École Polytechnique de Montréal

Département de génie informatique et génie logiciel

INF4705

Analyse et conception d’algorithmes

Travail pratique 3 : Rapport

Soumis par

Francis de Lasalle, #1706231

Zihui Zhong, #1687994

13 avril 2016

Table des matières

[Introduction 2](#_Toc446042585)

[Description du jeu de données 3](#_Toc446042586)

[Résultats expérimentaux 3](#_Toc446042587)

[Analyse et discussion 7](#_Toc446042588)

[Analyse asymptotique des algorithmes 7](#_Toc446042589)

[Algorithme vorace probabiliste 7](#_Toc446042590)

[Algorithme de programmation dynamique 7](#_Toc446042591)

[Heuristique d’amélioration locale 7](#_Toc446042592)

[Analyse empirique des algorithmes 7](#_Toc446042593)

[Algorithme vorace probabiliste 7](#_Toc446042594)

[Algorithme de programmation dynamique 7](#_Toc446042595)

[Heuristique d’amélioration locale 8](#_Toc446042596)

[Qualité des algorithmes 8](#_Toc446042597)

[Conclusion 9](#_Toc446042598)

[Références 9](#_Toc446042599)

# Introduction

Ce troisième travail pratique est une compétition visant à produire le meilleur algorithme possible pour résoudre un certain problème donné. La compétition se déroule entre les étudiants du cours INF4705.

Le problème consiste à trouver le meilleur ordonnancement possible pour minimiser le nombre d’écoliers dans une file ayant leur vue obstruée, tout en respectant certaines contraintes. Plus précisément, le problème se présente par une liste contenant la hauteur d’écoliers et une liste contenant des paires de numéros d’écoliers pouvant être jumelés. Les écoliers regardent tous devant eux et l’objectif est de minimiser le nombre d’écoliers ayant leur vue obstruée par un écolier plus grand devant lui (directement ou plus loin). De plus, un écolier peut être précédé ou suivit d’un autre écolier seulement si les deux figurent dans l’une des paires de numéros d’écoliers pouvant être jumelés.

Le choix d’algorithme à implémenter est à la discrétion des étudiants. L’algorithme sera exécuté pendant 3 minutes et la meilleure solution obtenue au terme de ces 3 minutes sera retenue. Il convient donc de judicieusement utiliser le temps nous étant imparti.

# Description du jeu de données

Le jeu de données se présente sous la forme de fichiers texte dont la première ligne indique le nombre d’écoliers et la seconde ligne le nombre de paires d’écoliers pouvant être jumelés. Le fichier se poursuit avec la liste de la hauteur de chaque écolier (ligne par ligne), puis la liste des paires d’écoliers pouvant être jumelés (ligne par ligne).

Plusieurs exemplaires sont fournis, ayant différents nombres d’écoliers et de paires (liens).

# Présentation de l’algorithme

## Présentation sous forme de pseudocode

L’algorithme peut-être brièvement représenté sous forme de pseudocode ainsi :

|  |
| --- |
| backtrack (List<Node> currentIn, Node currentNode)  Si currentIn comprend toutes les nodes  retourne true  Si valide(currentIn)  pour toutes les nodes n avec un lien avec currentNode  si n pas dans currentIn  ajoute n dans currentIn  si aucun thread est en attende de travail  si (backTrack(currentIn, n))  retourne true  sinon  envoie backTrack(currentIn, n) au thread  enlève n de currenIn  retourne false  valide(List<Node> currentIn)  vérifie que le graph est encore connexe si on enlève ces nœuds avec un DFS. (O(n))  consecutiveNodesSwap()  pour opt de 1 à 20  pour toute combinaison de nœud a,b dans le fil actuel  échanger opt nœud à partir de a avec opt nœuds à partir de b  si meilleur résultat et échange valide  conserver l’échange  sinon  Éffacer l’échange  nOptimisation(int nbOpt)  pour toute combinaison de n nœuds  créer nbOpt+1 sous fil avec le fil actuel  pour toute les permutations possibles avec ces nbOpt+1 sous fils  si meilleur résultat et valide  remplace le fil actuel par la permutation. |

## Analyse de complexité théorique

Backtrack :

Complexité de en meilleur cas (le premier parcours trouve la réponse.) et de en pire cas. Le cas moyen est très difficile à estimer car il dépend de la configuration des liens et des nœuds en plus de leurs nombres. Notre fonction de validation a une complexité de O(n) car c’est un parcours DFS et peut éliminer beaucoup de temps de calcul, mais encore une fois, cela dépend de la configuration.

consecutiveNodesSwap :

Complexité d’environ , choix de 2 nœuds dans la fil. La vérification a un temps d’exécution de pour vérifier la connexion et pour calculer le résultat dans le cas d’une connexion valide. Comme le nombre de connections est basse comparé au nombre d’éléments et les cas où la connexion est valide rare. Seul le choix est considéré.

nOptimisation

Complexité d’environ , choix de nbOpt nœuds dans la fil au lieu de 2, le reste de l’exécution est identique à consecutiveNodeSwap.

# Justification de l’originalité de l’algorithme

Le problème peut être abstrait à celui d’un parcours de graphe. En effet, chaque nœud du graphe est représenté par un écolier et chaque lien par une paire permise d’écoliers. Par exemple, pour 5 écoliers où l’écolier 1 peut être jumelé avec l’écolier 3 et 4, l’écolier 2 avec l’écolier 3 et 5 et l’écolier 3 avec l’écolier 4, le graphe ressemblerait à :

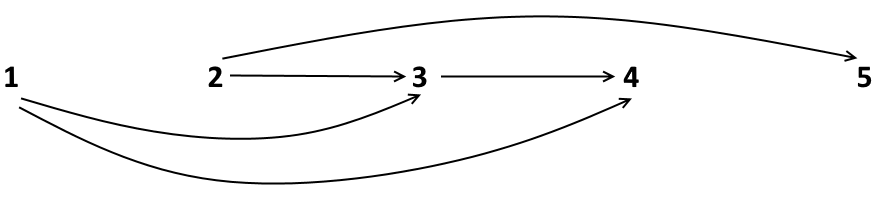


Figure 1. Abstraction du problème sous forme de graphe, notez que le graphe en réalité n’est *pas* orienté

L’objectif peut se reformuler à trouver une chaîne hamiltonienne qui minimise le nombre d’écoliers ayant leur vue obstruée. En effet, une chaîne hamiltonienne est une chaîne parcourant chaque élément passant par chaque élément du graphe une et une seule fois; la chaîne hamiltonienne abstrait donc le concept de file d’écoliers.

Trouver une telle chaîne se fait en faisant une fouille en profondeur du graphe. Un écolier est choisi et, parmi les écoliers pouvant être jumelés à celui-ci, un autre est choisi (s’il n’est pas déjà dans la chaîne) et ainsi de suite (cette sélection se fait de façon récursive). Les écoliers sont d’abord triés en ordre croissant de taille, afin que les premiers écoliers considérés dans la file étant construite soient les plus petits. La liste des liens des écoliers est aussi triée selon la taille de l’écolier liée. Cette astuce très simple permet de drastiquement améliorer la solution initiale. En effet, il est évident que la solution sera plus optimale en choisissant d’abord les plus petits écoliers.

À chaque ajout d’un écolier à la file d’écoliers, celui-ci est enlevé des écoliers restant dans le graphe et l’algorithme teste si le graphe restant (avec ces nœuds en moins) est toujours connexe. Il s’agit de l’astuce pour effectuer le retour arrière (« backtracking »). En effet, si le graphe restant après avoir enlevé les écoliers ajoutés à la file en construction n’est plus connexe, c’est qu’il n’existe aucune chaîne passant par chaque nœud et donc. Dans ce cas, il est impossible que les écoliers restants dans le graphe puissent être jumelés ensemble. Si la vérification de connexité échoue, l’algorithme retourne en arrière, c’est-à-dire qu’il considère le nœud suivant au dernier nœud ajouté plutôt que celui ayant fait échouer le test de connexité. Si le test de connexité réussit, l’algorithme considère un autre des nœuds pouvant être jumelé au dernier ajouté et ainsi de suite jusqu’à ce que la chaîne contienne tous les écoliers. Le test est effectué seulement si la taille du fil actuel est assez basse, pour éviter une perte de performance.

Comme cette recherche peut-être longue et comme il convient de maximiser l’utilisation de ressources matérielles disponibles pour optimiser la solution, cet algorithme est parallélisé sur plusieurs fils d’exécution différents (*threads*), afin de répartir la charge de travail. Une telle implémentation n’est pas triviale du tout, mais permet d’utiliser efficacement les ressources à notre disposition.

Une fois une première chaîne trouvée par l’algorithme de retour arrière, celle-ci est gardée comme point de départ d’une batterie d’algorithmes d’optimisation locale (heuristiques d’optimisation locale).

Le premier algorithme utilisé est une version « allégée » d’un algorithme « complet » de permutation de nœuds. La méthode nOptOptimization() permet d’échanger de place deux séquences de nœuds consécutifs au sein de la chaîne. Le nombre de nœuds échangés de place va de 1 à la moitié de la taille de la chaîne. L’exemple suivant illustre l’un des 3 échanges qui seraient considérés dans une chaîne de 5 éléments si un échange de 2 nœuds consécutifs est fait :

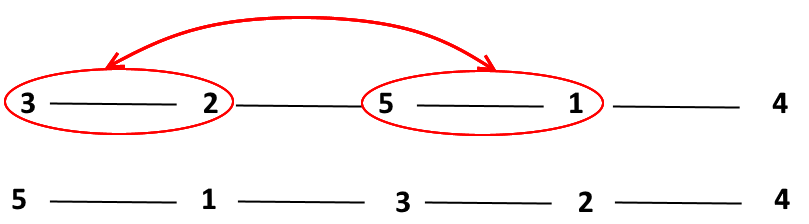


Figure 2. Exemple de l’algorithme d’optimisation locale par échange de nœuds

Dans cet exemple, l’algorithme tenterait aussi de faire 10 changements de place de 2 paires d’un nœud et aucun changement de place de paires de plus que 2 nœuds n’est possible.

Cette heuristique « maison » s’est avérée, selon nos tests, être fort performance et très peu coûteuse en temps de calcul. La complexité asymptotique de cet algorithme est en , ce qui correspond à la complexité d’un algorithme 2-opt « complet », mais permet de tester plusieurs transformations très variées. Nous considérons qu’il s’agit d’une heuristique originale et performante.

Cet algorithme s’exécute tant qu’une amélioration est observée. Plus précisément, des groupes de 1 à la taille maximale de nœuds sont échangés et l’algorithme recommence à 1 tant qu’une amélioration a été faite. La taille maximale des blocs de nœuds pouvant être échangés est fixée arbitrairement à 20, c’est-à-dire que pour de petites chaînes (moins de 40), l’algorithme testerait des échanges allant jusqu’à la moitié de la taille de la chaîne. En pratique, aucune amélioration n’a été observée lors d’échanges de groupes de plus de 20 nœuds (des améliorations ont été observées jusqu’à environ 16). Il est donc vraisemblablement inutile d’essayer d’échanger des groupes de plus 20 nœuds et cette limite existe donc afin de ne pas perdre inutilement de temps.

Lorsque cet algorithme est terminé, un second algorithme d’optimisation locale est lancé sur la chaîne précédemment optimisée. Il s’agit d’un algorithme récursif permettant de faire un *n*-opt « complet », où *n* va de 2 au nombre de liens dans la chaîne. Par *n*-opt, on entend un 2-opt où 2 liens sont brisés et toutes les combinaisons pouvant être reformées sont testées, pour un 3-opt, 3 liens sont brisés et toutes les combinaisons pour reformer la chaîne sont testées, etc. Un 2-opt « complet » a une complexité asymptotique en , un 3-opt complet une complexité asymptotique en , etc. Considérant le temps qui nous est imparti, un tel algorithme devient vite trop coûteux, mais il s’agit d’une étape finale d’optimisation. Après le 2-opt « complet », l’algorithme précédent d’optimisation est relancé pour essayer rapidement d’améliorer la solution, puis le 2-opt « complet » est relancé et ainsi de suite jusqu’à ce que ni l’un ni l’autre n’améliore la solution, c’est alors que le programme passe à un 3-opt, puis un 4-opt. Il s’agit donc d’une ultime tentative pour tenter d’améliorer la solution.

Nous croyons que le programme dans son ensemble est d’une bonne originalité et efficacité. En effet, une solution de bonne qualité est très rapidement retournée par la fouille en profondeur, grâce au tri initial des hauteurs. Dans tous les cas, la solution est la plus optimisée possible, selon les heuristiques utilisées. C’est à dire que, même pour les plus petits fichiers, le temps maximal d’exécution (3 minutes) sera utilisé (par les *n*-opt « complet »). Une solution de qualité est donc rapidement retournée pour de gros exemplaires et pour les petits, la solution est tentativement améliorée le plus possible.