Projet structure de données et algorithmique

Haas Antoine, Truong Tan Dat

19 décembre 2014

1 Introduction

Les arbres couvrants de poids minimum sont utilisés dans de nombreux domines tels que le réseau ou encore dans la géographie avec la recherche du plus court chemin entre toutes les villes d'un pays.
Le but de ce projet est de comparer les performances des algorithmes de Prim et de Kruskal implémentés à l'aide de plusieurs algorithmes de tri.

2 Structure du graphe

Un noeud contient son identifiant, un flag utilisé pour l'algorithme de Kruskal, un pointeur vers le noeud suivant, ainsi qu'un pointeur vers une liste des arcs dont ce noeud est le prédécesseur.
Chaque arc contient le noeud prédécesseur, le poids de l'arc, le noeud successeur ainsi qu'un pointeur vers l'arc suivant de la liste. Un graphe contient un pointeur vers le premier noeud du graphe, ainsi que le nombre de noeuds.
C'est une liste d'adjacence.

Cette structure permet d'obtenir aisément les arcs à utiliser dans les algorithmes de Prim et de Kruskal.

3 Algorithme de Prim

3.1 Principe

L'algorithme de Prim, tout comme l'algorithme de Kruskal, permet d'obtenir un graphe couvrant de poids minimum, en supposant qu'il soit connexe.
Il ajoute les noeuds au graphe un par un en ajoutant a chaque fois l'arc de poids minimum entre un noeud appartenant au graphe résultat et un noeud du graphe de départ si les deux sommets de cet arc n'appartiennent pas déjà au graphe résultat. Le déroulement de cet algorithme est comme suit :
Il initialise d'abord le graphe résultat et y ajoute un sommet quelconque du

graphe de départ.
Il ajoute ensuite tous les arcs qui sont adjacents à ce sommet à une file de priorité, qui permet de définir l'ordre de traitement des arcs de la file selon un poids croissant, ce qui permet d'obtenir à chaque fois l'arc de poids minimal.

Il suffit ensuite de sélectionner le premier arc dont un seul des deux sommets n'est pas dans le graphe résultat, d'ajouter le noeud manquant ainsi que l'arc au graphe, et enfin d'ajouter toute la liste des arcs adjacents à ce sommet à la file de priorité. Il suffit de répéter cette procédure tant que tous les noeuds ne sont pas dans le graphe résultat, ou bien tant que la file de priorité n'est pas vide(Le graphe n'est alors pas connexe, on ne peut obtenir qu'une composante connexe du graphe).

La file de priorité peut être représentée sous plusieurs formes, comme par exemple un tas binaire ou un tableau.

3.2 Pseudo-code

```
prim(g) Graphe -> Graphe
prim(g):
    initialiser la priority queue pq a vide
    initialiser un second graphe g1
    ajouter le premier noeud de g a g1
    ajouter tous les arcs qui ont pour predecesser le premier noeud a pq
    trier la priority queue

    tant que le nombre de noeuds de g1 est inferieur au nombre de noeuds de g
             et que la priority queue n'est pas vide on extrait le minimum de la pq
        si le successeur de l'arc minimum n'est pas dans g1
            ajouter le minimum a g1
            ajouter tous les arcs qui ont pour predecesseur le minimum
            trier la priority queue
        fsi
    ftant que
```

3.3 Complexité

La complexité de l'algorithme de Prim est décomposée en plusieurs complexités ajoutées.

- L'ajout successif des noeud se fait en (n).
- L'ajout successif des arcs à la file de priorité se fait selon la structure de données de la file de priorité utilisée, pour le tas binaire par exemple elle est égale au nombre d'arcs * log(nombre d'arcs).

La complexité de l'algorithme de Prim est donc celle de l'ajout successif des arcs à la file de priorité (A*log(A)) en théorie.

Cependant l'utilisation des identifiants dans les structures de graphe et non de pointeurs sur les noeuds en eux-même rend nécessaire une fonction de parcours de la liste de noeuds à chaque fois que l'on extrait un arc, ce qui multiplie cette complexité par n, le nombre de sommets du graphe, elle est donc de (n*(A*log(A))) dans notre algorithme.

4 Kruskal

4.1 Principe

Kruskal est un algorithme permettant de récupérer, avec la précondition de recevoir un graphe connexe en entrée, un arbre recouvrant de poids minimum, également abrégé ARPM.
Avant de pouvoir traiter l'algorithme de Kruskal en tant que tel, il faut tout d'abord trier les différentes arêtes du graphe en fonction de leurs différents poids. Une fois cette première étape effectué l'algorithme de Kruskal peut commencer. L'algorithme de Kruskal est un algorithme glouton, car il privilégie un minimum local afin d'obtenir le minimum global. Pour ce faire l'algorithme choisit l'une des arêtes du poids le plus faible et l'ajoute à un graphe initialisé vide. Il continue ainsi de choisir les plus petites arêtes autant de fois qu'il faut, à savoir n - 1, n étant le nombre de sommet. Cependant il y a une condition que Kruskal se doit de respecter afin de ne pas obtenir d'arbre, il ne doit pas ajouter de cycle lors de la construction du graphe, et ceci se vérifie lors de la tentative d'ajout d'une arête. Une méthode efficace pour vérifier cette condition est la méthode ensembliste de l'union-set. Celle-ci permet d'élire un représentant pour chaque composante du graphe non reliée entre elle, et permet de vérifier facilement lesquels ont peut connecter entre elles.

4.2 Pseudo-code

```
kruskal(Graphe) = ARPM
kruskal(g)
    liste = tri(arcs(g))
    ARPM t = initARPM
    tant que taille(arcs(t)) < taille(sommets(g) - 1) faire
        tête = tête(liste)
        supprimer_tête(liste)
        si !appartient_ensemble(tête)
            alors
                ajouter_ARPM(tête)
        fsi
    fin faire
    renvoyer t
```

4.3 Explication

arcs() : Liste des arcs du graphe.
sommets() : Liste des sommets du graphe.
tête() : Renvoie la tête de la liste.
appartient_ensemble() : Méthode de l'union-set.

5 Tas binaire

5.1 Principe

Principe du tas binaire min

Le tas binaire est régit par deux lois :

- Un noeud a zéro, une ou deux racines.
- Un noeud a un poids toujours plus petit que ses fils.

Prenant en compte ces deux lois, on peut tenter de construire un tas binaire grâce à deux fonction :

- L'ajout : On ajoute un noeud à la première position disponible du tas binaire, et on le compare a son père. Tant que ce noeud est plus grand que son père et qu'il n'est pas la racine du tas binaire, on échange le noeud et son père, cette opération est nommée percolate up en anglais.
- L'extraction : On garde la valeur de la racine en mémoire, puis on la remplace par le dernier noeud du tas, et tant que ce noeud est supérieur à l'un de ses fils, on échange le noeud et le maximum des deux fils.

5.2 Pseudo-code

5.2.1 Heapsort

```
heapsort(t) Priority_queue -> tableauRange
heapsort(t):
    init t1 un tableau au nombre d'elements que contient t
    init i a 0
    tant que le nombre d'element de t n'est pas nul
    rep
        t1[i] = extraction(t)
        i++
    frep
    retourner t1
```

5.2.2 Extraction

```
extraction(t) Priority_queue -> arc
extraction(t):
    init pere a 1
    init filsg a 2
```

```
    init filsd a 3
    sauvegarder val la valeur du noeud racine du tas
    remplacer la racine par le dernier noeud du tas

    /*Percolate down*/
    tant que le pere est superieur a un de ses fils
    rep
            echanger(pere,max(fils))
        pere = max(fils)
        filsg = pere*2
        filsd = pere*2+1
    frep
    retourner val
```

5.2.3 Ajouter arc à la priority queue

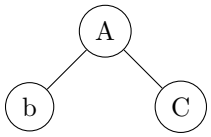
```
AjouterArcPQ(t,a) Priority_queue arc -> priority_queue
AjouterArcPQ(t,a):
    ajouter l'arc au tas a la premiere place disponible, si ce n'est pas la racine,
        init fils au nombre de noeuds de la priority queue
        init pere a fils/2

        /*Percolate up*/
        tant que le pere n'est pas racine
                 et que le poids du pere est superieur au poids du fils
        rep
            echanger(pere,fils)
            fils = pere
            pere = fils/2
        frep
    fsi
    retourner t
```

5.3 Complexité

Un tas binaire est un arbre binaire parfait, pour remplir le tableau trié, il faut échanger la racine et le dernier noeud du tas binaire et ensuite échanger ce noeud avec le maximum de ses fils tant que ce noeud est supérieur à ses fils. Le nombre d'échanges maximal fait est donc égal à la hauteur du tas, qui est donc de log(n) pour un arbre binaire parfait, et répéter cette opération n fois, la compléxité pour remplir le tableau trié est donc de n * log(n).

Percolate down



b est plus petit que A, échange des noeuds B et A.

6 Merge sort

6.1 Principe

Merge sort est un des tris les plus efficaces qui existe actuellement. Son implémentation, désuète il y a de ça quelques années est de nouveau d'actualité grâce à l'utilisation de thread.

La philosophie de merge sort est d'utiliser un des principes clé de l'informatique, à savoir diviser pour régner. Pour ce faire, le principe de merge sort est de fusionner, d'où le nom du tri, deux listes d'éléments, afin de former une nouvelle liste d'éléments triée. Tant que la liste n'est pas totalement triée, l'algorithme continue récursivement à trier la liste jusqu'à ce que celle-ci soit triée. La fusion des blocs s'effectue ainsi :

- Prendre le premier élément de chaque bloc,
- les comparer entre eux, et ajouter le plus petit des deux dans la liste servant à la fusion,
- déplacer le curseur vers l'élément suivant dans le bloc correspondant.

6.2 Fonction merge sort

6.2.1 Pseudo code

```
merge_sort(liste) = liste
merge_sort(liste)
    si taille(liste) == 1
        alors
            renvoyer liste
        sinon
            l = listenouv
            tant que !est_vide(liste) faire
                l1 = liste
                l2 = elt_suiv(liste)
                tri_bloc(l1,l2,l)
                liste = elt_suiv(l2)
                l = elt_suiv(l)
                l = listnouv
            fin faire
    fin si
merge_sort(liste)
```

6.3 Fonction tri bloc

6.3.1 Pseudo code

```
tri_bloc(liste, liste, liste)
tri_bloc(l1, l2, l)
    si est_vide(l2)
        alors
            l = l1
        sinon
            b1 = bloc(l1)
            b2 = bloc(l2)
            b = bloc(liste)
            tant que  !est_vide(b1) && !est_vide(b2) faire
                si valeur(b1) <= valeur(b2)
                    alors
                        /*valeur a deux arguments permet de modifier la valeur du bloc*/
                        valeur(b, valeur(b1))
                        b1 = bloc_suiv(b1)
                        b = bloc_suiv(b)
                    sinon
                        valeur(b, valeur(b2))
                        b2 = bloc_suiv(b2)
                        b = bloc_suiv(b)
                        b = bloc_nouv
                fin si
            fin faire
    fin si
```

6.4 Complexité

Merge sort peut se voir sous la forme d'un arbre binaire. Chaque noeud de l'arbre représente une fusion des listes de départ. La racine est la liste triée, alors que les feuillles sont toutes des listes de taille 1 comportant chacune un élément de la liste. Afin d'effectuer le tri, il faut, soit effectuer un parcours en largeur en partant des feuilles, soit effectuer un parcours postfixe, mais alors dans ce cas là l'exécution des opérations ne se fait pas nécessairement dans le bon ordre (à savoir partie gauche d'abord, puis partie droite).
Cette représentation sous forme d'arbre binaire permet donc de borner la complexité de merge sort en $\theta(n)\log_2(n)$, grâce à la hauteur de cet arbre. En effet, comme tri bloc à une complexité en $\theta(n)$, car cette fonction visite chaque élément des deux listes à triées une fois, et que la hauteur de cet arbre binaire, qui est proche d'un arbre parfait, est de $\log_2(n)$, on effectue les opérations n fois la hauteur de l'arbre à savoir $n\log_2(n)$.

7 Tests de performances

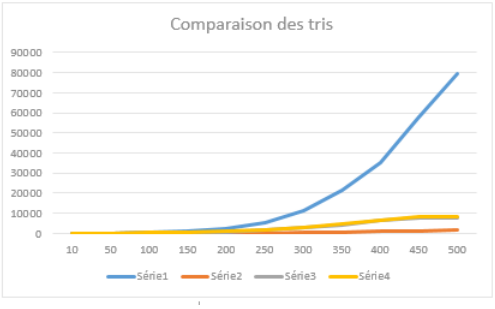
7.1 Générateur de graphes et méthodes de tests

Afin de tester nos différents algorithmes, autre qu'en utilisant de petit graphes fait main, nous avons utilisé un générateur de graphes trouver sur internet, source : http ://condor.depaul.edu/rjohnson/source/graph_ge.c.
Il s'agit d'un bon générateur de graphes, relativement user-friendly et suffisament complet pour en obtenir de différentes catégories. Afin de réaliser nos tests, nous avons choisi de travailler sur des graphes complets orientés de taille 10 et ensuite 50 puis une taille de graphes allant jusqu'à 500 et suivant un pas de 50. Ceci nous permet donc de voir une évolution lors des comparaisons, et d'obtenir des données sur des graphes de tailles suffisament conséquentes.

Une fois ces graphes générés, 5 de chaque taille pour avoir un échantillonage moyen suffisant, nous les testons en mesurant la vitesse d'exécution des algorithmes de Prim et Kruskal, et en utilisant soit une structure basée sur les binary heap, soit un tri basé sur merge sort. La mesure du temps est réalisée grâce à la fonction gettimeofday() qui permet d'obtenir le temps en millisecondes grâce à un petit calcul. Les valeurs ont ensuite reporté dans un tableau.

7.2 Graphique et données

Prim	Temps	0,3	46,25	256,2	950,4	2562,6	5216,8	11251,6	21184,6	35366,2	58084,8	79191,2
Merge sort	Echelle	10	50	100	150	200	250	300	350	400	450	500
Kruskal	Temps	0,2	7,276	18,264	53,2	96,2	171,4	363,6	474,4	970,8	1014,2	1684,8
Merge sort	Echelle	10	50	100	150	200	250	300	350	400	450	500
Prim	Temps	0,945	63,8	275,8	632,8	592	1496,4	2841,6	4362,4	6495,8	7751,4	7895
Binary heap	Echelle	10	50	100	150	200	250	300	350	400	450	500
Kruskal	Temps	1,34	58,8	255	741,4	917	1539	2922,8	4568,6	6407,2	8161,6	8371,2
Binary heap	Echelle	10	50	100	150	200	250	300	350	400	450	500



- Serie 1 : Prim et merge sort
- Serie 2 : Kruskal et merge sort
- Serie 3 : Prim et binary heap
- Serie 4 : Kruskal et binary heap

7.3 Analyse

Nous remarquons que les temps d'exécution ne sont pas fameux, et ceci est notamment du au fait de l'utilisation d'un algorithme possédant une forte complexité, ici les fonctions liées au graphe, et aussi de la vision ensembliste naïve qui crée une grosse complexité.
L'algorithme le moins robuste entre Prim et Kruskal est Prim, notamment avec merge sort qui est très coûteux. Cependant à l'aide d'une binary heap, Prim est même plus efficace que Kruskal avec une binary heap. L'algorithme qui est le meilleur dans toute les situations pour obtenir un ARPM est donc ici Kruskal et merge sort.

Afin d'améliorer ces implémentations, une bonne idée est de modifier la structure de données et d'utiliser au lieu d'un id des sommets dans la structure arc, d'utiliser un pointeur vers ce sommet.