Problème du chemin plus court et variantes (CSP, KSP)

ÉCOLE POLYTECHNIQUE
INF 442

José Manuel de Frutos PEI

Hypothèses

On considère qu'on a un graphe orienté et le poids des arêtes est non négatif. G = (V,A,c). Avec V l'ensemble des nœuds et A⊆V×V l'ensemble d'arcs. Soit aussi c(e)≥0 pour tout e∈ A le cout de chaque arc.
On décide de traiter d'abord la question 3 puis la 2.

Question1

Algorithme séquentiel

On a décidé dans cette question d'implémenter l'algorithme de Dijkstra pour trouver le chemin plus court entre un point source et un point objectif. Avec les hypothèses faites avant l'algorithme de Dijkstra est bien valable. Si le poids des arêtes pouvait être négatif alors on devrait implémenter des algorithmes comme celui de Bellman-Ford ou de Floyd-Warshall.

Voici le pseudo-code de l'algorithme de Dijkstra séquentiel. (Adaptation de "Introduction to Algrotithms. Thomas H. Cormens" Section 24.3)

```
INITIALIZE-SINGLE-SOURCE(G,S)
    for each vertex v ∈ G.V
        v.d = ∞
        v.π = NIL
    s.d = 0

RELAX(u,v,w)
    if v.d > u.d + w(u,v)
        v.d = u.d + w(u,v)
        v.π = u

DIJKTRA(G,w,s)
    INITIALIZE-SINGLE-SOURCE(G,S)
    S= ∅
    Q= G.V
    while Q≠ ∅
        u =EXTRACT-MIN(Q)
        S = S ∪ {u}
        for each vertex v ∈ G.Adj[u]
            RELAX(u,v,w)
```

Dans le pseudo-code chaque $v \in V$ a un attribut v.d qui est une borne supérieure du poids du chemin le plus court de la source s jusqu'à v.
$v.\pi$ est le nœud prédécesseur dans le chemin le plus court trouvé à ce stade qui va de la source et termine en v.
w(a,b) est le poids de l'arête qui commence en a et finit en b.

Dans notre implémentation on a suivi ce même pseudo-code, en utilisant un tableau dis[ ] pour représenter l'attribut v.d, un tableau previous[ ] pour $v.\pi$. Notre ensemble Q est ici une file de priorité (On utilise celle de la STL), avec comme fonction de comparaison celle qui range les nœuds en ordre croissant du poids du chemin le plus court trouvé de la source jusqu'au nœud. On utilise aussi un tableau vis[ ] pour indiquer si un nœuds a déjà été visité ou non. Ainsi on évite de tourné en rond et de jamais finir si le graphe possède un cycle.

L'algorithme a une complexité en temps de $O(|A|log|V|)$ car on utile une file de priorité.

Algorithme parallèle

Dans notre version parallèle on n'utilise plus une file de priorité mais un simple tableau pour Q. Cela fait que l'algorithme soit plus facilement parallélisable. Chaque processus se charge d'une partie des nœuds. On divise donc le tableau des nœuds en V/processus parties. Individuellement chacun trouve le minimum (le nœuds pour lequel le chemin qui va de la source jusqu'à lui est minimum). Chaque processus envoie le résultat au processus maitre (le processus 0) qui calcule le minimum global et informe au reste des processus de ce minimum. On considère après ce nœuds comme visité et traité et chaque procesus pour le groupe de noeuds qui lui est attribué applique la méthode de Relaxation.

Analyse de la compléxité
(1) La recherche du minimum dans un tableau se fait en temps linéaire donc pour p processus cela coute O(V)/P.
(2)Puis le processus maitre calcule le minimum global des p minimums locales trouvés. Cela coute P.
(3)On doit aussi actualiser les nœuds dans le tableau. cela ce fait en temps O(1) pour chaque nœuds adjacent. C'est à dire en temps N.
(4) On répète l'étape 1 et 2 pour chaque nœuds donc V/P fois.
On obtient ainsi une compléxité $O(V^2)/P + O(V) * P + O(NV)/P$.
Mais comme $|A| = O(NV)$ qui est $O(V^2)$ on a ainsi : $O(V^2)/P + O(V) * P$ comme compléxité. Si on peut chercher le minimum de cette fonction pour P. En dérivant on obtient que pour $P = \sqrt{V}$ la compléxité est optimale ayant $O(V^{3/2})$.
Dans cet analyse on n'a pas considéré le cout des communications. Mais évidemment en fonction de la topologie du réseau et des capacités physiques, ceci peut-être le facteur limitant.

Question3

Algorithme séquentiel

Dans cette question le but est d'énumérer les k-chemins plus court qui vont d'un nœuds source à un nœud destin. Pour cela on a implémenté l'algorithme de Yen.

```
YENKSP(G,s,t,k)
    A[0]=Dijkstra(G, s, t) ;
    B= [ ]
    for k from 1 to K :
        for i from 0 to size(A[k-1])-1 :
        spurNode = A[k-1].node(i) ;
        rootPath = A[k-1].nodes(0, i) ;
        for each path p in A :
            if rootPath == p.nodes(0, i) :
                remove p.edge(i, i + 1) from G ;
            for each node rootPathNode in rootPath except spurNode :
                remove rootPathNode from Graph ;
            spurPath = Dijkstra(G, spurNode, t) ;
            totalPath = rootPath + spurPath ;
            B.append(totalPath) ;
            restore edges to G ;
            restore nodes in rootPath to G ;
        if B is empty :
            break ;
        B.sort() ;
        A[k] = B[0] ;
        B.pop() ;
    return A ;
```

On assume que l'ensemble A va avoir les k-plus courts chemins. B de sa part détiendra les potentiels k-plus courts chemins.
D'abord on détermine le plus court chemin. N'importe quel algorithme pour déterminer le plus court chemin d'un graphe vaudra. (DIJKTRA par exemple).
Pour trouver le k-énième plus court chemin l'algorithme assume que les k-1 chemins plus courts ont été trouvé. La k-énième itération consiste a trouver toutes les possibles déviations et choisir celle de poids minimum. Pour cela on choisi d'abord un chemin racine, "rootPath". Ce chemin est choisi en trouvant le sous-chemin de A^{k-1} qui suit les i premier nœuds de A^j (où j va de 1 à k-1). Alors si un tel chemin est trouvé, le poid des arêtes (i, i+1) de A^j est mis à infini ie on enlève l'arête du graphe. Le "spur path" (C'est à dire le chemin qui va du nœud i jusqu'au nœud objectif) est trouvé en calculant le chemin le plus court (exemple algorithme de DIJKTRA). Le fait d'avoir enlevé les arêtes (i, i+1) assure que le "spur path" est différent. On obtient ainsi un potentiel k-énième chemin plus court en rassemblant le "rootPath" et le "spur path". On ajoute ce nouveau chemin à B. Puis les arêtes qui ont été enlevées sont remises et on répète le processus pour le i suivant. Finalement après avoir calculé toutes les déviations valables du k-1 chemin plus court on enlève de B le chemin de poids minimal et on l'ajoute à A. C'est le k-énième chemin plus court.

Dans notre implémentation on a défini plusieurs classes. Une classe Path qui représente l'objet chemin, une classe Graph pour représenter le graphe. Une classe Dijkstra qui, donnée un graphe, un noeud source et un noeud objetif permet de calculer le plus court chemin. Et finalement une classe Yen algorithm qui exécute l'algorithme de Yen décrit au dessus.
Dans notre implémentation de l'algorithme de Yen on utilise un multiset (définit dans la STL) avec une fonction d'ordre qui range en ordre croissant de poids les chemins. Ainsi

le premier élément de B est le k-plus court chemin.
A est un simple vector.
L'algorithme de Dikjstra est grosso modo celui implémenté dans la question 1. Et les classes Graph et Path possédent des méthodes d'utilité comme enlever les arêtes entre deux nœuds d'un graphe les restaurer et aussi l'enchaînement de deux chemins.

La complexité en temps de l'algorithme de Yen dépend évidemment de l'algorithme utilisé pour trouver le chemin le plus court. En utilisant l'algorithme de Dijkstra qui a comme complexité $O(|A|log|V|)$ on déduit :
L'algorithme de Yen fait Kl appelles à Dijkstra, avec l la longueur du A^{k-1} chemin plus court. En faisant une approximation grossière on peut dire que l est dans le pire des cas $O(|A|)$. Donc on déduit que la complexité est majorée par $O(K|A|^2log|V|)$.

Algorithme parallèle

Comme on voit l'algorithme est facilement parallélisable car donné une k-ème étape le calcule des $size(A^{k-1}) - 1$ possibles déviations est indépendant. Donc on peut paralléliser cette étape. Chaque processus calculera un ensemble de déviations. En particulier si $size(A^{k-1}) - 1 = n$ et on dispose de p processus alors chaque processus calcule n/np déviations. Si $n < np$ on aura des processus inactifs.
Après ce calcul on rassemble tout ces potentiels chemins plus court dans un processus maitre (dans notre cas le processus 0) qui choisit le minimum et l'ajoute à son A et puis on actualise l'ensemble A et B de tout les processus.

Dans la parallélisation choisie, si le nombre de processus est plus grand que l, on n'aura des processus inactives et donc la compléxité ne s'améliorera pas. Donc si on considére que l'on a suffisamment de processus pour calculer chaque déviation en parallèle la compléxité sera : $O(K|A|log|V|)$

Question2

Algorithme séquentiel

On sait que "The constrained shortest Path problem" est NP-Difficile (voire https ://smartech.gatech.edu/bitstream/handle/1853/28268/garcia_renan_200905_phd.pdf pag 22). C'est à dire il n'y a pas d'algorithme connu qui n'est pas un temps d'exécution exponentiel. Cela justifie notre suivante approche. (Pour d'autres approches voir la référence d'avant pag 22-29).
On décide donc avec l'algorithme de Yen d'énumérer pas à pas les chemins les plus courts jusqu'à trouver un qui vérifie la contrainte. Ce chemin sera bien le chemin le plus court vérifiant la contrainte. S'il n'y pas un telle chemin on continuera jusqu'à avoir exploré tous les chemins possibles. Dans ce cas on terminera car B sera vide.

Algorithme parallèle

L'approche parallèle est la même que pour l'algorithme de Yen. Chaque processus calculera un ensemble de déviations. Après ce calcule on rassemble tout ces potentiels chemins plus court dans le processus maitre qui choisit le minimum et voit s'il vérifie la contrainte. Si oui on finit en indiquant aux autres processus la fin des calculs. Si ce n'est pas le cas on ajoute ce chemin à A et puis on actualise l'ensemble A et B de tout les processus et on continue.