Master 2 d'Informatique 2013-2014

Complexité Algorithmique Avancée

Couverture par Sommets

Enseignants :
Philippe DUCHON

Chargé de TD :
Alexandre K.ZVONKINE

Auteurs :
Thibaut THOMAS
Adrien LABORDERIE
Anthony GUEGUENOU
Julio MATARRANZ

TABLE DES MATIERES

I. Introduction

II. Algorithmes de générations
A. Arbres
B. Graphes
C. Graphes bipartis
D. Graphe ayant une petite couverture

III. Recherche de couverture
A. Glouton
B. Optimal pour arbre
C. Optimal pour bipartis
D. 2-approché pour graphes avec parcours en profondeur
E. 2-approché pour graphe avec élimination du voisin
F. Paramétrique pour graphe ayant une petite couverture

IV. MiniSAT
A. Présentation
B. Utilisation dans notre projet
C. Résultats

V. Tests
A. Tests unitaires
B. Tests de performances
C. Interprétation des résultats

VI. Conclusion

I. Introduction

Dans le cadre du cours de Complexité et Algorithmique Appliquée, il nous a été demandé d'implémenter en C++ des algorithmes qui sont à la fois optimaux pour des classes particulières de graphes, approchés pour des graphes quelconques, et de complexité acceptable sur le thème de la couverture par sommets.

II. Algorithmes de générations

A- Arbres

La génération d'arbre aléatoire est réalisée au travers d'un algorithme ayant en paramètres le nombre de sommets. Afin de pouvoir générer des arbres plus précis, notamment un arbre binaire, nous avons décidé de rajouter deux paramètres sous forme d'entiers qui sont le nombre minimum et le nombre maximum de fils qu'un sommet peut atteindre.

On peut déterminer la complexité de cet algorithme à travers deux étapes : tout d'abord la boucle for qui permet de créer le nombre de sommets souhaités est en O(n), et ensuite l'accès au vecteur qui stockera le sommet est en O(log(n)). On a donc une complexité en O(n*log(n)).

B- Graphes

Cet algorithme permet de générer un graphe quelconque. Il prend en paramètre le nombre de sommets voulus, et la probabilité d'avoir une arête entre deux sommets.
Nous avons une boucle *while* qui parcourt tous les sommets du graphe, et une boucle for qui parcourt tous les sommets déjà créés pour faire les arêtes ce qui donne une complexité en $O(n^2)$.

C- Graphes bipartis

Cet algorithme permet de générer un graphe biparti. Il prend en paramètre le nombre de sommet voulu, et la probabilité d'avoir une arête entre deux sommets. Nous utilisons une variable qui va délimiter les deux set du graphe. Cette variable prend une valeur aléatoire entre 1 et (nombre de sommet)-1. Par exemple, si nous voulons un graphe bipartis à 10 sommets et que notre variable délimitante vaut 4, il y aura 4 sommets dans un set et 6 dans l'autre.
Dans un premier temps l'algorithme créer tous les sommets. Il rajoute ensuite les arêtes entre les sommets du 1er set et ceux du 2ème set. A la fin de cette étape, il se peut qu'il y ai des sommets qui n'ai pas d'arête si la probabilité d'avoir une arête n'est pas de 1. C'est pourquoi nous avons rajouté une 3ème étape où on parcourt tous les sommets, et si il y en a un qui n'a pas d'arête, on lui en rajoute une avec un sommet aléatoire du set opposé.
Nous avons une boucle for pour créer tout les sommets; deux boucle for imbriquées (mais chacune ne parcourt pas la totalité des sommets) pour créer les arêtes; une boucle for qui parcourt tous les sommets.
Nous avons donc une complexité en $O(n+n^2+n) = O(n^2+n)$.

D- Graphe ayant une petite couverture

Cet algorithme permet de générer un graphe quelconque ayant une petite couverture donné. Il prend en paramètre le nombre de sommet voulu, la probabilité d'avoir une arête entre deux sommets, et la taille de la couverture.
Dans un premier temps l'algorithme créer tous les sommets. Ensuite, il crée des arêtes entre les sommets : couvrant/couvrant et couvrant/non-couvrant. Il ne peut donc pas y avoir d'arête entre deux sommets non-couvrant.
Nous utilisons une boucle for pour créer tous les sommets, puis deux boucles for imbriquées pour créer les arêtes, ce qui nous donne une complexité en $O(n^2)$.

III. Recherche de couverture

A- Glouton

Dans cet algorithme, on va chercher le sommet avec le plus grand degré et le supprimer, et ce tant qu'il reste des sommets avec un degré supérieur à 0 dans le graphe. On a donc une boucle while qui se termine quand il n'y a plus de sommet avec un degré supérieur à 0 (et qu'on a donc obtenu la couverture), et une boucle for à l'intérieur qui parcours tous les sommets restant afin de trouver celui avec le plus grand degré. On a donc une complexité en $O(n^2)$.

B- Optimal pour arbre

Définition préliminaire :

*« Un **couplage maximal** est un couplage M du graphe tel que toute arête du graphe possède au moins une extrémité commune avec une arête de M. Ceci équivaut à dire dans l'ensemble des couplages du graphe, M est maximal au sens de l'inclusion, i.e. que pour toute arête a de A qui n'est pas dans M,* $M \cup \{a\}$ *n'est plus un couplage de G. »*

Cet algorithme est basé sur le parcours en profondeur. A partir de n'importe quel nœud du graphe G donné, on fait de ce nœud la racine de l'arbre puis on parcourt ensuite entièrement l'arbre.
A chaque fois qu'on retourne un nœud/sommet fils, on vérifie si celui ci et son sommet parent ont été marqué. Si aucun des deux n'a été marqué alors on les marque et on marque également l'arête entre ces deux sommets, et on termine par ajouter le sommet parent à la couverture de sommet.
Lorsque le parcourt de l'arbre est terminé, nous retournons à la racine en ayant construit une couverture de sommet optimal dont la taille est égale au couplage maximal de l'arbre.

Chaque arête de l'arbre qui n'a pas été marqué est forcement adjacente à une arête qui a été marqué.

Étant donné que cet algorithme retourne une couverture par sommet de taille égale au couplage maximal de l'arbre, nous pouvons considérer que la couverture par sommet est optimal, étant donné qu'aucune couverture ne peut être plus petite que celle-ci.

La complexité globale de l'algorithme se résume à celle d'un parcours en profondeur classique, soit en $O(n+m)$ avec n le nombre de sommets et m le nombre d'arêtes.

C- Optimal pour bipartis

Nous n'avons pas réussis à implémenter cet algorithme pour le moment. Nous avons cependant distingué deux possibilités pour cette algorithme.
Première, la recherche d'une couverture minimale pour un graphe biparti est un problème NP-COMPLET. En revanche, le théorème de König nous dit que le calcul d'une couverture minimale pour un graphe biparti est équivalent au couplage maximal d'un même graphe, qui lui est plus facilement calculable. Une solution serait donc d'appliquer sur notre graphe un algorithme de couplage maximal puis de faire les traitement du théorème de König pour obtenir la couverture minimale.
Une autre solution est de transformer notre graphe en graphe de flux pour ramener notre problème à un problème de flux qui lui aussi est plus facilement calculable.

D- 2-approché pour graphes avec parcours en profondeur

Pour faire cet algorithme nous avons utilisé la version non récursive du parcours en profondeur donné en cours. L'algorithme applique ce parcours en profondeur sur le graphe et on récupère ainsi un arbre couvrant. On supprime alors tous les sommets de cet arbre qui n'ont pas d'arête et on obtient ainsi la couverture. Il y a une boucle for pour colorier tous les sommets en blanc; une boucle while qui permet de parcourir tous les sommets qui n'ont pas été traités (qui ne sont donc pas colorié en noir); une boucle for imbriqué dans la boucle while qui permet de traiter tous les voisins du sommet courant; et enfin une boucle for où l'on récupère dans notre couverture tous les sommets du graphe qui ont un degré supérieur à 0.

Cet algorithme a donc une complexité en O(m+n). (m étant les arêtes, n les sommets)
Pour un graphe qui comporte des sommets sans arête, il y a certain cas où notre algorithme ne trouve pas de couverture et nous n'avons pas réussis à corriger le problème pour le moment.

E- 2-approché pour graphe avec élimination du voisin

Pour cet algorithme, on prend le premier sommet qui a un degré supérieur à 0 et son voisin. On les ajoute à la couverture et on réitère l'opération jusqu'à ce qu'il n'y ai plus de sommet dans le graphe.
Il y a donc une seule boucle while, ce qui donne une complexité en O(n). On constate que cet algorithme est rapide mais il ne donne pas la meilleure des couvertures possibles.

F- Paramétrique pour graphe ayant une petite couverture

Cet algorithme prend deux paramètres : un graphe, et une taille de couverture souhaitée. Il va alors chercher dans un graphe ayant une petite couverture la plus petite couverture possible. Par exemple, pour un graphe ayant une petite couverture de 3, si l'on demande à notre algorithme de chercher une couverture de taille 3, il trouvera et donnera une couverture minimale de 3. Si on lui demande de chercher une couverture de taille 2, il retournera une couverture de taille 2 si elle existe, et rien sinon.
Cet algorithme utilise une fonction récursive pour créer la couverture qui sera testée lors des derniers appels récursif. Avant le renvoie du résultat, si une couverture a été trouvée, nous utilisons une dernière boucle for pour parcourir les sommets de la couverture et enlever les doublons.

IV. MiniSat

A- Présentation

« MiniSat est un minimal SAT solveur, développé pour aider les chercheurs et les développeurs. Minisat est distribué sous une licence MIT et est utilisé sur un grand nombre de projets. » - site internet *minisat.se* .

B- Utilisation dans notre projet

Pour utiliser MiniSat, nous créons d'abord un graphe à partir d'un fichier « *fileTest.txt* » contenu dans le répertoire *source.* Nous passons ensuite ce graphe en paramètre de notre fonction VertexCoverToSat().

MiniSat nous renvois les résultats dans le fichier minisat-result dans le répertoire racine de notre projet.

A partir de ce graphe nous créons le fichier input pour Minisat, pour cela nous stockons toutes les arrêtes dans un vecteur de pair<int,int> et inscrivons toutes ces arrêtes dans le fichier.

C- Résultats

Voici un exemple : Le contenu du fichier *fileTest.txt* est de la forme :

1: 3 4 5 6
2: 3 6 7
3: 1 2 5 7
4: 1 7
5: 1 3 7
6: 1 2
7: 5 4 3 2

Nous appelons ensuite l'exécutable de MiniSat à travers un objet de type Minisat que nous instancions, puis nous passons en paramètre notre graphe et nous obtenons en sortis un fichier *« minisat-result »* contenant :

SAT

-1 2 3 4 5 6 -7 0

Le traitement de MiniSat nous renvoie :

//

```
This is MiniSat 2.0 beta
WARNING: for repeatability, setting FPU to use double precision
============================[ Problem Statistics ]=============================
|                                                                             |
|  Number of variables:  7                                                    |
|  Number of clauses:    22                                                   |
|  Parsing time:         0.00        s                                        |
============================[ Search Statistics ]==============================
| Conflicts |          ORIGINAL         |          LEARNT          | Progress |
|           |    Vars  Clauses Literals |    Limit  Clauses Lit/Cl |          |
==============================================================================
|         0 |       7       22       44 |        7        0   -nan |  0.000 % |
==============================================================================
Verified 22 original clauses.
restarts              : 1
conflicts             : 0            (-nan /sec)
decisions             : 3            (0.00 % random) (inf /sec)
propagations          : 7            (inf /sec)
conflict literals     : 0            (-nan % deleted)
Memory used           : 4.43 MB
CPU time              : 0 s

SATISFIABLE
```

//

Et l'interprétation que nous en faisons à travers le retour de notre fonction *SatToVertexCover()* est la couverture par sommet suivante :

Couverture
2
3
4
5
6

V. Tests

A- Tests unitaires

Nous avons testé notre structure *Graph* pour garantir le bon fonctionnement des méthodes de cette classe. Vous pourrez trouver de plus amples détails dans notre fichier *graphTest.cpp* situé dans le dossier *testUnitaires*.

B-Tests de performances

Nous avons réalisé ces tests sur une machine du cremi salle 001 (4 cœurs).
Ci-dessous le retour de nos tests de perfomances.

Algorithme sur 1000 sommets	**Temps**
TreeOptimal	0.030000 seconds
twoAppr_neighbourhood	2.300000 seconds
twoAppr_depthSearch	23.470000 seconds
glouton	2.640000 seconds
opti_para_minCoverGraph	> 660.670 seconds

Algorithme sur 2000 sommets	**Temps**
TreeOptimal	0.110000 seconds
twoAppr_neighbourhood	17.890000 seconds
twoAppr_depthSearch	167.500000 seconds

glouton	20.300000 seconds
opti_para_minCoverGraph	> 660.670 seconds

C- Interprétation des résultats

Nous constatons que l'algorithme le plus rapide est l'algorithme de recherche de couverture de sommets optimal sur l'arbre.

Pour la recherche de couverture sur un graphe quelconque nous constatons que l'algorithme le plus rapide est le "2-approché_neighbours". Cela est logique car cette algorithme n'est pas très compliqué. Il y a seulement une boucle while ce qui le rend rapide d'exécution. En revanche, son défaut est qu'il ne renvoi pas du tout les couvertures les plus petites possible.
L'algorithme glouton est légèrement plus lent mais il renvoi généralement de meilleur résultat (c'est à dire une couverture plus petite).
L'algorithme "2-approché_depthSearch" est quant à lui beaucoup plus lent ce qui le rend difficilement utilisable pour des graphes très grand.
L'algorithme le plus rapide est le "optimalTree" qui est vraiment beaucoup plus rapide que les autres.
L'algorithme censé être le plus performant, "opti_param_smallCoverGraph", n'est pas utilisable sur des grands graphes à cause de sa lenteur d'exécution (en effet, il y a de la récursion et de nombreuses boucles).

VI. Conclusion

Le développement et les résultats de ce projet nous ont appris que chaque algorithmes a ses points forts et ses points faibles. Le 2 approché avec éliminations des voisins est rapide mais sa couverture de sommet retourné n'est pas optimale, à contrario l'algorithme glouton est beaucoup plus lent mais permets de renvoyer une couverture par sommet minimale.

Ce projet nous a permis de réaliser que le choix de l'architecture et des structures est un choix primordiale tant la complexité peut devenir grande en fonction de ce choix.