Rapport du projet d'OPTI2

RÉSOLUTION APPROCHÉE ET EXACTE D'UN PROBLÈME D'ARBRE COUVRANT

Adrien BLASSIAU

ENSIIE - 5 janvier 2020

Sommaire

INTRODUCTION

On cherche à résoudre de deux approches différentes le problème **(MBV)** qui consiste à trouver dans un graphe un arbre couvrant avec un minimum de sommets de degré > 2 :

- avec une heuristique, la méthode MBVST (**partie 2**)
- avec une formulation du problème en programme linéaire que l'on résout ensuite avec `GLPK` (**partie 3**)

Dans un premier temps (**partie 1**), on va s'intéresser à la génération aléatoire d'un graphe connexe ayant certaines propriétés intéressantes. On utilisera les graphes générés dans cette partie pour effectuer des tests comparatifs entre les méthodes exactes et approchées et ainsi **évaluer les performances** des algorithmes implémentés.

On précise que toutes les fonctions importantes du projet ont été testées avec `CUnit`. Une documentation du projet avec `Doxygen` est aussi disponible. Je vous invite à lire le fichier `README.md` dans lequel des informations pour lancer les différents exécutables sont présentes. Enfin, tous les résultats sont disponibles dans le fichier `output.txt` si vous ne souhaitez pas lancer les exécutables.

1. GÉNÉRATION DE GRAPHES ALÉATOIRES

Un résumé **très** condensé de cette partie peut être obtenu en lançant `bin/main`. L'output de cet exécutable est présent dans le fichier `output.txt` si vous ne voulez pas le lancer.

Une structure de données contenant un ensemble d'informations relatives à un graphe g a été implémentée. Elle est définie dans le fichier `graph.c`.

1. La fonction demandée se nomme `roy_warshall` et est présente dans le fichier `src/random_graph.c`. En argument de cette fonction, on passe la matrice d'adjacence du graphe contenue dans la structure de données définie plus haut.

Remarque : En entrée de la fonction, le sujet parle d'un graphe orienté alors que l'on travaille avec des graphes non orientés. La plupart des fonctions développées ne seront adaptées qu'au graphe non orienté car ce sont les seuls qui nous intéresse dans la quasi totalité du sujet.

2. On teste l'algorithme sur le graphe orienté suivant :

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

On obtient :

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

Le graphe initial est **fortement connexe** donc le résultat obtenu est logique. En effet, il existe un chemin entre toute paire de sommets dans le graphe initial.

3. L'algorithme de Roy-Warshall relie avec une arête les sommets pour lesquels il existe un chemin les reliant. Pour tester que deux sommets i et j sont dans la même composante connexe, il suffit donc de vérifier qu'après avoir appliquer Roy Warshall sur un graphe G qui donne G', qu'il y a un 1 dans la case de coordonnées (i,j) de la matrice d'adjacence associée à G'.

Les trois fonctions liées à cette question sont toutes dans le fichier `src/random_graph.c` :

- `test_x_y_strongly_connected` pour un graphe orienté
- `test_x_y_connected` pour un graphe non orienté
- `test_connected_v2` (et `test_connected`) qui appelle `test_x_y_connected`

4. La fonction demandée se nomme `generate_random_graph` et est présente dans le fichier `src/random_graph.c`.
Une **première version** avait été implémentée : elle construisait un graphe de manière aléatoire avec le bon nombre d'arêtes et de sommets et s'il n'était pas connexe, elle le jetait et en construisait un autre. Cette approche permet de construire un graphe de manière aléatoire mais prend beaucoup de temps pour les graphes avec beaucoup de sommets (plus d'une minute pour les graphes de plus de 100 sommets). J'ai donc opté pour une **deuxième version** qui met en place deux améliorations.

Première amélioration, soit $G(V, E)$ un graphe. L'**algorithme de Roy-Warshall** a une complexité en temps en $\mathcal{O}(|V|^3)$ pour construire la matrice d'adjacence du graphe G' qui renseigne sur l'existence d'un chemin entre deux sommets. Un simple **parcours en profondeur (DFS)** permet de vérifier que deux sommets sont dans la même composante connexe en $\mathcal{O}(|V| + |E|)$. **Un premier gain de temps considérable est possible en remplaçant Roy-Warshall par un DFS**.

Deuxième amélioration, une fois un graphe généré, on ne le jette pas s'il n'est pas connexe. On décide de relier ses n composantes connexes entre elles avec $n - 1$ arêtes et d'en supprimer $n - 1$ de manière aléatoire. On applique ensuite récursivement l'algorithme de vérification.

Remarque : Vous pouvez changer la méthode utilisée avec le paramètre `method` de `generate_random_graph` (0 pour Roy-Warshall et 1 pour DFS). On utilise le DFS par défaut car il est bien plus rapide que Roy-Warshall.

5. On obtient les densités suivantes :

- La densité d'un graphe de taille 20 est : 34
- La densité d'un graphe de taille 50 est : 73
- La densité d'un graphe de taille 100 est : 129
- La densité d'un graphe de taille 400 est : 459
- La densité d'un graphe de taille 600 est : 674
- La densité d'un graphe de taille 1000 est : 1095

On génère les graphes demandés. Cette génération est rapide, elle prend moins de 1 seconde grâce aux améliorations apportées plus haut. Les graphes se trouvent dans le fichier `output.txt`.

Remarque : Le sujet demande 10 graphes mais on ne dispose que de 6 tailles. J'ai donc généré 6 graphes qui se trouvent dans le fichier `output.txt`.

2. Résolution approchée

L'ensemble du code de l'algorithme **MBVST** se trouve dans le fichier `heuristic.c`. La fonction principale s'appelle `MBVST`. On a respecté le pseudo-code fournit. L'algorithme fonctionne et fournit de bons résultats.

Pour plus d'informations sur les résultats obtenus avec les graphes générés, je vous invite à lire la fin de la partie 3 qui traite de cela.

3. Résolution exacte

Soit $G(V, E)$ un graphe avec $|V| = n$ et $|E| = m$.

Remarque : Le sujet utilise E sans le définir. On imagine qu'il représente le même ensemble que $\mathcal{E}$.

1. On explique chacune des contraintes :

- **contrainte 1a** La contrainte nous assure que notre graphe aura $n - 1$ arêtes **sans assurer la connexité**.
- **contrainte 1b** La contrainte formalise le fait que si l'arête (i, j) est dans la solution, alors, si on l'enlève, tout sommet k est soit dans la même composante que i, soit dans la même composante que j mais pas dans les deux, sinon le graphe est toujours connexe et on n'a pas un arbre ce qui est absurde (car un arbre est un graphe connexe minimal) !
- **contrainte 1c** Cette contrainte formalise le fait que si l'arête (i, j) est dans l'arbre et que l'on enlève une arête (i, k) où k est différent de i et de j, alors j n'est pas dans la même composante que k. Par l'absurde, si j était dans la même composante que k, il existerait deux chemins qui relieraient i et j dans l'arbre :
  - l'arête (i, j)
  - le chemin empruntant l'arête (i, k) puis allant de k à j (on a supposé que k et j sont dans la même composante connexe donc il existe un chemin les reliant).

  Or s'il existe deux chemins différents reliant deux sommets, on a un cycle, ce qui est absurde car le graphe est un arbre. Donc cette contrainte permet d'éviter la présence de cycle.

2. Avec les contraintes **b** et **c** réunies on évite d'avoir des cycles dans la solution.

3. On obtient un sous graphe sans cycle avec **b** et **c**, en rajoutant la contraintes **a**, il possède n-1 arêtes. On obtient donc un **arbre couvrant** de G

4. La variable z_i indique si le sommet i est un sommet de branchement ou non. On en veut le moins possible, donc on minimise la somme des z_i dans l'objectif. La **contrainte d** met automatiquement à 1 z_i si le sommet i a plus de 2 voisins dans la solution car si il reste à 0, la quantité à gauche de $\leq$ est strictement plus grande que 2.

5. Il y a $m + 2 * m * n + n$ **variables** et $1 + n * m + m + n$ **contraintes**.

6. Le programme linéaire a été implémenté avec `GLPK`. Il est disponible dans le fichier `lp.c`.

Remarque : L'écriture d'un PL en `GLPK` (et encore plus en `CPLEX`) en `C` est excessivement fastidieuse et, je pense, une réelle perte de temps. Il me semble bien plus rapide et adapté d'utiliser ces outils via une interface dédiée.

7. Voir question 8.

8. On obtient les résultats suivants :

	MBVST	PL
20	1	0
50	5	3
100	13	9
400	69	>26
600	110	?
1000	200	?

J'ai arrêté `GLPK` au bout de trois heures pour le PL du graphe de taille 400. Les résultats sont aussi disponibles dans le fichier `output.txt`. On remarque que l'heuristique proposée fournit de bons résultats.

4. Conclusion

Ce projet a clairement mis en lumière plusieurs manières d'aborder un problème `NP-Complet`. On obtient des écarts entre les solutions approchées et exactes. C'est en fonction du contexte et des contraintes imposées que l'on choisira une approche de résolution plutôt qu'une autre.

Je voulais finir par préciser que le projet ne se finissait pas en une après-midi, comme vous nous l'aviez indiqué, très loin de là. Il m'a pris **plusieurs jours complets**, sans compter le temps que j'ai du prendre pour expliquer certains points à des camarades.