Master 1 - Ingénierie Informatique

Ensemble stable maximum

Emmanuel Herrmann
Bryce Marbois
Martin Rety

Table des matières

1 Exécution

Un jar est fourni, permettant l'exécution de l'algorithme. Le seul paramètre nécessaire est le nom du fichier .graphe (ainsi que son chemin s'il n'est pas situé dans le même répertoire que le jar.

Exemple : `java -jar ensemble_stable_maximum.jar test.graphe`

Si la syntaxe du fichier .graphe est exactement celle donnée en exemple, le fichier pourra être lu. Si cependant un graphe plus conséquent est donné, il est possible de retirer les virgules et les crochets de la liste de voisins pour avoir une écriture plus condensée. Cela est du au fait que nous découpons chanque ligne via le délimiteur ' '. Il est donc indispensable de garder les espaces et sauts de ligne entre chaque sommet. Les caractères '[', ']' et ',' sont automatiquement supprimés de la ligne, mais si d'autres caractères sont rajoutés dans le fichier, il ne seront pas supprimés par le parseur et pourront entraîner une erreur de lecture. Il est par contre obligatoire de noter les noms des sommets de 0 à n-1.

2 Implémentation

2.1 Structure du projet

Notre projet est organisé de la façon suivante : Une classe réservée à l'exécution de l'algorithme à proprement parler, quatre classes correspondant aux quatre briques, et une dernière classe graphe pour notre sutructure de données.

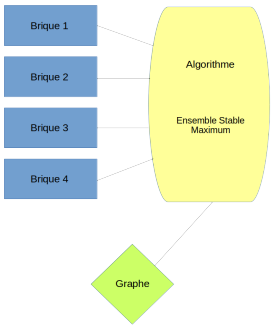

Figure 1. Structure

Le main appelle l'algorithme qui appelle à son tour les quatre briques suivant l'algorithme récursif de Fomin, Grandoni et Kratsch. Il retourne un entier en sortie (correspondant à la taille du plus grand sous-ensemble trouvé) qui est ensuite affiché dans la méthode correspondante.

Un objet graphe a été créé pour contenir la structure de données ainsi que plusieurs méthodes utilitaires. Un graphe est composé d'une Hashmap, dont la clé est un entier représentant le sommet, et la valeur une ArrayList d'entiers contenant les voisins de ce sommet. On a ainsi un accès très simple au sommet et à ses voisins, ce qui a beaucoup facilité l'implémentation de l'algorithme. De plus, l'ajout de plusieurs méthodes utiles permettent de manipuler cet objet très simplement. On peut trouver par exemple un textttoString() pour afficher proprement le graphe (ce qui a été particulièrement utile dans la phase de debug), des fonctions pour supprimer ou ajouter un sommet, voir son voisinage, ... Ce sont des méthodes reprises par les différentes briques, ce qui permet de factoriser le code.

Brique 1 La Brique 1 est construite de façon à retourner une composante connexe du graphe et le graphe sans cette composante. Ainsi, on ne parcourt le graphe qu'une seule fois (via un parcours en largeur), car il suffit de tester si le graphe sans composante connexe est vide ou non. S'il l'est, alors il n'y a pas de composante connexe et on peut poursuivre l'algorithme avec la brique 2.

Pseudo-code de la Brique 1 :

```
test(Graphe g)
Liste fifo;
sommetCourant = prendreUnSommet();
Graphe composanteConnexe;
Graphe gSansConnexe;
gSansConnexe.supprimerSommet(sommetCourant);

TantQue fifo n'est pas vide Faire
    sommetCourant=defiler(fifo);
    Pour tous les voisins v de sommetCourant Faire
        Si v n'a pas encore été rencontré
            Ajouter v à fifo;
            composanteConnexe.ajouterSommet(v);
            gSansConnexe.supprimerSommet(v);

Retourner composanteConnexe, gSansConnexe;
```

Brique 2 La brique 2 sest composée de deux méthodes : une de test pour voir s'il existe un sommet dominant, et une d'exécution pour retourner le graphe transformé.

Le test vérifie s'il existe un sommet possédant au moins le même voisinage qu'un autre sommet, et s'il sont voisin entre eux.

Pseudo-code de Test :

```
test(Graphe G){
    Pour tous les sommets s1 de G Faire
```

```
        Pour tous les sommets s2 de G Faire
            Si s1 != s2 Alors
                Si s1 et s2 sont voisins Alors
                    Pour tous les voisins v de s1 Faire
                        Si v est n'est pas voisin de s2 Alors
                            Retourner -1;
                    Retourner s2;
    Retourner -1;
}
```

Brique 3 Comme la brique précédente, celle-ci est composée de deux méthodes : une de test et une d'exécution. Le test vérifie si le graphe possède un sommet 2-pliable grâce à un parcours du graphe et de vérification du voisinage de chaque sommet. Run retourne le graphe transformé par la brique 3.

Pseudo-code :

```
Test(Graphe G){
Pour tous les sommets s de G Faire
    Si s est de degré 2 Alors
        Si les deux sommets voisins de s ne sont pas voisins entre eux Alors
            Retourner s;
        }
    }
    Retourner -1;
}
```

Brique 4 La brique 4 est séparée en deux parties : celle qui renvoie le graphe moins v et ses miroirs, et l'autre qui renvoie le graphe moins v est ses voisins.

Pseudo-code de miroir :

```
Miroir(sommet s, Graphe G){
    Liste sommetsASupprimer;
    sommetsASupprimer.ajouter(s);
    Liste voisin_distance2;
    Pour tous les voisins u de v Faire
        Pour tous les voisons w de u Faire
            Si w != v, v n'est pas un voisin de w et voisin_distance2 ne contient pas w
                voisin_distance.ajouter(w);
    Liste clique_a_tester;
    Pour tous les u dans voisin_distance2 Faire
        Pour tous voisins w de v Faire
            Si w n'est pas un voisin de u Alors
                clique_a_tester.ajouter(w);
```

```
        Si clique_a_tester est une clique Alors
            Si sommetsASupprimer ne contient pas u Alors
                sommetsASupprimer.ajouter(u);
    Pour tous les sommets u dans sommetsASupprimer Faire
        G.supprimerSommet(u);
    Retourner G;
}
```

3 Justification de la brique 3 (pliage)