Projet LSINF1121 - Algorithmique et structures de données

-

Rapport intermédiaire Mission 6

Groupe 26

Laurian Detiffe (6380-12-00)

Sundeep Dhillon (6401-11-00)

Alexis Macq (5910-12-00)

Xavier Pérignon (8025-11-00)

Thibaut Piquard (4634-13-00)

Thomas Wyckmans (3601-12-00)

Année académique 2015-2016

Questions et réponses

1.
2.
3.
4. La solution est d'utiliser un Breadth-first search (BFS). En fesant celà, on trouve les noeuds adjacents au départ.Si ceux-ci ont déjà étés visités, on les ignore. Autrement, on l'ajoute aux éléments du tableau qui renverra le chemin le plus cour. On réitère cette opération jusqu'à atteindre la sortie. On renvoie ensuite le tableau obtenu qui nous donnera le chemin désiré. Notre complexité sera de O(n*p) pour une matrice d'ajacence et celà peut diférer selon l'implémentation.

```
Private void bfs(Graph G, int debut,int finish)
{
        Queue<Integer> q = new Queue<Integer>();
        IsAlreadyVisited[debut] = true;
        q.enqueue(debut)
        while(!q.isEmpty())
        {
        int v = q.dequeue();
        if ( v == finish)
                {
                break;
                }
        for (int w : G.adjacent(v))
                if(! IsAlreadyVisited[w])
                {
                        path[w] = v ;
                         IsAlreadyVisited[w] = true;
                        queue.enqueue(w);
                }
        }
}
```

5. On utilise un Depth-first search(DFS) qui vérifie à chaque fois dans le cas ou le noeud trouvé a déjà été visité si c'est un noeud déjà parcouru dans ce chemin du DFS. Dans ce cas, il y a cycle.

```
Public boolean hasCycle(Graph G, int v, int u,hasCycle)
{
         IsAlreadyVisited[v];
        for ( int w : G.adjacent(v))
                {
                if(! IsAlreadyVisited[w])
                {
                        dfs(G,w,v,hasCycle);
```

```
                }
                else if (w!=u) hasCycle = true;
                }
        return hasCycle;
}
```

6.

```
Public void Topologic(Graph G,int n)
{
        Int[]d = new Int[n]
        Queue<Integer> q = new Queue<Integer>();
        int k;
        for (int i=0; i++;i <= n)
        {
                d[i]=i.degre()
                if(d[i]==0)
                {
                        q.add(i)
                }
        }
        while(!q.isEmpty())
        {
                k=q.remove()
                d[k]=d[k]-1;
                for(int i : G.adjacent(k))
                {
                        d[i]=d[i]-1;
                        if(d[i]==0)
                        {
                                q.add(i)
                        }
                }
        }
}
```

7. Afin de compléter le MST partiel et retrouver un MST complet dans le graphe G, je vais appliquer l'algorithme de Kruskal. Cet algorithme est simple, il récupère tous les Lien (Edge) triés de manière croissante (plus petit poids au plus grand) et va ajouter au fur et à mesure tous ceux qui ne forme pas de cycle :

```
Public static void RecoverMST(EdgeWeightedGraph G, Queue<Edge>, UF uf)
{
        MinPQ<Edge> pq = new MinPQ<Edge>(G.edges());
        while (!pq.isEmpty() && mst.size() < G.V()-1)
        {
                Edge e = pq.delMin();                //Get min weight edge
                    on pq
```

```
                int v = e.either(), w = e.other(v);  //and its vertices.
                If (uf.connected(v,w)) {continue;}  //ignore ineligible
                    edges.
                uf.union(v,w);                      //merge components
                mst.enqueue(e);                     //Add edge to mst
        }
}
```

`http://algs4.cs.princeton.edu/43mst/KruskalMST.java.html`

Cet algorithme se base sur une règle fondamentale du MST, il nepeut y avoir de cycle à l'intérieur d'un MST. Grâce à « union-find », on peut savoir si deux somments sont déjà connectés ou non, et s'il faut donc traiter cet Edge (l'ajouter au MST).
Il est aussi possible de régénérer l'union-find depuis le MST existant, simplement en parcourant celui-ci et en liant chacun des différents sommets dans chacune des liants de ce MST partiel. La complexité temporelle de cette méthode est de O(Elog(E)).

8. On peut élargir ce problème à un problème de MST plus vaste, c'est-à-dire : comment mettre à jour votre MST après un Edge quelconque ait été mis à jour dans votre Graphe ?
   Il y a plusieurs cas possible :

   (a) L'edge mis à jour est dans votre MST et sa valeur décroît : il n'y a strictement rien à faire.
   (b) L'edge mis à jour n'appartient pas au MST et sa valeur décroît : il faut ajouter l'Edge dans le MST, ce qui créera un cycle au sein du MST. Il suffit ensuite de parcourir le MST depuis un des deux Vertice de cet Edge (via BFS ou DFS) et retirer dans ce cycle l'Edge au poids le plus haut (Complexité O(N)).
   (c) L'Edge mis à jour est dans votre MST et sa valeur croît : il faut retirer l'Edge de cet MST, ce qui créera deux éléments connectés qui doivent être raccordés. On peut facilement calculer ces deux composants via DFS ou BFS (Complexité O (N)), il faut ensuite parcourir les Edge restants par ordre croissant et ajouter le premier Edge qui relie les deux éléments connectés.
   (d) L'Edge mis à jour ne fait pas partie du MST et sa valeur croît : Le MST actuel est toujours un MST.

   Maintenant, ces diffétents cas ne couvrent pas spécifiquement notre probléme tel que je le comprends, c'est-à-dire, inclure quoi qu'il arrive l4Edge e même si celui-ci ne devrait pas se trouver dans le MST. Pour arriver à résoudre ce problème, il suffit de tweaker un petit peu le cas 2. Au moment du retrait du poids le plus haut, si celui-ci est l'Edge e, alors on retirera le second plus haut.

9. Oui, en utilisant une priority queue qui contiendrait les noeuds candidats à la prochaine relaxation de la manière suivante :

   1 - On ajoute le noeud source s à un arbre et ses noeuds adjacents à la priority queue avec leur distance à s comme clef.

   2 - On retire de la priority queue le noeud n de clef minimale, on le rajoutte à l'arbre (= on le relaxe) et on ajoute ses voisins à la queue de priorité avec comme clefs leur distances à n + la distance de n à s comme étant leurs distance à s si celle-ci est plus petite que la distance à s actuelle ou si celle-ci est la première distance à s ajoutée pour le noeud en cours de traitement.

   3 - On répète l'étape 2 jusqu'à ce que chaque noeud ait été ajouté et retiré une fois de la priority queue.
   L'arbre ainsi formé contient les plus courts chemins de s à tout noeud v de V.

   L'algo de Dijkstra a une complexité spatiale proportionnelle à |V| et une complexité temporelle proportionnelle à |E|log(|V|) dans le pire des cas pour calculer tous les plus courts chemins des noeuds de V à une source appartenant à V.
   Démonstration :
   Le bottleneck dans cet algorithme est le nombre de comparaisons entre clefs des noeuds dans les méthodes insert() et delMin() de la priority queue. Le nombre de noeuds dans la priority queue est au plus |V| ce qui nous donne la complexité spatiale.
   Dans le pire des cas, le coût d'une insertion est proportionnel à log(|V|) et le coût d'une suppression est proportionnel à 2log(|V|). Puisqu'on a au plus |V| noeuds inséré et |V| noeuds supprimés, la complexité temporelle est |E|log(|V|).

   (Alexis)

10. Soit le graphe suivant :

    En empruntant le cycle w-z-x une infinité de fois, on sait montrer que la distance minimale entre y et w est $-\infty$ or en appliquant l'algorithme de Dijkstra, on ne peut relaxé qu'une fois chaque noeud ce qui implique qu'on ne peut parcourir le cycle w-z-x de longueur négative qu'une et une seule fois et l'algorithme de Dijkstra ne pourra donc pas nous permettre d'obtenir la solution optimale, à savoir $-\infty$. Il gardera toutefois la même complexité malgré la présence de poids négatifs puisqu'en appliquant l'algorithme de Dijkstra, on ne peut relaxé qu'une seule fois chaque noeud.

    (Alexis)

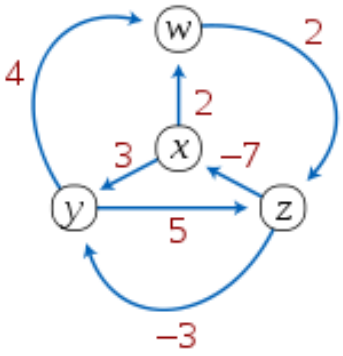
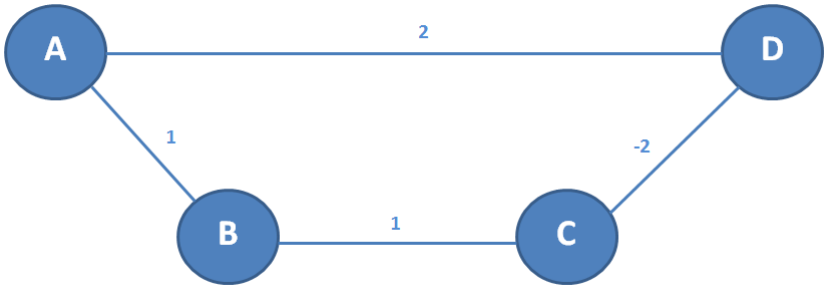


FIGURE 1 – Graphe contenant un cycle de longueur impaire (w-z-x)

11. Soit G un graphe avec des poids potentiellement négatif mais il n'y a pas de cycle négatif. Je cherche le chemin le plus court entre un noeud u et un noeuds v. J'ai à ma disposition une implémentation de Dijkstra qui ne permet pas de gérer les poids négatifs. Il me suffit dès lors d'augmenter tous les poids d'une même quantité correspondant a la valeur absolue du plus petit poids et d'appliquer Dijkstra sur ce graphe. Cette méthode est-elle valable ? Si non, montrez un contre exemple. (Xavier)

    Non, c'est méthode n'est pas valable. Par exemple, considérons le graphe où il existe deux chemins de A vers B, l'un traversant un seul arc de longueur 2, et l'autre traversant des arcs de longueur 1, 1 et -2. Le deuxième chemin est plus court, mais si on augmente de 2 tous les poids des arcs (valeur absolue du plus petit poids), le premier chemin a maintenant une longueur de 4, et le second a une longueur de 6, inversant le chemin le plus courts.



    Cette méthode ne fonctionnera que si tous les chemins possibles entre les deux noeuds utilisent le même nombre d'arcs.

12. Soit G un graphe avec des poids positifs. Je cherche le chemin le plus long entre

un noeud u et un noeuds v. J'ai à ma disposition l'implémentation de Bellman-Ford (qui supporte les poids négatifs). Il me suffit dès lors de calculer le plus court chemin sur le même graphe avec l'opposé des poids. Est-ce que cette méthode est valable ? Si non pouvez-vous proposer une méthode pour le calcul de plus long chemin ? Votre méthode s'applique-t-elle à tous les graphes ? Si non quels-types particuliers de graphes peut-elle gérer ? (Xavier)

Cette méthode est valable pour certains graphes. Le principe le plus important pour l'utilisation de cette méthode est qu'il n'y ait pas de cycle dans le graphe, dans laquelle les arcs ont une somme négative. Dans ce cas, une boucle infinie serait générée et aucun plus long chemin ne serait trouvé.