Ch. 8 : Recherches de plus courts chemins dans un graphe valué

1. Présentation du problème
2. Conditions d'existence
3. Algorithmes de résolution
   - Algorithmes de Dijkstra
   - Algorithme de Bellman

Recherches de plus courts chemins dans un graphe valué

- Dans tout ce paragraphe, $G = (X, E, p)$ est un graphe orienté valué. Néanmoins les algorithmes présentés peuvent être très facilement adaptés au cas d'un graphe non orienté.
- La recherche d'un chemin de coût minimum entre deux sommets d'un graphe orienté valué est un des problèmes les plus classiques des graphes et les plus riche d'application à l'optimisation en général (c'est ainsi que Mappy trouve l'itinéraire minimisant la consommation d'essence ou le prix des péages ou la distance parcourue) et en particulier en informatique (en particulier le routage dynamique des paquets dans une transmission réseaux).

- On rappelle que si $C = (a_1, a_2, ...a_n)$ est un chemin, on appelle coût de ce chemin, la somme des coûts des arcs le constituant.

$$cout(C) = \sum_{i=1}^{n} p(a_i).$$

- Le but de ce paragraphe est alors de chercher à construire (s'il en existe) un chemin joignant deux sommets x et y de G et minimisant le coût total du chemin. On parle également de "plus court chemin" entre x et y.

Conditions d'existence

Definition

les poids des arêtes peuvent être négatifs, on appelle circuit absorbant un circuit de coût total négatif. Si un graphe contient un circuit absorbant, alors tout passage par ce circuit fait diminuer le coût du chemin. Il est alors clair qu'il n'existe pas de chemin de coût minimum.

Conditions d'existence

Théorème

Le problème du plus court chemin admet une solution si y est accessible à partir de x et qu'il n'existe pas de circuit absorbant. S'il n'y a pas de circuit de coût nul, les plus courts chemins sont des chemins élémentaires. S'il y en a nous nous restreindrons à la recherche de chemins élémentaires.

Hypothèses et algorithmes

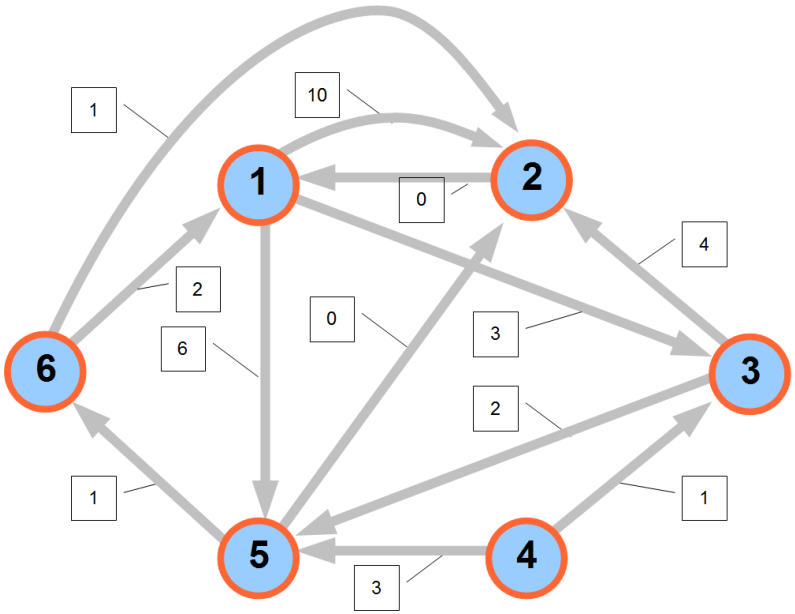
Nous allons voir trois algorithmes permettant de construire des plus courts chemins, il en existe d'autres, ils diffèrent par

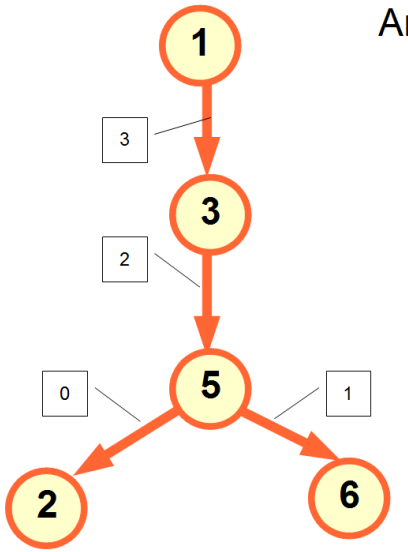
- les hypothèses qu'ils utilisent pour s'assurer qu'il n'existe pas de circuit absorbant :
  Que le graphe ait des poids positifs, qu'il n'est pas de circuit du tout ou qu'il n'ait tout simplement pas de circuit absorbant.
- Le but recherché :
  Déterminer le plus court chemin entre deux sommets x et y, entre x et tous les sommets qu'il peut atteindre, ou entre tous les couples (x, y) de sommets du graphe.
- Leur complexité.
  Quand on a le choix, on doit bien sûr choisir l'algorithme le plus efficace.

Algorithmes de Dijkstra

- Nous allons voir deux versions de l'algorithme de Dijkstra. La première (Dijkstra1) s'applique à tout graphe sans circuit absorbant, sans hypothèse supplémentaire mais n'a pas une très bonne complexité dans le pire cas.
- Le deuxième algorithme (DijkstraOpt) se limite au cas (fréquent quand la fonction de poids est un côut ou un temps) où toutes les arêtes ont un poids positif. On est alors sûr qu'il n'y a pas de circuit absorbant. cette hypothèse supplémentaire permet de limiter la complexité de l'algorithme.
- Dans les deux cas, l'algorithme permet de calculer à partir d'un sommet s les plus courts chemins entre s et tous les sommets accessibles à partir de s. (C'est ce qu'on appelle son arbre collecteur).

Le graphe G_7 (à poids positifs) et l'arbre collecteur obtenu par l'algorithme de Dijkstra)





Arbre collecteur

Algorithme de Dijkstra. Version 1

- Partant d'un sommet s, on cherche tous les sommets i accessibles à partir de s. Cela nous fournit un chemin de s à i dont on note le coût et le sommet d'origine (s).
- Tant que ce procédé met à jour de nouveaux chemins de côut minimum, on itère :
  - Pour chaque sommet modifié à l'étape précédente, on regarde si en passant par ce sommet, on obtient un nouveau chemin minimum vers un de ses voisins.
  - Si tel est le cas, on met à jour le coût du chemin et le sommet d'origine.
- En fin d'algorithme, on dispose de tous les chemins accessibles, des coûts minimaux entre s et ces sommets et du chemin lui même en remontant les sommets d'origine.

Dijkstra version 1

```
Algorithme Dijkstra;
DEBUT
{initialisations}
M:={s};
Pour i de 1 à n faire
  Si (s,i) est une arête alors
    Début
     D[i]:=cout(s,i);
     P[i]:=s;
     ajouter(i,M);
     Fin
  Sinon D[i]:=infini;
```

```
Tant que M<>{} faire
    Début
    x:=enleveTete(M);
    Pour tout successeur  y de x faire
      Début
      d:=D[x]+cout(x,y);
      si d<D[y] alors
          Début
          D[y]:=d;
          P[y]:=x;
          ajouter(y,M)
          Fin
      Fin
    Fin
FIN.
```

Dijkstra version 2

- Cette première version ne nécessite aucune hypothèse particulière mais a l'inconvénient suivant : Un même sommet se retrouver mis à jour plusieurs fois consécutives ce qui nécessite à chaque fois l'exploration de ses voisins (voir TD).
- Dans le cas où toutes les arêtes ont des poids positifs, on peut éviter cet inconvénient en choisissant à chaque étape le sommet dont le chemin depuis s est de coût minimal.
- La seule modification à apporter est ce calcul de minimum mais on est sûr que chaque sommet ne sera mis à jour qu'une seule fois.
- Nous verrons en TD pourquoi l'hypothèse des poids positifs est indispensable.

Dijkstra version 2

```
Algorithme Dijkstra;
DEBUT
{initialisations}
M:={};
Pour i de 1 à n faire
  Si (s,i) est une arête alors
    Début
    D[i]:=cout(s,i);
    P[i]:=s;
    ajouter(i,M);
    Fin
  Sinon D[i]:=infini;
```

```
Tant que M<>{} faire
    Début
    x:=choisir_min(M,d);
    enlever(x,M);
    Pour tout successeur  y de x faire
      Si y est dans M alors
      Début
      d:=D[x]+cout(x,y);
      si d<D[y] alors
          Début
          D[y]:=d;
          P[y]:=x;
          ajouter(y,M)
          Fin
      Fin
    Fin
FIN.
```

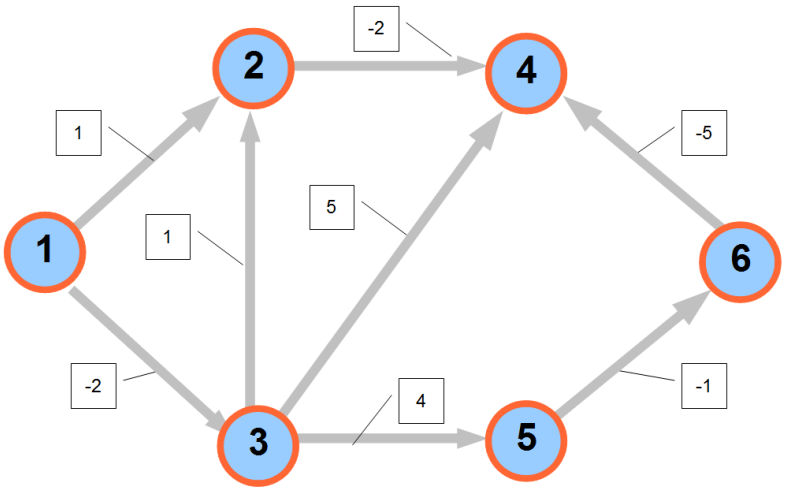
Calcul de la complexité

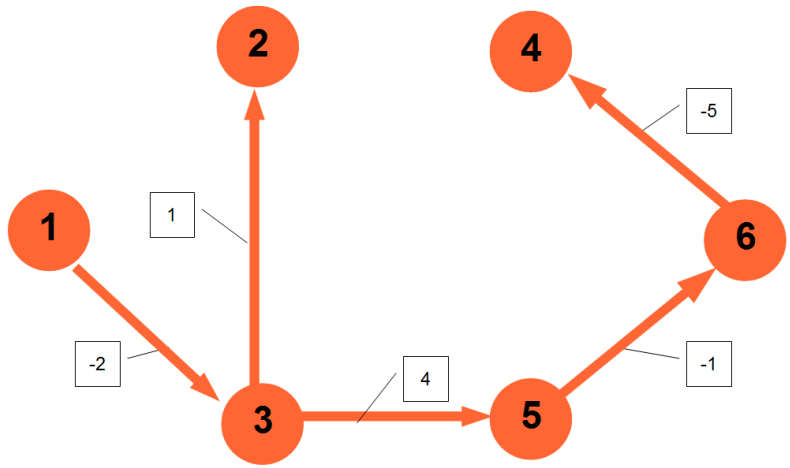
- L'initialisation est en $O(n)$.
- La boucle de traitement va parcourir une et une seule fois chaque sommet accessible à partir de s. La boucle tant que sera donc parcouru au plus n fois. A la ième étape M contient $n-i$ sommets et donc la complexité de ChoisirMin est en $O(n-i)$. le parcours des successeurs de x est en $O(d+(x))$.
- La boucle de traitement a donc une complexité de $\sum_{i=1}^{n}(n-i)+\sum_{x\in X} d^{+}x$. (on considère la boucle "tant que" en disant d'une part qu'elle parcourra tous les sommets dans un certain ordre et d'autre part qu'elle traitera des ensembles M de cardinal $n-i$ pout toute valeur de i).
- D'où une complexité de $O(n^2)+O(m)=O(n^2)$.

Algorithme de Bellman

- Cet algorithme est utilisé dans les graphes sans circuits dont les arcs ont des poids quelconques. On est ainsi assuré qu'il n'existe pas de circuit absorbant.
- Quitte à se restreindre à la composante connexe de s (i.e. à l'ensemble des sommets accessibles à partir de s,) on peut supposer que tout sommet est accessible à partir de s.
- L'idée (récursive) de l'algorithme est la suivante :" un plus court chemin de s à y doit passer par un des prédécesseurs x de s et suivra alors le plus court chemin de s à x puis l'arc (x, y)."
- On note M l'ensemble des sommets x pour lesquels on connaît un plus court chemin de s à x et $S = X \backslash M$.
- On va construire l'ensemble M suivant le principe suivant : Tant que S est non vide, il existe toujours un sommet y de S dont tous les prédécesseurs sont dans M.

Le graphe G_8 (sans cycles) et l'arbre collecteur des plus courts chemins issus du sommet 1 (obtenu par l'algorithme de Bellman)





Arbre collecteur des plus courts chemins issus de 1

Démonstration

- Par hypothèse S est non vide et chacun de ses éléments admet au moins un prédécesseur puisqu'il est accessible à partir de s.. Si tout sommet y de S admet un prédécesseur dans S alors on peut construire une suite infinie (y_n) d'éléments de S tels que y_{i+1} est prédécesseur de y_i. Comme S est fini cette suite contient nécessairement un circuit ce qui est exclus par hypothèse.
- En pratique si un tel sommet n'existe pas c'est qu'on a traité tous les sommets accessibles à partir de s.
- Pour ce sommet y Soit P_y l'ensemble de ses prédécesseurs (tous dans M). Le plus court chemin de s à y peut être calculé par

$$min_{x \in P_y}(dist(s,x) + p(x,y)).$$

  y passe alors dans M.
- L'algorithme s'arrête lorsque S est vide ou qu'on a traité tous les sommets accessibles à partir de s.

Exemple

Raffinons l'algorithme en tenant à jour deux tableaux indexés par X : $DIST$ tel que $Dist[x]$ est la longueur du plus court chemin entre s et x une fois celle-ci déterminée et P tel que $P[x]$ soit le prédécesseur de x dans ce plus court chemin. Le tableau P servira a expliciter les chemins une fois qu'il seront déterminés. On obtient l'algorithme suivant :

```
{Dist  est initialisé à une valeur conventionnelle infinie
S est initialisé à X\S }
Dist[s]:=0;P[s]:=s; fini:= faux ;
```

```
tant que  non fini faire
  Début
    fini:=vrai;
    pour tout élément y de S faire
{On traite tous les y de S dont tous les
prédécesseurs sont dans M}
        Oky:=vrai; d:=infini;
        pour tout prédecesseur x de y faire
           si Dist[x] <> infini
         alors si  Dist[x]+p(x,y)< d alors
              Début
         d:= Dist[x]+p(x,y)
         P[y]:=x;
         Fin
         sinon Oky:=false;
```

```
        Si OKy alors {On traite y}
        Début
        fini:=false;
        enlever(y,S);Si S=[] alors fini:=vrai;
        Dist[y]:=d
        Fin
Fin
```

Complexité

- La version présentée ci-dessus est simple a comprendre mais a une complexité en $O(n^3)$:
- A la $i$ème étape, S contient $n-i$ sommets. Dans le pire cas c'est le dernier élément de S qui est le bon ce qui nécessite de parcourir tous les prédécesseurs de tous les éléments de S.

  L'algorithme a donc une complexité de $C(n) = \sum_{i=1}^{n}(\sum_{y=1}^{n-i} d^-(y))$.
- On arrête le parcours des prédécesseurs dès qu'on en trouve un dans S donc on en parcourt toujours moins de $|M|+1 = i+1$.
- D'où

$$\begin{aligned} C(n) &\leq \sum_{i=1}^{n}(\sum_{y=1}^{n-i} i+1) \\ &= O(\sum_{i=1}^{n} O(n^2) = O(n^3) \end{aligned}$$

Complexité

- Cette complexité peut être ramené à $O(m)$ en évitant la boucle de parcours de recherche d'un élément y de S satisfaisant.
- Il suffit pour cela d'initialiser et de tenir à jour un vecteur PS donnant pour chaque élément de S le nombre de ses prédécesseurs dans S. pour ne pas parcourir ce tableau, on tient à jour une liste L des éléments de valeur nulle du tableau PS. C
- hoisir un y qui convient consiste alors a prendre l'élément de tête de L ce qui se fait en $O(1)$. On obtient l'algorithme suivant :

```
{Dist  est initialisé à une valeur conventionnelle infinie
Dist[s]:=0;P[s]:=s;

{construction du tableau PS et de la liste L}
L:=[];
pour y de 1 à n faire
  Début
   PS[y]:=0
   si y=s alors PS[y]=-1 {on exclus s en lui donnant une v
   pour chaque prédécesseur z de y faire
   si z<>s alors PS[y]:=PS[y]+1;
   Si PS[y]=0 alors ajouter(y,L)
  Fin;
```

```
tant que  L<>[] faire
Début
  y:=enleveTete(L);
      pour tout prédécesseur x de y faire
          si  Dist[x]+p(x,y)< Dist[y] alors
         Début
         Dist[y]:= Dist[x]+p(x,y)
         P[y]:=x;
         Fin
         {Mise à jour de PS et L}
         {tout successeur de y a un prédécesseur de moins
         Pour chaque successeur z de y faire
          Debut
          PS[z]:=PS[z]-1;
          Si PS[z]=0 alors ajouter(z,L)
          Fin
Fin
```

Complexité

- La complexité de l'algorithme est alors de $\sum_{y=1}^{n} d^{-}(y) = O(m)$ pour l'initialisation de PS et L.
- La partie traitement ayant une complexité de $\sum_{y=1}^{n} d^{-}(y) + d^{+}(y) = O(m)$. On a donc une compléxité totale en $O(m)$. donc toujours meilleure que $O(n^2)$.