lambda(G)+1$ stables. Où chaque sommet de niveau i sera placé dans le stable N_i. Chaque stable N_i représente un niveau de G.

2.3 Proposition :

$G=(X,U)$ est un graphe sans circuits si et seulement si
X admet une partition $\{N_0 \cup N_1 \cup \ldots\ldots \cup N_p\} / x \in N_i \Leftrightarrow v(x)=i$

3. Graphe pondéré (Réseau)

3.1 Poids d'un arc

Soit $G=(X,U)$ un graphe orienté,
On définit $p : U \rightarrow \mathfrak{R}$ une application qui associe pour chaque arc $u \in U$ de G une valeur réelle $p(u)$ appelée poids de l'arc u. Un tel graphe est appelé graphe pondéré, graphe valué ou réseau.

3.2 Poids d'un chemin

On définit le poids d'un chemin γ comme la somme des poids des arcs de γ, $p(\gamma)=\sum_{u\in\gamma} p(u)$. On l'appelle aussi distance.

3.3 Circuit absorbant

Un circuit γ est dit absorbant si son poids est négatif ($p(\gamma)< 0$).

4. Algorithme de Bellman-Ford

4.1 Problème traité

A chaque sommet $x \in X$, on veut associer un chemin de poids optimal joignant la source du graphe $r \in X$ à x dans le réseau $R=(X, U, p)$

4.2 Principe

L'algorithme consiste à affecter à chaque sommet $x \in X$ d'un réseau $R=(X,U,p)$ sans circuits, une valeur $\pi(x)$ (appelée potentiel de x) qui représente le poids du chemin optimal reliant r à x. Il se base sur la liste des prédécesseurs de chaque sommet.

4.3 Les données

En entrée

X : ensemble des sommets
n : Nombre de sommets
U : ensemble d'arcs sous forme (x,y)
p : un vecteur de m éléments où $p[u]$ représente le poids du l'arc u dans le réseau.
r : La source du graphe.

En sortie :

π : vecteur de n éléments. Chaque entrée $\pi[x]$ représente le poids du chemin optimal de r vers x
Pré : vecteur de n éléments. Chaque entrée *Pré[x]* représente le prédécesseur de x dans le chemin optimal de r vers x.

Autres :

S : ensemble des sommets qui sont extrémité terminale d'un chemin commençant par r et pour lesquels $\pi[i]$ est calculé définitivement.

4.4 Algorithme

```
Debut
     S←{r};
     Pour tout x∈X
     Faire
          π[x]← +∞ ;
          Pré[x] ←NULL ;
     Fait
     π[r]← 0;
     Pour tout (x ∈ X-S) tel que (∀ u ∈ U avec T(u)= x on a I(u)∈S)
     Faire
           Pour tout ((y, x) ∈ U)
           Faire
                Si π[x] > π[y]+p[(y,x)]
                      Alors π[x] ← π[y]+p[(y,x)];
                            Pré[x] ← y;
                fSi
           Fait
               S ← S ∪{x} ;
      Fait
Fin
```

4.5 Remarques

- On dit que l'algorithme retourne une arborescence optimale.
- Si le chemin optimal est le chemin de poids maximal, on change dans l'algorithme « *min* » par « *max* » et « $+\infty$ » par « $-\infty$ ».
- Si $\forall x \in X\ p(x)=1$, l'algorithme calculera le plus court chemin en nombre d'arcs.
- La complexité de l'algorithme est $o(n^2)$.

- Si le graphe contient plusieurs sources ou le sommet initial *r* n'est pas la source du graphe, il est exigé de partitionner le graphe en niveaux au préalable et mettre tous les sommets de niveau ≤ *v(r)* dans *S*. A chaque itération, faire le choix du prochain sommet celui qui n'est pas dans *S* ayant le plus petit niveau.

5. Algorithme de Dijkstra

5.1 Problème traité

Le problème est le même que celui traité par Bellman, sauf qu'ici le réseau *R=(X,U,p)* peut être avec ou sans circuits mais à condition que les poids soient tous positifs ou nuls. Le sommet de départ *r* n'est pas nécessairement une source.

5.2 Principe

L'algorithme calcule le chemin de poids minimal en partant du sommet *r* et en prolongeant le chemin à chaque itération. Cette méthode s'appelle, calcul de la plus courte distance de proche en proche. A chaque étape, pour un sommet donné *x*, on ajuste les valeurs $\pi[y]$ pour tout sommet *y* successeur de *x*.

5.3 Les données

En entrée

X : ensemble des sommets
n : Nombre de sommets
U : ensemble d'arcs sous forme *(x,y)*
p : un vecteur de *m* éléments où *p*[*u*] représente le poids du l'arc *u* dans le réseau.
r : un sommet donné de *X*.

En sortie :

π : vecteur de *n* éléments. Chaque entrée $\pi[x]$ représente le poids du chemin optimal de *r* vers *x*
Pré : vecteur de *n* éléments. Chaque entrée *Pré[x]* représente le prédécesseur de *x* dans le chemin optimal de *r* vers *x*.

Autres :

f : vecteur de *n* éléments. Chaque entrée *f*[*k*] contient un sommet de *X*. le vecteur *f* à la fin sera trié dans un ordre croissant par rapport aux valeurs de $\pi[f[k]]$.
S : ensemble des sommets qui sont extrémité terminale d'un chemin commençant par *r*.
I(*u*) : donne l'extrémité initiale de l'arc *u.* *T*(*u*) : donne l'extrémité terminale de l'arc *u.*

5.4 L'Algorithme

```
Début
     s←{r};     k←1;     f[k]←r;
     Pour tout x∈X faire π[x]← +∞ ; fait
     π[r]←0 ;
     Tant que (k<n) et (π[x]<+∞)
     Faire
          Pour tout u∈U / (I(u)=f[k]) et (T(u)∉S)
          Faire
               x = T(u);
               Si (π[x] > π[f[k]]+p[u])
                     Alors π(x) ← π[f[k]]+p[u];
                          Pré[x] ← f[k];
               fSi
          Fait
          x ← y / y ∈ X-S et π[y] minimal;
          k ← k+1;
          f[k] ← x;
          S ←S ∪ {x};
     Fait
Fin
```

5.5 Remarque

- On dit que l'algorithme retourne une arborescence optimale.
- La complexité de l'algorithme est $o(m^2)$ où m est le nombre d'arcs.
- Si le graphe est non orienté, on peut associer à chaque arête $\{x,y\}$ de poids p, deux (02) arcs (x,y) et (y,x) de même poids p, puis on applique l'algorithme de Dijkstra.

6. Algorithme de Bellman-Kalaba

6.1 Problème traité

Le problème est de trouver le poids du chemin optimal à partir d'un sommet donné r vers tout autre sommet dans le graphe. Le graphe peut contenir des circuits et les poids peuvent être positifs nuls ou négatifs. L'algorithme permet de détecter les circuits absorbants.

6.2 Les données

En entrée

X : ensemble des sommets
n : Nombre de sommets
U : ensemble d'arcs sous forme (x,y)
p : un vecteur de m éléments où $p[u]$ représente le poids du l'arc u dans le réseau.
r : un sommet donné de X.

En sortie :

π : vecteur de n éléments. Chaque entrée $\pi[i]$ représente le poids du chemin optimal de r vers i.
Pré : vecteur de n éléments. Chaque entrée *Pré[x]* représente le prédécesseur de x dans le chemin optimal de r vers x.

Autres :

π_old : vecteur de n éléments. Contient les valeurs de π de l'itération précédente.

6.3 L'Algorithme

```
Debut
     Pour tout x∈X
        Faire
          π_old[x]← +∞ ;
          Si (r,x) ∈ U Alors  π[x]← p(r,x);
                              Pré[x] ← NULL ;
                       Sinon  π[x]← +∞ ;
                              Pré[x] ← NULL ;
          fSi
        Fait
     π[r]← 0; i ← 1;
     Tant que ((π_old ≠ π) et(i <= n))
     Faire
        π_old ← π; i ← i+1;
        Pour tout (x ∈ X-{r})
        Faire
           Pour tout ((y, x) ∈ U)
           Faire
                Si π[x] > π_old [y]+ p[(y, x)]
                       Alors π[x] ← π_old[y]+p[(y, x)];
                             Pré[x] ← y;
                fSi
           Fait
        Fait
     Fait
     Si (π_old ≠ π)    Alors Retourner("Il existe un circuit absorbant");
                       Sinon retourner(π,Pré) ;
Fin
```

6.4 Remarques

- Si à la fin de l'exécution on obtient toujours $\pi\_old \neq \pi$ alors le graphe contient un circuit absorbant.
- On dit que l'algorithme retourne une arborescence optimale.
- De même que pour Dijkstra, si le graphe est non orienté, on peut associer à chaque arête $\{x,y\}$ de poids p, deux (02) arcs (x,y) et (y,x) dont les poids sont p, puis on applique l'algorithme de Bellman-Kalaba.

7. Algorithme de Floyd

7.1 Problème traité

Le problème est de trouver le poids du chemin optimal entre toute paire de sommets dans le graphe. Le graphe peut contenir des circuits et les poids peuvent être positifs nuls ou négatifs.

7.2 Les données

En entrée

X : ensemble des sommets
n : Nombre de sommets
U : ensemble d'arcs sous forme *(x,y)*
p : un vecteur de m éléments où $p[u]$ représente le poids du l'arc u dans le réseau.

En sortie :

π: matrice de n x n éléments. Chaque entrée $\pi[i,j]$ représente le poids du chemin optimal de i vers j.
Pré : matrice de n éléments. Chaque entrée *Pré[i,j]* représente le prédécesseur de j dans le chemin optimal de i vers j.

7.3 L'Algorithme

```
Début
     Pour i de 1 à n
     Faire
        Pour j de 1 à n
        Faire
           Si (i=j) Alors π[i,j] ← 0;     Pré[i,j] ← NULL;
                    Sinon  Si((i,j)∈U)  Alors  π[i,j] ← p[(i,j)];       Pré[i,j] ← i;
                                        Sinon  π[i,j] ← +∞ ;    Pré[i,j] ← NULL;
                           fSi
           fSi
        Fait
     Fait
     Pour k de 1 à n
        Pour i de 1 à n
           Pour j de 1 à n
                Si (π[i,j] > π[i,k]+ π[k,j]) Alors π[i,j] ← π[i,k]+ π[k,j];
                                                   Pré[i,j] ← Pré[k,j];
                fSi
Fin
```

7.4 Remarques

- La complexité de l'algorithme est $o(n^3)$
- L'algorithme ne permet pas de détecter les circuits absorbants mais (à la différence de Dijkstra et Bellman-Ford) il se termine malgré la présence de circuits absorbants.
- Si le graphe est non orienté, on applique le même principe que pour Dijkstra et Bellman-Kalaba pour transformer le graphe en graphe orienté puis on pourra applique l'algorithme de Floyd.