The Prize-collecting Steiner Tree Problem and Underground Mine Planning

by

Weibei Li

A research paper
presented to the University of Waterloo
in partial fulfillment of the requirements for the degree of
Master of Mathematics
in
Computational Mathematics

Supervisor: Nicholas C. Wormald

Waterloo, Ontario, Canada, 2010

Abstract

Planning the extraction of ore from a mine is not an easy problem. In today's high competition for capital, a high level of gurantee would be required with regard to profit maximization and cost minimization. In this paper, we model the underground mine planning problem by the K-cardinality Prize-Collecting Steiner Tree (KPCST) problem, which is a special case of the standard Prize-Collecting Steiner Tree (PCST) problem on graphs. The problem is NP-hard and a polynomial-time algorithm does not exist unless $P = NP$. KPCST permits a positive cost on the edges and a positive prize on the vertices. The vertices with non-zero prize are called profitable vertices. KPCST asks for a subtree with exactly K profitable vertices maximizing the sum of the total prizes of all vertices minus the sum of the total costs of all edges in the subtree. The main application considered in this paper is the planning of ore extraction in an underground mine.

We first transform the original undirected graphs to directed graphs, concentrating on the formulation on directed graphs, which is based on the one in Ljubić et al. [31] and propose an exact algorithm, which models the KPCST problem as an IP problem and uses the branch-and-cut algorithm. The algorithm relaxes an exponential set of connectivity constraints and retains only the violated constraints found by a maximum flow algorithm. We also present a heuristic algorithm in order to try to reduce the computational time of the branch-and-cut method. We then implement the proposed algorithm and test its performance on simulated mining problems. A practical mining grid graph was tested under different cases of the edge costs. Finally, up to 81 vertex instances have been solved to optimality in several seconds.

Acknowledgements

I would like to thank Nicholas C. Wormald for all of his guidance with this project.

Dedication

This is dedicated to my family and the one I love.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Description

Planning and designing the extraction of ore from a mine is not an easy problem. In today's high competition for capital, a high level of guarantee would be required with regard to profit maximization and cost minimization. In a designing scenario of the mining problem, the input network is a set of interconnected tunnels which provides access to ore zone and haulage of ore from the designated ore zone to the mill. The goal is to design underground mine layouts minimizing associated costs. Thomas et al. [39] described a mathematical network model and used two software tools for designing in two industry case studies. One of the softwares called Underground Network Optimization tool (UNO) was applied to design an extension to an Australian gold mine. The other one called Decline Optimization Tool (DOT) was used to find an efficient decline for accessing a large orebody. In the designing phase, the working structure of an underground mine which is a set of interconnected tunnels is mainly considered, whereas the ore to mine, which has been decided by the planning phase, is not considered.

For planning, the orebody is first broken down into mineable blocks of ore called *stopes*. In a typical planning scenario of the mining problem, the input is a set of stopes to be extracted and a potential network for extracting the ore. The net profit can be estimated for each stope, and costs of the network are dominated by the labor and other charges for reaching the stopes underground. Because of the geographical situation, the costs on the path to each stope are different. In this paper, we will focus on the planning phase of the mining problem.

The determination process to plan the extraction of the ore faced by a profit oriented company consists of two parts:

1. A subset of stopes with high profit has to be selected, and the size of the subset is limited to a certain number K. In practice, different scenarios can be planned according to different K values. Moreover, the number K is constrained by various other considerations, such as bottlenecks in traffic if too many stopes are mined in one area. So for simplicity, we are assuming that the number of stopes to be selected is exactly K.

2. The mining network may connect to the existing network or the surface and the network has to be designed to connect all selected stopes in a cost-efficient way, i.e., there must be tunnels to the selected stopes and the cost of tunnels forms a negative component of the total profit function to be maximized.

The problem above has a natural formulation on graph theory, where the graph corresponds to the mining map, with vertices of the graph representing the stopes of the ore and the intersection of the tunnels, and the edges representing the tunnels. The prize p associated with a vertex is an estimation of the net return by extracting that stope. The vertices corresponding to tunnel intersections have no prize. The cost c associated with an edge is the cost of opening up the tunnels. The aim is to maximize the sum of the prize p of the vertices minus the sum of the cost c of the edges.

The definition of the problem to be addressed in this essay is given as follows:

K-cardinality Prize-Collecting Steiner Tree (KPCST) problem on a graph with edge costs and vertex prizes (as mentioned before, prize is a number associated with a vertex and the number is estimated by the profit of extracting the corresponding stope in the mining problem) asks for a subtree with K vertices maximizing the sum of the total prize of all vertices in the subtree minus the sum of the total cost of all edges in the subtree.

The classical Prize-Collecting Steiner Tree (PCST) problem does not require the number of profitable vertices in the solution while KPCST problem takes into account the cardinality of the set of profitable vertices in the solution. Therefore, the KPCST problem constitutes a generalization of the PCST problem. Of course, one may solve PCST by repeatedly solving the K-cardinality problems for $K = 1, ..., |V|$, where $|V|$ is the number of vertices in the graph.

Given an undirected graph $G = (V, E)$, let c_e which is a non-negative number be the cost on the edge e, and let p_v which is a non-negative number be the prize on the vertex v. The K-cardinality Prize-Collecting Steiner Tree (KPCST) Problem is to find a connected subgraph $T = (V_T, E_T)$ of G, containing exactly K profitable vertices, that maximizes the objective function

$$NW1(T) = \sum_{v \in V_T} p_v - \sum_{e \in E_T} c_e \tag{1.1}$$

or equivalently, that minimizes the objective function

$$NW2(T) = \sum_{e \in E_T} c_e - \sum_{v \in V_T} p_v. \tag{1.2}$$

In other words, the goal is to find a connected subgraph $T = (V_T, E_T)$ of G, containing exactly K profitable vertices, that minimizes the objective function

$$GW(T) = \sum_{v \notin V_T} p_v + \sum_{e \in E_T} c_e. \tag{1.3}$$

When the objective function (1.1) or (1.2) is used, the problem in the literature (e.g., Johnson, Minkoff and Phillips [27]) is known as the Net-Worth maximization (NW) problem. On the other hand, when the objective function (1.3) is used, the problem is sometimes referred as the Goemans-Williamson minimization (GW) problem (Goemans and Williamson [23]). From the optimization point of view, the two problems are equivalent. However, as far as the computation of the worst case performance ratio of approximation algorithms is concerned, they are not equivalent. Feigenbaum, Papadimitriou and Shenker [20] have shown that it is NP-hard to derive an approximation within any constant factor for the NW problem, while for the GW problem there exists several approximation algorithms which we will present later in this section. In this essay, we concentrate on (1.3) as the objective function which is also considered in Ljubić et al. [31], Goemans and Williamson [24], Lucena and Resende [33] and Canuto, Resende and Tibeiro [6].

It is obvious that the optimal solution T is a tree. An intuitive explanation is that if the solution is not a tree, since it is connected, it must have a cycle. Then we can remove an edge from the cycle to decrease the objective function without violating the connectivity requirement. We have two kinds of vertices throughout this paper: profitable vertices as mentioned before, defined as $R = \{i \in V \mid p_i > 0\}$; and non-profitable vertices with $p_i = 0$. Throughout this essay, we assume that $R \neq \emptyset$.

The goal of this paper is to present an exact algorithm to solve the planning of the mining problem to optimality within a reasonable time, rather than finding a lower bound or providing heuristic methods. As a matter of fact, Philpott and Wormald [35] introduced a heuristic method for determining the optimal extraction of ore and waste from an open-cast mine. The heuristic method may be useful for a given application, especially for the large size instances. But it is also important to test exact algorithms.

1.2 Investigation and Adaptation

In order to find an exact algorithm to solve the mining problem to optimality, we investigate much related previous work which will be discussed in Chapter 2. We follow the work by

Ljubić et al. [31] which is to design networks for planning and expansion of heating service and make some necessary adaptations for the planning of the mining problem.

In Ljubić et al. [31], they modeled the heating problem by the PCST problem. They gave an integer linear program (ILP) formulation on a transformed directed graph using connectivity inequalities and applied the branch-and-cut method to solve LP-relaxation of the ILP problem. The separation of sets of violated inequalities was generated by a maximum flow algorithm. A preprocessing method which reduced the size of many instances significantly was introduced. They have solved instances of up to 2500 vertices and 62,500 edges, most of which to optimality with a decrease of two orders of magnitude in the computational time comparing with Lucena and Resende [33].

In our paper, we used the similar formulation and main algorithms: a branch-and-cut algorithm to solve the IP problem and a separation algorithm to find the violated constraints. Considering this particular mining problem in practice, however, we need to do some adaptation of the formulation and algorithms to our problem.

First, we modeled the mining problem by the KPCST problem which requires exactly K profitable vertices to be selected. With this extra restriction, most of the preprocessing methods introduced in Ljubić et al. [31] are not suitable since the vertices or the edges cannot be simply discarded if K profitable vertices are required in the solution.

Second, we discuss a heuristic method based on that in Philpott and Wormald [35] to reduce the computational time.

The third, as mentioned in Section 1.1, the mining network must connect to the existing network or the surface, that is, a point of access is chosen from the existing network or the surface, and the network joining the stopes will be connected to the access point. At this point, the rooted version where the root represents the access point must be considered. These problems are discussed in Section 3.4.

Finally, we implemented the main algorithms with CPLEX. We tested about 200 instances under our test environment: a desktop PC with 64-bit AMD Phenom(tm) 9950 Quad-Core Processor 2.20GHz and 6.00 GB RAM. All the instances of up to 81 vertices are solved to optimality in several seconds. We analyzed the main barriers to solve large size problems and provided some possible improvements.

1.3 Remainder of Paper

The paper is organized as follows. In the next chapter, we give a literature survey on the related problems and various variants of the problems, and illustrate their real-world applications. The integer linear programming formulations for a directed graph of the problem is presented in Chapter 3. We introduce some variants of the mining problem: the rooted problem and the essential vertex problem in Chapter 3. A brief description of the algorithm and the details of the algorithm are explained in Chapter 4. Computational results are discussed in Chapter 5 and finally the conclusions are identified in Chapter 6.

Chapter 2

Literature Survey and Applications

2.1 Classification of Related Problems and Previous Work

The Steiner Tree (ST) problem (see, e.g., S. Chopra et al. [11] and A. Lucena [32]) on a graph with edge costs is expected to find the minimum edge cost tree for a given set of *terminal* vertices. The *terminal* vertices here means the vertices which are required to be included in the solution. The ST problem has been shown to be NP-hard by Garey, Graham and Johnson [22] and hard even to approximate. There is a 1.55-approximate algorithm due to Robins and Zelikovski [37]. Recently, Byrka et al. [5] proposed an improved approximation algorithm for the Steiner tree problem but approximation within 95/94 is known to be NP-hard (Chlebik and Chlebikova [10]). Charikar et al. [7] gave an algorithm that achieves an approximation ratio of $O(k^{\frac{2}{3}} \log^{\frac{1}{2}} k)$, where k is the number of pairs of vertices that are to be connected for the directed Steiner problem.

Two related problems, both of which are generalizations of the Steiner Tree (ST) problem, are discussed in the literature: Prize-Collecting Steiner Tree (PCST) problem and Node Weighted Steiner Tree (NWST) problem.

The Prize-Collecting Steiner Tree (PCST) problem has been introduced by Bienstock et al. [4] and Goemans et al. [23]. The terminology "prize-collecting" was introduced in Balas [3] for the Travelling Salesman Problem. Then this term became widely used in the description of various combinatorial optimization problems. Prize-collecting is commonly applied when there is a cost to be paid if including a vertex in the solution or a penalty to be incurred if excluding the vertex. There are often other constraints included, which create variants of PCST. For example, the problem in this paper adds a K-cardinality constraint to obtain the K-cardinality Prize-Collecting Steiner Tree problem (KPCST). People have considered other aspects of prize-collecting Steiner trees, like the Prize-Collecting Steiner Forest (PCSF) problem discussed in Hajiaghayi and Jain [26]. PCSF problem takes into account the edge costs, the terminal pairs and the penalties for the terminal pairs. The objective is to find a forest F and a subset Q minimizing the sum of the costs and the penalties. The terminal pairs are either connected by F or contained in Q.

As mentioned earlier, PCST is a generalization of ST problem. ST problem instances with nonnegative edge costs may be remodeled as PCST instances by assigning a sufficiently large positive prize to each terminal and zero prize to each nonterminal. Hence, the optimal PCST solution is guaranteed to contain all the terminals which are required in the ST solution. Therefore, PCST must also be NP-hard since ST with nonnegative edge costs is a special case of PCST.

The Node Weighted Steiner Tree (NWST) problem which permits weights on the vertices and costs on the edges is the same as ST problem but with weights on the vertices. Given a graph with edge costs and vertex weights, the NWST problem finds a subtree containing all terminals and minimizing the sum of the total cost of all edges in the subtree and the total profit of all vertices not contained in the subtree. This is equivalent to PCST, except that, in PCST, there are no mandatory vertices in the tree, i.e., the set of terminal vertices is empty.

Segev [38] first introduced the NWST problem and Duin and Volgenant [18] further analyzed it. Segev considered a special case of NWST, called the *single point weighted Steiner tree* problem, where a vertex has to be included in the solution in addition to the terminal vertices. In Cornone and Trubian [15], they associated a size parameter with each vertex and add a *knapsack* constraint, which creates a problem called Knapsack Node Weighted Steiner Tree (KNWST) problem.

Table 2.1 is a summary of these related problems which reports the functions and constraints they take into account.

Table 2.1: Summary of the related problems

Problem	edge cost	vertex prize	vertex size	root	terminals	solution cardinality
Steiner Tree (ST)	✓				✓	
Node Weighted Steiner Tree (NWST)	✓	✓			✓	
Single point Weighted Steiner Tree	✓	✓		✓	✓	
Knapsack Node Weighted Steiner Tree	✓	✓	✓		✓	
Prize-Collecting Steiner Tree (PCST)	✓	✓				
K-cardinality Prize-Collecting Steiner Tree (KPCST)	✓	✓				✓

The previous work on PCST and NWST focused on three aspects: (1) the generalization of approximation algorithms to approximate the optimum value of the problems within certain constant factor; (2) the development of fast and reasonable heuristic methods to be used for real world problems; and (3) the obtaining of lower bounds and exact algorithms.

Segev [38], who first proposed the NWST problem, developed two Lagrangian relaxation bounding procedures for single point weighted Steiner tree problem. Two mathematics formulations, one of which is called *tree-type formulation* and the other one is called *flow formulation*, were provided. In addition, three heuristic procedures for feasible solutions were developed, two based on greedy approaches and the other based on a subgradient method. These procedures were tested in complete graphs of 40 vertices at most. Finally, single point weighted Steiner tree problem was shown to be an NP-complete problem.

Bienstock et al. [4], who first introduced PCST, proposed a factor 3 approximation algorithm. This algorithm was first developed as a heuristic procedure based on Christofides's heuristic method for the Travelling Salesman Problem (Christofides, [13]). Goemans and Williamson [24] modified a primal-dual method to give an approximation algorithm which runs in $O(|V|^2 \log |V|)$ time and the worst-case performance ratio is $2 - \frac{1}{|v|-1}$. They developed an edge pruning method which can be viewed as a greedy algorithm in the last part of the algorithm to obtain the final solution. Goemans and Williamson algorithm was improved by Cole et al. [14] by reducing the running time from $O(|V|^2 \log |V|)$ to $O(k(|V| + |E|) \log^2 |V|)$. Johnson, Minkoff and Phillips [27] also improved Goemans and Williamson algorithm by giving a new strategy for the pruning phase. The new strategy has been shown to have a better performance based on the results on graphs of up to 25,600 vertices. A 2-approximation algorithm for finding the best subtree over all choices of root was also given without an increase in running time. Feofiloff et al. [21] presented a new algorithm which achieves a ratio of $2 - \frac{2}{|v|}$ within the same time. Then Bateni et al. [2] gave an improved $(2 - \epsilon)$-approximation algorithm.

There are some other approximation algorithms for NWST, e.g., Klein and Ravi [29] obtained a greedy algorithm whose performance ratio is $2 \ln |T|$, where $|T|$ is the number of terminals. The ratio was then improved by Guha and Khuller [25] to $1.603 \ln |T|$ with another simple greedy algorithm.

Some *metaheuristic approaches* for PCST have been proposed (Metaheuristics make few or no assumptions about the problem being optimized and can search very large spaces of candidate solutions). For example, Canuto et al. [6] provided a local search method with perturbations on graphs with up to 1000 vertices and 25,000 edges, most of which were solved to optimality. Klau et al. [28] developed a *memetic* algorithm which includes an exact subroutine for the problem on trees and a *steady-state evolutionary* algorithm to create candidate solutions with less computational time. The memetic algorithm is used to reduce the instance by eliminating edges which are probably not contained in the solution. Then an exact method is applied on the resulting graph.

Besides approximation algorithms and heuristic methods, a wealth of exact algorithms and lower bounds have been presented. Engevall et al. [19] proposed a new formulation of NWST and derived a Lagrangian bound. A Lagrangian heuristic procedure based on Lagrangian relaxation for generating near-optimal solutions was also used. In this new formulation, the authors introduced an artificial root vertex 0 and the edges between vertex 0 and $i \in V \backslash \{1\}$ and restricted that the degree of the root vertex 0 should be 1. An optimal solution is obtained by removing vertex 0 and the edge connected to vertex 0 from the solution tree of the transformed graph. The lower bounds were stronger than those in Segev's algorithm [38].

Lucena and Resende [33] have also proposed a method for obtaining lower bounds. They presented a cutting plane algorithm for the PCST which cuts are generated from subtour elimination constraints. They also used reduction tests given by Duin and Volgenant [18] and 114 instances with up to 1000 vertices and 25,000 edges. Most of the bounds are proved to be optimal. Ljubić et al. [31] later improved Lucena and Resende's results with a branch-and-cut algorithm.

2.2 Applications

In addition to the mining problem, several network design problems can be modeled as PCST. One well-known application is the design of telecommunication access networks (Cunha, Lucena, Maculan and Resende [16]). The problem is to decide whether to create or expand a network offering services to new customers. Every customer could give some profit to the company but there is also a connection cost and labor cost when offering the services to each customer. There exists a natural trade-off between the profit that the new customer could provide and the cost when offering the services to the new customer. Such a problem can be modeled in graph by representing the customers as the vertices and physical links as edges. Laying down the optical fiber and servicing a customer imply a cost, while the profit can be gained from the customers. Sometimes the service provider can be represented as a root which must be included in the solution. In Cordone and Trubian [15], considering the budget limitation or government laws preventing monoplies, capacity constraints are added to the problem. Another similar application is the design of cable television network.

Ljubić et al. [31] have reported a similar application in the planning and expansion of district heating networks. The goal is to find the most profitable subset of customers and cost-efficient way to connect them to the heating plant. One more example, as described in Cordone and Trubian [15], is building connected platforms in an offshore area for an oil company. The edges represent the pipelines, whereas the vertices represent the sites where platforms are installed. The sites generate profits and the costs consist of the installation and the maintenance costs.

PCST can also be used as a subproblem in more general problems. Chawla et al. [8] considered the PCST as part of a mechanism which selects a set of clients to gain services and determine multicasting networks to offer services. Both nodes and edges are considered to be selfish agents. The mechanism provides guarantees to obtain some fraction of the obtainable profit, or demonstrates that no profitable solution exists if market is sufficiently unprofitable. In Duin and Vlogenant [18], the uncapacitied facility location problem is reduced to PCST by assigning positive prizes to customers and assigning negative prizes to the facilities, representing the cost of these facilities.

Chapter 3

Integer Linear Programming Formulations of the KPCST Problem

Here are some notations we use in the remainder of this paper:

- $E(S)$ denotes the edges in the subgraph induced by S.
- For a directed graph with vertex set V, consider a set of vertices $S \subset V$ and its complement $\bar{S} = V \backslash S$. Then cut sets are defined as:

  $$\delta^{-}(S) = \{(i,j) | i \in \bar{S}, j \in S\}$$

  and

  $$\delta^{+}(S) = \{(i,j) | i \in S, j \in \bar{S}\}.$$

- Given a variable x_{ij} for an edge (i,j), $x(S) = \sum_{(i,j) \in S} x_{ij}$ denotes the sum of the x variables of the internal edges in S, $|S| \geq 2$.

3.1 Transformation To the Rooted Directed Graph

We transform the original undirected graph for the KPCST problem to a directed graph in order to achieve a tighter Linear Programming relaxation (LP-relaxation). The fact that LP-relaxation for the directed graph is tighter than that for the undirected graph was proven by Chopra and Rao [12]. Moreover, Chopra, Gorres and Rao [11] demonstrated this fact through some experiments comparing the lower bounds for directed graph and undirected graph.

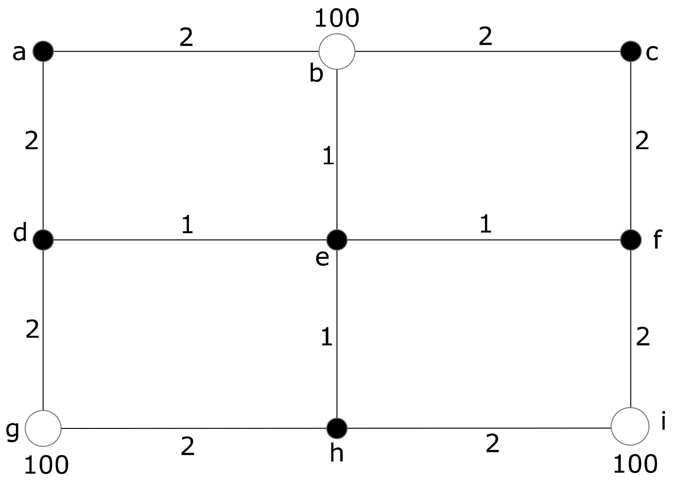
The *transformation to the rooted directed graph* is as follows:

1. Expand the unrooted graph to a rooted graph by introducing an artificial root r.
2. Change a profitable vertex i to two vertices, say a and b: a is a profitable vertex and b is a non-profitable vertex. The new profitable vertex is assigned the same prize as i, i.e., $p_a = p_i$. The prize of the non-profitable vertex is 0, i.e., $p_b = 0$. Connect a and b with an edge both ways, one of which called edge ba goes to a and the other one called edge ab goes to b. Replace the original profitable vertex i by the new non-profitable vertex b. Repeat this procedure until all the profitable vertices have been changed. Let the resulting graph to be $G' = (V', E')$ and let the newly added arc set to be A.
3. Let $G_d = (V_d, E_d)$ be the transformed directed graph. The vertex set $V_d = V' \cup \{r\}$ contains the vertices of the graph G' and an artificial root vertex r. The arc set E_d contains two directed arcs (i, j) and (j, i) for each edge $(i, j) \in E$ together with the arcs from the root r to the profitable vertices $i \in R', R' = \{i \in V_d \mid p_i > 0\}$ and the arc set A.
4. Let c_{ij} be the original cost on the edge $(i, j) \in E$, and let p_i be the prize on the vertex $i \in V_d$. The arc cost vector c' of G_d is defined as

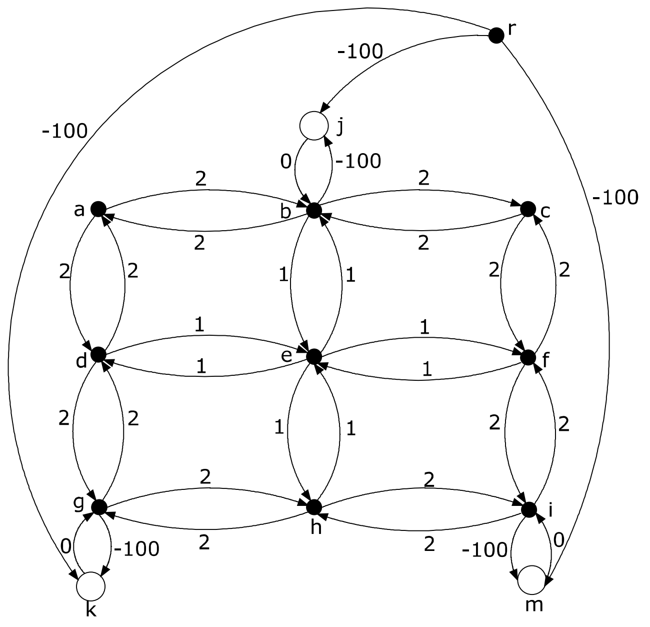
$$c'_{ij} = \begin{cases} c_{ij} - p_j & \forall (i,j) \in E_d, j \in V_d, i \neq r \\ -p_j & \forall (r,j) \in E_d, j \in R' \end{cases}$$

   That is, the arcs from the root to the profitable vertices (r, j) are assigned a cost $-p_j$ while all other arcs (i, j) with $i \neq r$ are assigned a cost $c_{ij} - p_j$. For the arcs in the set A, for example, $ab \in A$, which goes to the non-profitable vertex, is assigned a cost 0; while $ba \in A$, which goes to the profitable vertex, is assigned a cost $-p_a$.
5. Remove all prizes from vertices. This is because they have been encoded in the edge cost according to the definition of the new arc cost vector.

Figure 3.1 is an example to illustrate this transformation.



(a) Original undirected graph



(b) Transformed directed graph

Figure 3.1: An example of the transformation to the rooted directed graph. The numbers on the edges are the costs of these edges. The letters on the vertices are the names of the vertices. The hollow circles are profitable vertices with prize 100, while the solid circles are non-profitable vertices with prize 0. An artificial root r is added in figure (b). Three original profitable vertices, $\{b, g, i\}$, are moved to $\{j, k, m\}$, and $\{b, g, i\}$ are changed to non-profitable vertices. They are connected by the extra arcs $\{bj, jb, gk, kg, im, mi\}$. As can be seen in figure (b), the root is adjacent to the three new profitable vertices $\{j, k, m\}$.

3.2 Integer Programming Formulation For the Steiner Arborescence Problem On the Directed Graph

Once transformed, the problem is to find a min-cost subtree T_d rooted at r that containing exactly K profitable vertices with minimum sum of edge costs; we call this the *Steiner arborescence* problem. This directed tree T_d that all paths in the tree have to be oriented away from the root is called a *Steiner arborescence* (Rao et al. [36]). The optimal solution to the KPCST is identified by removing the artificial root r and all the arcs that are connected to the root. A feasible arborescence is a subgraph which corresponds to a solution of the KPCST with the additional restriction that r must have degree 1 in G_d, to be precise, the root can only be adjacent to one profitable vertex. This ensures that the solution of the original graph is connected after removing the artificial root r and the edges connected with r. Since if the degree of r is more than 1, i.e., r is adjacent to more than one profitable vertex, then the extra profitable vertices adjacent to the root may be isolated in the optimal solution for the original undirected graph, although it is connected in the transformed directed graph. An optimal KPCST can be achieved by a feasible arborescence with minimum total arc cost since it is obvious that minimizing the new arc cost function in the transformed directed graph is equivalent to minimizing the objective function (1.3) in the original undirected graph.

In order to model the Steiner arborescence problem of finding a minimum Steiner arborescence by an integer program, we introduce two variable vectors associated with arcs and vertices: (1) $x \in \{0,1\}^{|E_d|}$ where $x_{ij}, (i,j) \in E_d$ is 1 if and only if (i,j) is included in the solution of the Steiner arborescence problem and 0 otherwise, and (2) $y \in \{0,1\}^{|V_d|-1}$ where $y_i, i \in V_d \backslash \{r\}$ is 1 if and only if i included in the solution, that is:

$$x_{ij} = \begin{cases} 1 & (i,j) \in T_d \\ 0 & \text{otherwise} \end{cases} \quad \forall (i,j) \in E_d \qquad y_i = \begin{cases} 1 & i \in T_d \\ 0 & \text{otherwise} \end{cases} \quad \forall i \in V_d$$

The Integer Programming (IP) formulation for the Steiner arborescence problem on the transformed directed graph is similar with the one used in I. Ljubić et al. [31], with the difference that they did not consider the cardinality of the set of profitable vertices in the solution. Thus, one more constraint for the cardinality limit is added for this specific problem. This IP formulation is as follows:

(IP1)

$$\min \sum_{ij \in E_d} c'_{ij} x_{ij} \tag{3.1}$$

subject to

$$\sum_{ji \in E_d} x_{ji} = y_i \qquad \forall i \in V_d \backslash \{r\} \tag{3.2}$$

$$\sum_{ri \in E_d} x_{ri} = 1 \tag{3.3}$$

$$x(\delta^-(S)) \geq y_i \quad i \in S, \forall S \subset V_d \text{ such that } r \notin S, \tag{3.4}$$

$$\sum_{p_i > 0} y_i = K \tag{3.5}$$

$$y_r = 1 \tag{3.6}$$

$$x_{ij}, y_i \in \{0,1\} \qquad \forall (i,j) \in E_d, \forall i \in V_d \backslash \{r\} \tag{3.7}$$

We name this IP formulation as "IP1". In order to ensure that the objective function (3.1) and (1.3) have the same value, the constant term $\sum_{i \in V_d} p_i$ is added above.

Constraint (3.2), which is called *in-degree* constraint, guarantees that each vertex selected in the solution must have exactly one predecessor on its path from the root. The so-called *root-degree* constraint (3.3) makes sure that the artificial root r has out-degree 1, thus it is only adjacent to a single profitable vertex, which is crucial for the connectedness of the solution to KPCST according to our analysis above.

Constraints (3.4) are called *connectivity* inequalities for this problem. They guarantee that each selected vertex i must be connected in the solution, i.e., there must be a directed path from the root r to i. The justification for this is as follows: Let D be the digraph induced by the edges with non-zero x values. let set $S' = \{$all vertices in D to which there is no path from the root$\}$. If D is disconnected, then S' is nonempty. However, this would contradict the constraints (3.4). Therefore, in the solution, D contains a path from the root r to $i, \forall i \in T_d$.

Constraint (3.5) requires the cardinality of the set of profitable vertices in the solution of IP1 problem to be K. Constraint (3.6) guarantees that the root is in the solution.

From constraint (3.2) and (3.4), it is easy to see that the feasible solution of IP1 problem can be viewed as a set of paths from the root vertex to the other vertices and there is exactly one directed path from the root to each vertices. Thus the feasible solution of the IP1 problem is a rooted directed tree called Steiner arborescence as mentioned before. The optimal solution of IP1 problem is a feasible arborescence with minimum total arc cost.

3.3 Strengthening the Formulation

As mentioned before, the feasible solution of IP1 problem is already a Steiner arborescence. But in order to reduce the computational time, we add extra constraints. These constraints are also used in Ljubić et al. [31].

3.3.1 Out-degree Constraint

The feasible solution of the Steiner arborescence problem is a set of paths from the root to the selected vertices. In view of the structure of this solution, we notice that a non-profitable vertex could be a *leaf* of the solution tree only if it is connected through a zero cost edge (a leaf in a tree means a vertex with the out-degree to be 0). But this case is not necessary for our solution, thus we use the following constraint to prevent a non-profitable vertex from being a leaf.

$$y_i \le \sum_{ij \in E_d} x_{ij}, (\forall i \notin R, i \ne r) \tag{3.8}$$

Moreover, from the constraint (3.2), $\sum_{ji \in E_d} x_{ji} = y_i$, constraint (3.8) can be written as:

$$\sum_{ji \in E_d} x_{ji} \le \sum_{ij \in E_d} x_{ij}, (\forall i \notin R, i \ne r) \tag{3.9}$$

Constraint (3.9) expresses a trivial fact that the in-degree is always smaller than or equal to the out-degree for a non-profitable vertex whose out-degree is at least 1, because the in-degrees of the vertices except the root in the solution tree of the Steiner arborescence problem are always 1.

Constraint (3.9) is referred as the *flow-balance* constraint in some literature. It was introduced by Koch and Martin in [30] for the Steiner Tree problem.

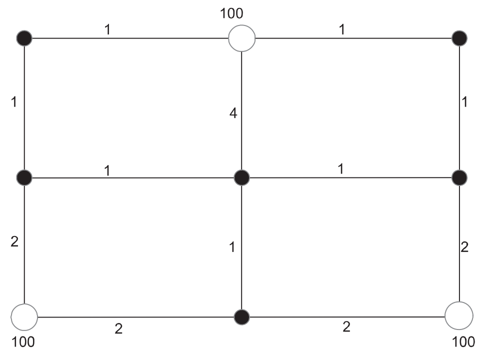
Figure 3.2 is an example to show that constraint (3.9) indeed strengthens the LP-relaxation of (3.2)–(3.7) .

3.3.2 One-way Constraint

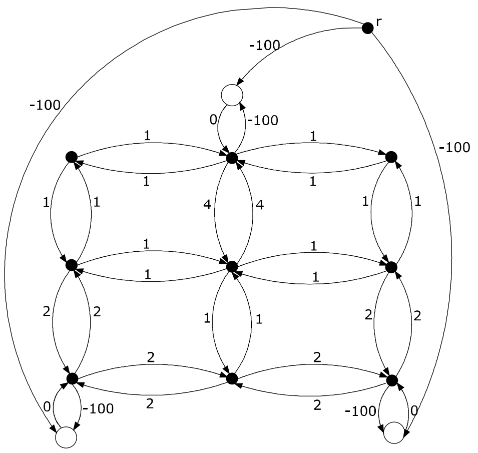
The structure of the solution of the Steiner arborescence problem, in which the paths are from the root to the selected profitable vertices, implies that every edge can be only oriented in one way. The constraint (3.10) expresses this fact.

$$x_{ij} + x_{ji} \le y_i, \forall i \in V_d \backslash \{r\}, (i, j) \in E_d \tag{3.10}$$

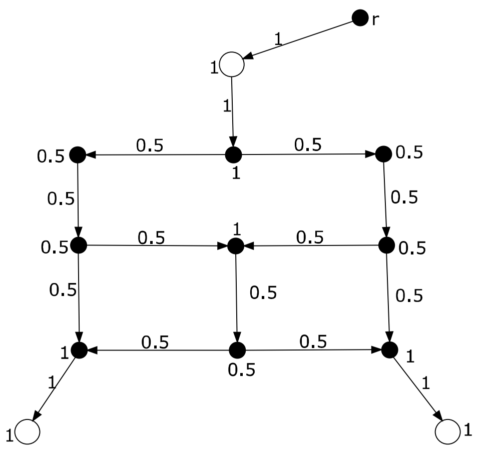
Ljubić et al. [31] compared the computational results with and without constraint (3.10) and found that this constraint was crucial for their test instances to reduce the time. Some of their test instances could not be solved without this constraint. We do not have enough time to do this test but we may do this test in the future.



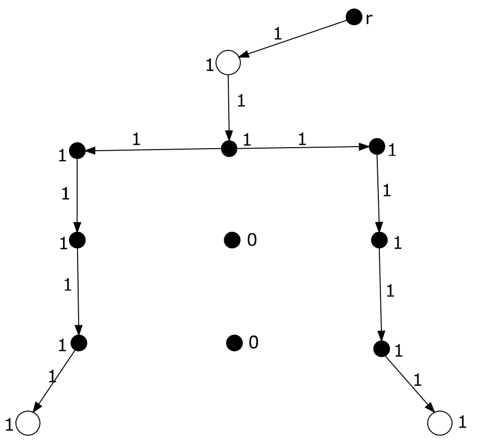
(a) Original undirected graph



(b) Transformed directed graph



(c) Solution of the LP-relaxation of (3.2)-(3.7) without out-degree constraint (3.8) or (3.9). The total cost of the arcs in the square matrix is 7.5.



(d) Solution of the LP-relaxation of (3.2)-(3.7) with out-degree constraint (3.8) or (3.9). The total cost of the arcs in the square matrix is 8.

Figure 3.2: An example showing a strengthening of the LP-relaxation of (3.2)-(3.7) by adding out-degree constraint. The hollow circles are profitable vertices with prize 100, while the solid circles are non-profitable vertices with prize 0. The numbers on the edges are the solution values of y variables and the numbers on the vertices are the solution values of x variables.

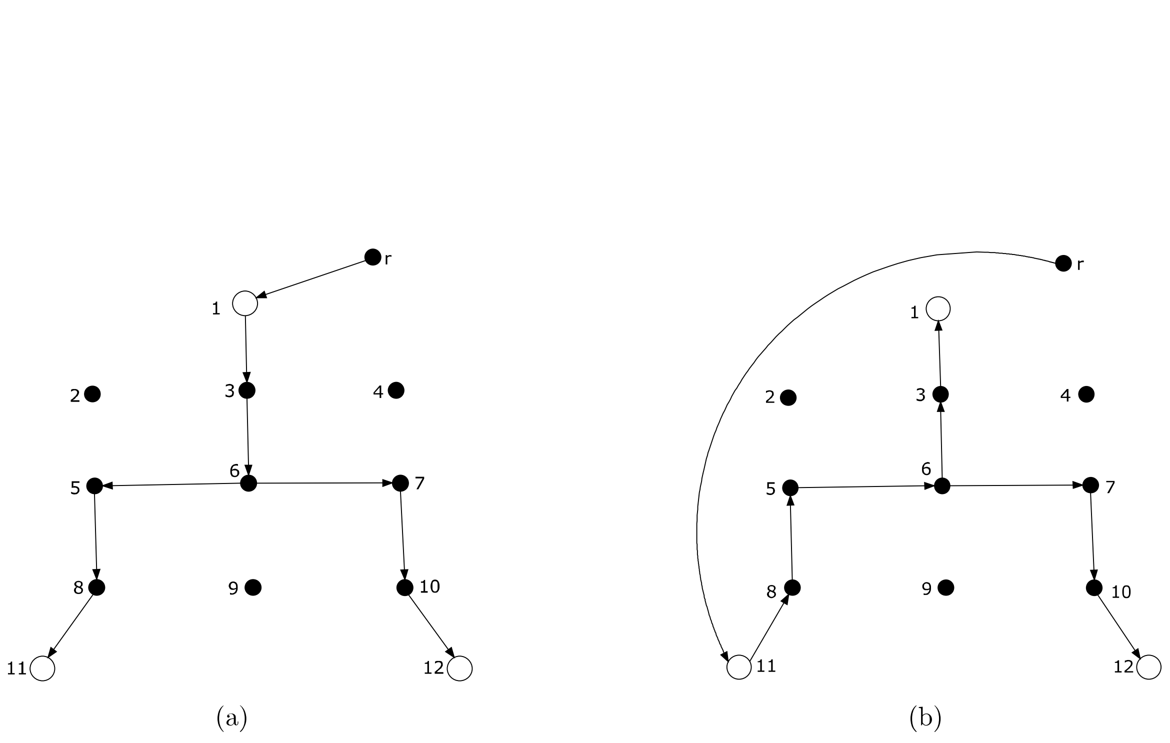


Figure 3.3: An example of two feasible solution of Steiner arborescence problem representing the same KPCST solution. $K = 3$.

3.3.3 Asymmetry Constraints

Considering the solution of the Steiner arborescence problem and the KPCST problem, there exists some cases that the different feasible solutions of Steiner arborescence problem represent the same KPCST solution. Figure 3.3 illustrates an example.

In order to exclude these unnecessary solutions of the Steiner arborescence problem to reduce the computational time, we add constraints called *asymmetry* constraints.

$$x_{rj} \leq 1 - y_i \quad \forall i < j, i \in R \tag{3.11}$$

These constraints require that, for every feasible solution of the Steiner arborescence problem, the root is adjacent to the profitable vertex with the smallest index.

3.4 Variants of the Mining Problem

The planning of the mining problem is a very complex and large problem and the KPCST problem is a vast simplification of it. More cases under different conditions need to be considered in practice. As discussed in Chapter 1, the rooted version of the KPCST problem which contains a root representing the access point to which the network joining the stopes is worthy being considered.

We first transform the rooted version of the KPCST problem to the Steiner arborescence problem on a directed graph. The transformation is the same as it we discussed in Section 3.1. Then we adapt the IP1 formulation discussed in Section 3.2 to the rooted version.

In order to satisfy the requirement that the root r must be contained and connected in the solution, we simply add a so-called *essential-vertex* constraint:

$$y_r = 1 \tag{3.12}$$

An alternate method is to assign the root a large enough prize to ensure the root to be selected in the solution.

There are some other cases of the planning of the mining problem which we need to consider. For example, we may plan to augment a network by a set of new stopes. The existing network can be shrunken into a single stope which must be included in the solution of the mining problem. Thus we model the problem by essential vertex KPCST: these shrunken vertices must be contained and connected in the solution. Apparently, this problem is almost the same as the rooted version of the KPCST problem. The formulation can be modified by adding essential-vertex constraints. Let $D = \{i \in V_d \mid \text{ i is an essential vertex}\}$

$$y_i = 1, \quad i \in D \tag{3.13}$$

Then, notice that the selected profitable vertices should not be in the set of D, so that the constraint (3.5) is modified to:

$$\sum_{i \in V_d} y_i = K, i \in R, R = \{i \in V_d | p_i > 0\}, i \notin D, \tag{3.14}$$

Of course, an alternate method is to assign these essential vertices large enough prizes to ensure these essential vertices to be selected in the solution.

Chapter 4

Our Algorithms

4.1 Overview of Our Algorithms

In this chapter we give a general overview of the algorithm to solve IP1 problem. The next section will give the details of some steps.

In general, our algorithm contains 4 steps: heuristic method, initialization, branch-and-cut and separation. This algorithm is mainly based on the one in Ljubić et al. [31] and some necessary adaptations are made for IP1 problem. For example, we do not use any preprocessing steps in our algorithm while Ljubić et al. [31] applied such steps to reduce the computational time. Instead, we describe a heuristic method to try to reduce the computational time. Moreover, Ljubić et al. [31] used the branch-and-cut algorithm once and applied the separation algorithm at each node of the branch-and-cut tree; while we run the separation algorithm after every branch-and-cut step finishes and may resolve the IP problem using the branch-and-cut method again, i.e., we use the branch-and-cut algorithm several times in order to implement our algorithm more efficiently (we did not find the way to incorporate new constraints during the branch-and-cut procedure due to limited time). The flowchart of the general algorithm is given in figure 4.1.

Step 1. *Heuristic method*

We describe a heuristic method for the Steiner arborescence problem. It is adapted from the method in Philpott and Wormald [35] with some modifications according to the problem we are concerned with here. The heuristic method in Philpott and Wormald [35] solved the problem in graph theory arising from a model for the planning of the extraction of ore from an open cast mine. They modeled the mining network as a mining graph which is a directed graph having a root, together with a weight function associated with vertices of the graph. They used *growtree* and *findtree* to find a heaviest subtree which contains

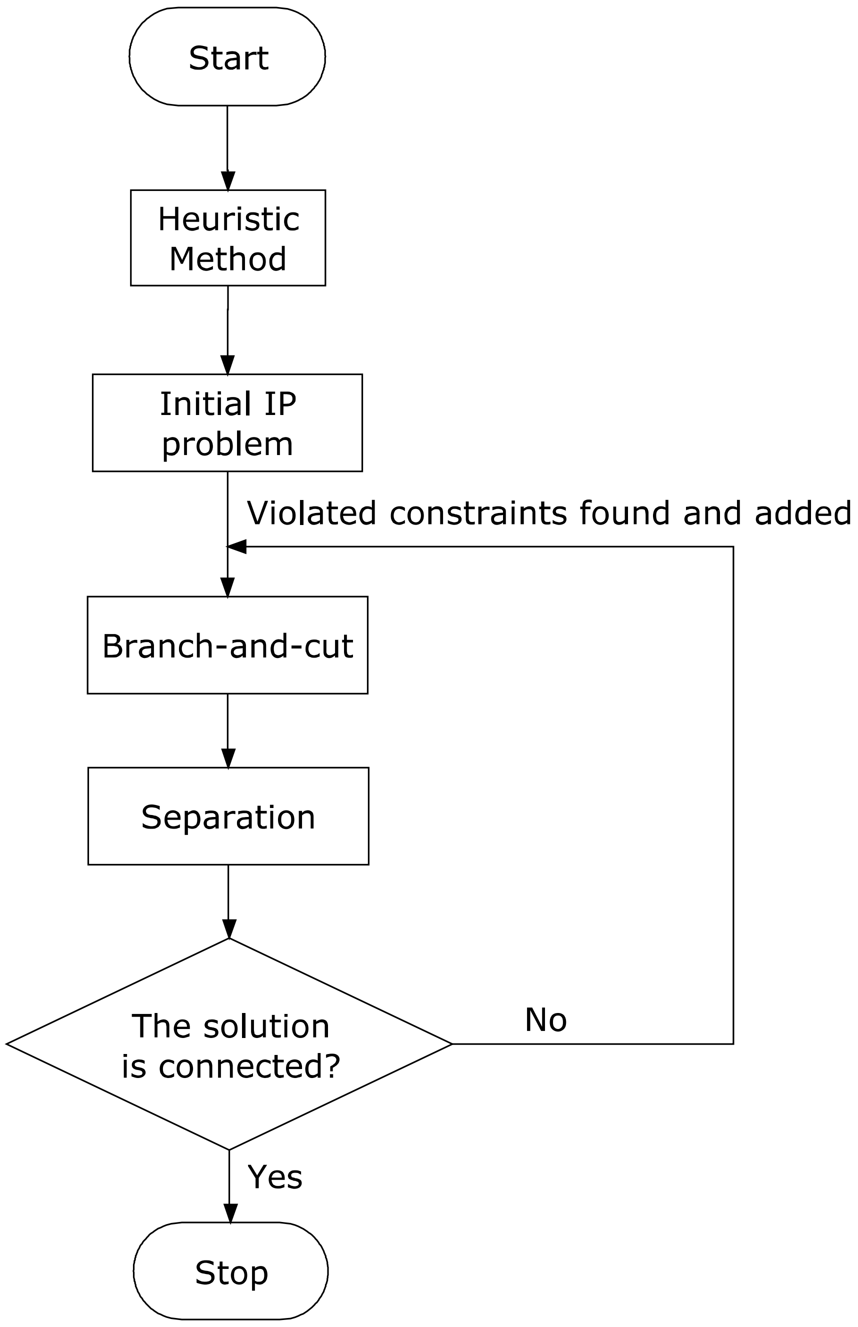


Figure 4.1: Flowchart of the algorithm used

a certain number of vertices and the root in the mining graph. Therefore, the problem considered by Philpott and Wormald [35] just has the vertex weight function, while the Steiner arborescence problem considered here has both the edge cost function and the vertex weight function. In addition, We took into account both the profitable vertices and the non-profitable vertices, while Philpott and Wormald [35] only considered the profitable vertices. Thus we modified the two main algorithms, growtree and findtree, to solve our problem. The details will be discussed in the next section. After we obtain the solution, set h to be equal to the value of $\sum_{v \notin V_T} p_v + \sum_{e \in E_T} c_e$ found.

Step 2. *Initialization of IP problem*

We initialize the IP problem which is called "IP2" problem in a directed graph to be as follows:

(IP2)

$$\min \sum_{ij \in E_d} c'_{ij} x_{ij} + \sum_{i \in V_d} p_i$$

subject to

$$\begin{aligned}
\sum_{ji \in E_d} x_{ji} &= y_i && \forall i \in V_d \backslash \{r\} \\
\sum_{ri \in E_d} x_{ri} &= 1 \\
\sum_{p_i > 0} y_i &= K \\
x_{rj} &\le 1 - y_i && \forall i < j, i \in R \\
y_r &= 1 \\
x_{ij}, y_i &\in \{0,1\} && \forall (i,j) \in E_d, \forall i \in V_d \backslash \{r\}
\end{aligned}$$

The connectivity constraints (3.4): $x(\delta^-(S)) \ge y_k$ are not inserted at the beginning since there are exponentially many of them. Without this connectivity constraint, the solution of IP2 problem may be unconnected. Thus we need to add enough constraints to ensure the solution to the final IP problem is valid, which in this case means that it is connected. These can be done in the *separation* step which will be discussed later. We do not have to add all of the connectivity constraints because once the solution of the final IP problem is connected, those added constraints have done their job.

When initializing, we also added the asymmetry constraint (3.11). In fact, we could have added more strengthening constraints as described before, see (3.8) and (3.10), in this initial IP problem. Adding and testing these constraints could possible be included in future work on this problem.

For the rooted or the essential vertex version of the KPCST problem, we can solve them by adding the essential-vertex constraint for the root or the essential vertices in the initial IP problem and then proceeding as with the unrooted version.

Step 3. *Branch-and-cut*

We use a branch-and-cut algorithm to solve IP2 problem in order to get an optimal integral solution. Branch-and-cut is a method of combinatorial optimization for solving IP problems, which is a hybrid of *branch-and-bound* and *cutting plane* methods.

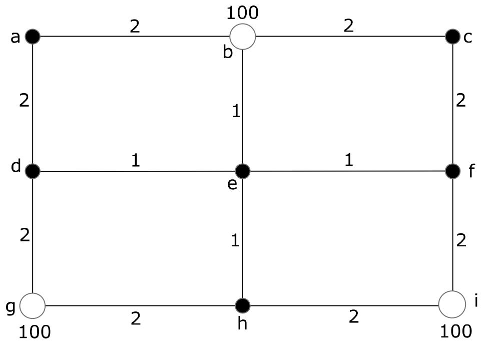
Branch-and-cut method first solves the linear program which is called the LP-relaxation of IP problem without the integer constraints using the regular simplex algorithm to get an optimal solution. If this solution is not integral, that is, any of the variables in the solution is not integer, the cutting plane algorithm is applied to find further linear constraints which are violated by the current fractional solution but satisfied by all feasible integer points. If such a constraint exists, it is added to the linear program. Then the new LP problem with the added constraint is resolved and a different solution which is hopefully "less fractional" is obtained. This process is repeated until no more cutting planes can be found or until an integer solution is found. This integer solution then can be viewed as optimal solution. In our algorithm, the cutting plane part is automatically implemented by CPLEX which is discussed in Chapter 5.

Then the branch-and-bound part of the algorithm is started. The problem is split into two subproblems, one with the additional constraint that the variable is larger than or equal to the next integer greater than the current fractional value, and one where this variable is less than or equal to the next lesser integer. The new linear subprograms are then solved using the simplex method and the process repeats until an optimal solution is found or the problem is proved to be infeasible. During the branch and bound process, further cutting planes can be applied. These processes will be described in detail in Section 4.2.

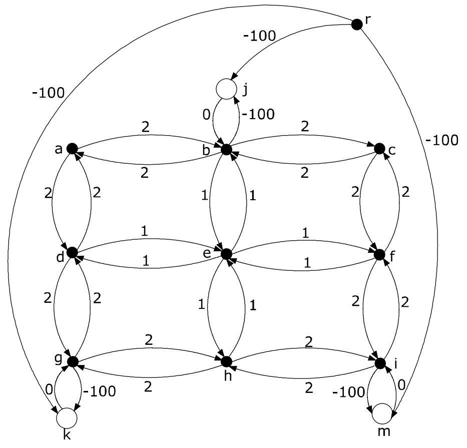
Step 4. *Separation and Termination*

After we obtain an integral solution of IP2 problem in Step 3 using the branch-and-cut algorithm, we need to check the connectivity of this solution. If the solution is not connected, then the solution must violate some of the connectivity constraints (3.4). The corresponding constraints are called *violated constraints*. On the other hand, if there exists violated constraints, the solution is not connected. Therefore, during the separation phase, we first check the connectivity of the integral solution with the connectivity constraints (3.4). If there exists a violated constraint of type (3.4), some such violated constraints are added dynamically into IP2 problem. Then go to step 3; otherwise, we find the integral connected solution, i.e., the optimal feasible solution, and stop.

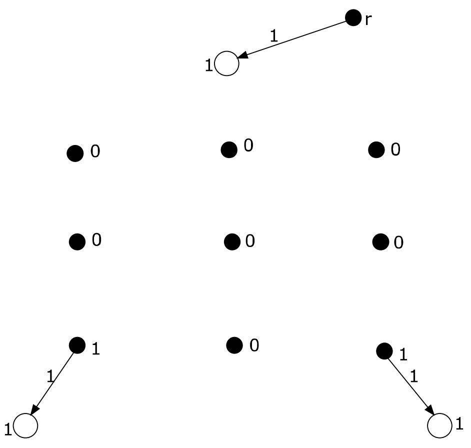
Figure 4.2 and 4.3 give an example of an application of the algorithm. First the original undirected graph is transformed to a directed graph. Then the branch-and-cut method is applied to solve the initial IP2 problem. After that, the separation method is used to check the connectivity of the solution and the violated constraints are added to IP2 problem. This process is repeated until the optimal feasible solution is found. In the example, three times of separation method are used and the problem is resolved by branch-and-cut method four times.



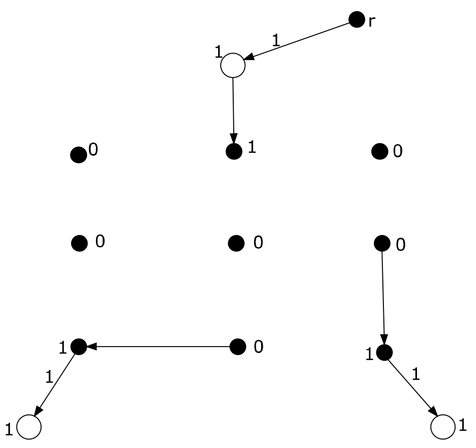
(a) Input undirected graph.



(b) transformed directed graph after transformation to Steiner arborescence problem.

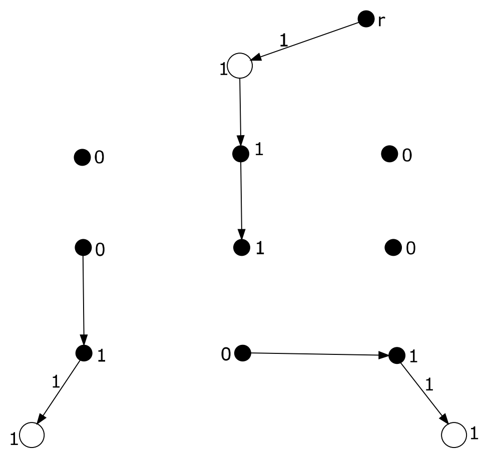


(c) Integral solution of the IP2 problem with branch-and-cut. Values of x and y variables are shown. Apparently, the solution is not connected, so the violated constraints added to IP2 problem.

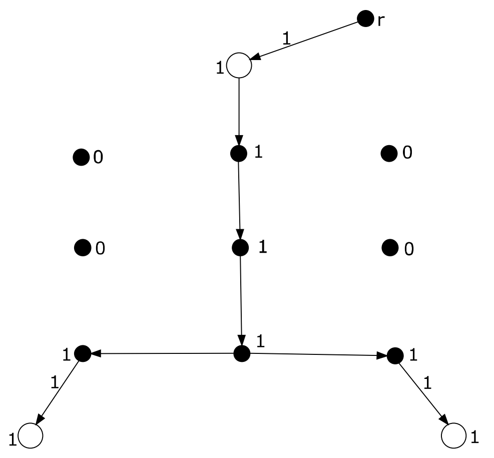


(d) Integral solution of the IP2 problem with branch-and-cut after the first connectivity check in the separation phase. The solution is not connected. The violated constraints added to IP2 proble.

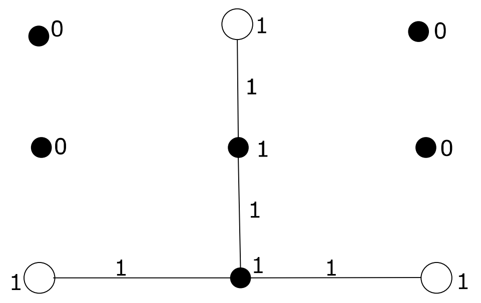
Figure 4.2: An example of the application of the general algorithm. The hollow circles are profitable vertices with prize 100, while the solid circles are non-profitable vertices with prize 0.



(a) Integral solution of the IP2 problem with branch-and-cut after the second connectivity check in the separation phase. The solution is not connected. The violated constraints added to IP2 proble.



(b) Integral solution of the IP2 problem with branch-and-cut after the third connectivity check in the separation phase. The solution which is integral and connected is the optimal feasible solution.



(c) The corresponding optimal solution in the original undirected graph.

Figure 4.3: An example of the application of the general algorithm. The hollow circles are profitable vertices with prize 100, while the solid circles are non-profitable vertices with prize 0.

4.2 Algorithm Details and Explanations

4.2.1 Our Heuristic Method

In this section, we present a modified heuristic method based on the one discussed in Philpott and Wormald [35]. Two main algorithms: *growtree* and *findtree*, are used in this method to find a rooted directed subtree which contains exactly K profitable vertices minimizing the sum of the edge costs for the Steiner arborescence problem on the transformed directed graph by the transformation described in Section 3.1. Recall that there is no prize associated with the vertices in this transformed directed graph for the Steiner arborescence problem and the edge costs have both positive and negative number, (see, Figure 3.1b in Chapter 3). For convenience, we change the negative edge costs of the transformed directed graph to be 0 and assign the profitable vertices their original prizes (see, Figure 4.4a). Hence, the problem is changed to find a rooted directed subtree which contains exactly K profitable vertices minimizing the sum of the edge costs minus the sum of the profitable vertex prizes on this modified directed graph using the heuristic method.

Growtree algorithm

As we discussed above, the input graph for the heuristic method is the modified directed graph $G'_d = (V_d, E_d)$. Then we restrict the out-degree of the root to be 1 by connecting the root to the profitable vertex with the largest prize. *Growtree* is used to construct a feasible arborescence $T = (V_T, E_T)$ where $R \subset V_T$, that is, a rooted directed subtree containing all the profitable vertices in which the out-degree of the root is 1. In our searching process of the growtree algorithm, two kinds of vertices, profitable vertices and non-profitable vertices, are considered; while only one kind of vertices in the findtree algorithm is described in Philpott and Wormald [35]. Another difference is that both the edge costs and the vertex prizes are considered here, while only the vertex weights are considered in Philpott and Wormald [35]. The description of the growtree algorithm is as follows.

Algorithm 1 Growtree algorithm

```
Require: A directed graph $G'_d = (V_d, E_d)$ with the costs associated with the edges and
  the prizes associated with the profitable vertices.
Ensure: A directed subtree $T = (V_T, E_T)$ which contains all the profitable vertices and
  the root.
  Restrict the out-degree of the root to be 1 by connecting the root to the profitable vertex
  with the largest prize, say $j$.
  set $T = \{r\} \cup \{j\} \cup \{rj\}$
  repeat
    Find a profitable vertex $i^*$ such that the prize $p_{i^*} = max\{p_i | i \in R$ and $i \notin T\}$. If there
    are more than one profitable vertices which have the same largest prize, we choose one
    of them randomly.
    Find the least cost path from $T$ to $i^*$, i.e., consider all paths from any node in $T$ to $i^*$
    and choose the one with the least cost of them and call the path $P$. Similarly, if there
    are more than one paths to $i$ which have the same least cost, we choose one of them
    randomly.
    {Comment: Since the edge costs are nonnegative, we use Dijkstra's algorithm, which
    solves the single-source shortest path problem for a graph with nonnegative edge
    path costs. Here, Dijkstra's algorithm is stopped once the shortest path has been
    determined.}
    $T = T \cup \{P\}$
  until all the profitable vertices are in $T$
```

Findtree algorithm

Findtree is a recursive algorithm based on dynamic programming, which can be applied to find exactly K profitable vertices together with the root r in T to satisfy the minimization requirement. Roughly speaking, *growtree* set up the skeleton framework for mining all the profitable stopes, while *findtree* chooses the best K stopes to extract.

The arcs of the directed tree obtained from *growtree* are oriented from the root r. Before we give the description of this algorithm, it is necessary to introduce some definitions which follow Philpott and Wormald [35] that we will need in findtree:

1. *son* of a vertex u: u is a vertex of T, a *son* of u is a vertex adjacent from u. If u is a leaf then v is null.
2. $B(u) = \{$the branch of G_d which consists of u together with all its descendants$\}$.
3. Order the sons of a vertex u. $S(u, v) = \{$the set of sons of u which come after v based on the order of the sons of u $\}$. If u is a leaf then $B(u)$ is just u and $S(u, v)$ is $\emptyset$.

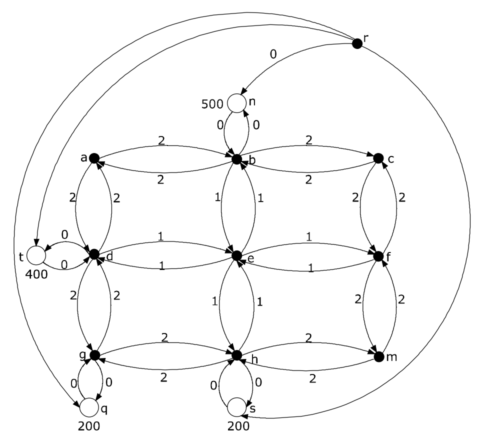
4. $B(u, v) = \{$the subtree with minimal sum of edge cost containing no vertices in $S(u, v)$ $\}$.

The main function of the findtree algorithm is findtree(u, v, f) where u is a vertex of T, v is a son of u, (If u is a leaf then v is null.) and j is a positive integer at most K. The output of this function is a subgtree of $B(u, v)$ which contains f profitable vertices and the vertex u if u is a profitable vertex minimizing the sum of the total edge cost in the subtree. If we set u to be the root r, v to be the last son of r and j to equal to K, then findtree(r, v, k) will find the subtree of T with the minimum sum of the total edge cost, which contains K profitable vertices and the root r. The differences between the findtree algorithm here and that presented in Philpott and Wormald [35] are the same as the differences of the growtree algorithm we described in Section 4.2.1.

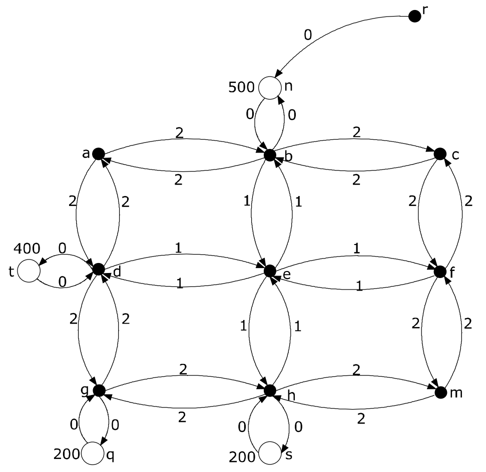
The description of the recursive algorithm findtree is as follows. An example of the heuristic method including growtree and findtree algorithm is given in figure 4.4 and 4.5.

Algorithm 2 Findtree algorithm

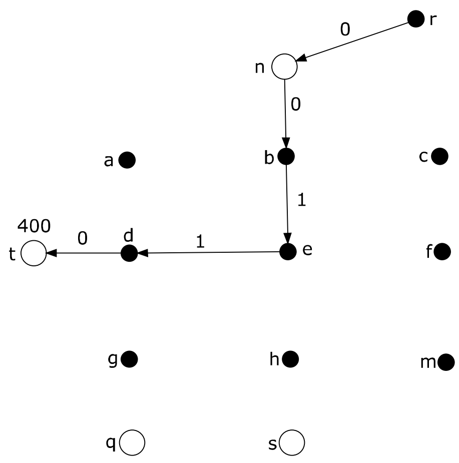
```
Require: findtree($u, v, f$), $u$ is a vertex of $T$, $v$ is a son of $u$, (If $u$ is a leaf then $v$ is null.)
    and $j$ is a positive integer at most $K$
Ensure: subtree of $B(u, v)$ which contains $f$ profitable vertices with the minimum sum of
    the total edge cost
  if u is null or j=0 then
    return ∅
  else if u is a leaf and j > 0 then
    if j=1 && u is a profitable vertex then
      return {u}
    else
      return null
    end if
  else
    $T_{best} = T$;
    $w$=the last son of $v$;
    if $v$ is the first son of $u$ based on the order of the sons of $u$ then
      if $u$ is a profitable vertex then
        $T = findtree(v, w, j - 1) \cup \{u\}$
      else
        $T = findtree(v, w, j) \cup \{u\}$
      end if
      if $T$ is not null and the sum of the total edge cost minus the sum of the total prize
      of $T$ is less than this value of $T_{best}$ then
        $T_{best} = T$;
      end if
    else
      $v'$=the son of $u$ immediately before $v$ based on the order of the sons of $u$
      for all i=1 to j do
        $T = findtree(u, v', i) \cup findtree(v, w, j - i)$
        if either of the two trees in the above union is null then
          $T$ = null;
        else
          if the sum of the total edge cost minus the sum of the total prize of $T$ is less
          than this value of $T_{best}$ then
            $T_{best} = T$;
          end if
        end if
      end for
    end if
  end if
  return $T_{best}$;
```



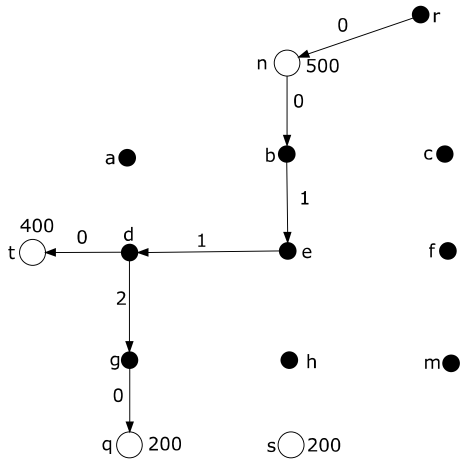
(a) The input graph



(b) Restrict the out-degree of the root to be 1 by connecting the root to the profitable vertex n which has the largest prize. $T = \{r\} \cup \{n\} \cup \{rn\}$.

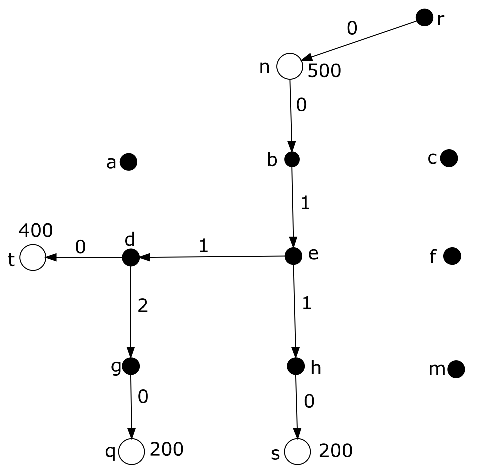


(c) Growtree algorithm. Add the least cost path $P1$ from T to the current most profitable vertex t. $T = T \cup P1$ is shown.

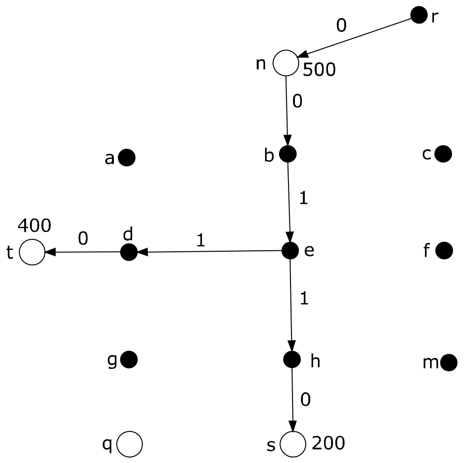


(d) Growtree algorithm. Add the least cost path $P2$ from T to the current most profitable vertex q which is chosen randomly between p and s since $p_q = p_s$. $T = T \cup P2$ is shown.

Figure 4.4: An example of the heuristic algorithm. The hollow circles are profitable vertices, while the solid circles are non-profitable vertices with prize 0. The numbers on the profitable vertices are the prizes and the numbers on the edges are the costs. The letters on the vertices are the names of the vertices.



(a) Growtree algorithm. Add the least cost path $P3$ from T to the current most profitable vertex s. The output of growtree algorithm $T = T \cup P3$ is shown.



(b) Findtree algorithm. Select K profitable vertices minimizing the sum of the total edge costs minus the sum of the total vertex prizes. $K = 3$.

Figure 4.5: An example of the heuristic algorithm. The hollow circles are profitable vertices, while the solid circles are non-profitable vertices with prize 0. The numbers on the profitable vertices are the prizes and The numbers on the edges are the costs. The letters on the vertices are the names of the vertices.

4.2.2 Branch-and-cut Algorithm

A branch-and-cut algorithm is used to solve the problem IP2. Denote the optimal integral solution by x^* and y^* with an minimum cost value $\bar{z}$. The classical branch-and-cut algorithm (see, Mitchell [34]) for an IP minimization problem is as follows:

Step 1. *Initialization*

Denote the initial IP2 problem by IP^0 which is at the root node of the branch-and-cut tree where each node is an IP problem. Define the set of *active* nodes to be $L = \{IP^0\}$ (Here *active* node means that the problem at the node is not processed or pruned). Set the upper bound $\bar{z} = h$ (h is the objective function value obtained from the heuristic method) and the lower bound $z_l = -\infty$. The upper bound $\bar{z}$ is the best value of the objective function found so far. The incumbent objective value $\bar{z}$ will be replaced by a better value if found.

Step 2. *Termination*

If $L = \emptyset$, then the integral solution x^* and y^* that yield the incumbent objective value $\bar{z}$

are optimal. If $\bar{z} = h$, this means after we process all the IP problems in the branch-and-cut tree, we cannot find a better integral solution than the heuristic method.

Step 3. *Node selection*

Select a node $IP^l, l \in L$ in the branch-and-cut tree and delete this current node from the active node set. This node will now be processed.

Step 4. *Relaxation*

Solve the LP-relaxation of $IP^l, l \in L$, obtained by replacing the integrality requirements $x_{ij} \in \{0,1\}, \forall(i,j) \in E_d$ and $y_i \in \{0,1\}, \forall i \in V_d \backslash \{r\}$ by $0 \leq x_{ij} \leq 1, \forall(i,j) \in E_d$ and $0 \leq y_i \leq 1, \forall i \in V_d \backslash \{r\}$. If the optimal objective value of the relaxation is finite, we set z_l to be this objective value and the solution, x^l and y^l, to be the optimal solution for the LP-relaxation of problem IP^l; if the relaxation is infeasible, we set $z_l = +\infty$ and go to Step 5; otherwise, if the optimal solution is negative infinite, we set z_l to be $-\infty$.

Step 5. *Fathoming and pruning*

After obtaining the lower bound z_l for the LP-relaxation of IP^l, we fathom and prune the current node in the branch-and-cut tree by one of the following criteria.

(a) If $z_l \geq \bar{z}$, prune this node, that is, remove this node from the branch-and-cut tree and go to Step 2.

(b) If $z_l < \bar{z}$, and the solution vectors x^l and y^l are integral, update $\bar{z} = z_l$, set this node to be new incumbent node, and delete all problems l' from L with $z_{l'} \geq \bar{z}$, remove the corresponding nodes in the branch-and-cut tree, and go to Step 2.

(c) If $z_l < \bar{z}$ and the solution vectors are not all integral, go to Step 6 to branch at this node.

Step 6. *Partitioning*

Split the node problem IP^l into two smaller subproblems, typically by branching on a variable that violates its integrality constraint, and then go to Step 2. For example, we branch the problem by fixing a fractional variable x_{ij} or y_i to be 0 and 1. These subproblems are added to the branch-and-cut tree as active nodes.

Our strategy of the branching process for IP2 problem is to branch using the vertex variables y first, and then branch using the edge variables x after no more fractional y variables can be found. There are some other strategies such as to branch using the edge variables first and then the vertex variables. In our implementation, we only use the first strategy because we do not have time to test other strategies. This is a possible item for future work.

Here are the details of our first strategy. Given the set of fractional vertex variables $S_1 = \{y_i \mid y_i \text{ is fractional }\}$ and the set of fractional edge variables $S_2 = \{x_{ij} \mid x_{ij} \text{ is fractional}\}$, let

$$y'_i = \begin{cases} y_i & y_i \leq 0.5 \\ 1 - y_i & otherwise \end{cases} \quad \forall y_i \in S_1$$

We select the vertex variable $y^*_i \in S_1$ where $y'^*_i = max\{y'_i\}$ to branch. This selection method is discussed in the handbook of CPLEX and used in some sample codes. Denote the subproblems by $\{IP^{l_1}\}$ where y^*_i is fixed to be 1, and $\{IP^{l_0}\}$ where y^*_i is fixed to be 0. Add the two subproblems to L. If S_1 is $\emptyset$, i.e., no more vertex variables are fractional, then select the edge variable $x_{ij} \in S_2$ to branch through the same way with the vertex variables. Set the lower bound for the subproblems to be z_l which is the same as the one for the parent problem IP^l. Go to Step 2.

4.2.3 Separation Algorithm

As described in Section 4.1, the separation algorithm is to check the connectivity of the solution of the IP2 problem obtained by the branch-and-cut algorithm, find some violated constraints and add them into the formulation of IP2 problem. We keep adding these violated constraints till a connected solution is found. This algorithm was also used in Ljubić et al. [31].

Recall that the connectivity constraint (3.4) is $x(\delta^-(S)) \geq y_i, i \in S, r \notin S, \forall S \subset V_d$ which requires that the sum of the values of the variables of the edges in the cut set of S is larger than or equal to the value of the variable y_i. As proved in Section 3.2, this constraint guarantees the connectivity of r and i. Since S is arbitrary, this constraint must be satisfied by the sum of the values of the variables of the edges in any cut set between r and i if r and i are connected. Define the *support network* to be the rooted directed graph we obtained by the transformation presented in Section 3.1. Set the arc capacities of this network to be the solution obtained with the branch-and-cut algorithm. The connectivity constraint then requires that the sum of the capacity of each edge in any cut set between r and i should be greater than or equal to the value of y_i. Denote the *capacity of a cut* to be the sum of the capacity of each edge in the cut set. Therefore, the capacity of the minimum cut between r and i which has the minimum capacity must satisfy this constraint. Moreover, if r and i are not connected, the capacity of the minimum cut between them must violate this constraint. Thus we need to find the minimum cut between r and i to check the connectivity of r and i. Once we find a minimum cut whose capacity does not satisfy the connectivity constraint (3.4), we call this cut a *violated cut* and add the corresponding violated constraint into IP2.

We use the maximum flow algorithm on the support network to find the minimum cut according to the Max-flow min-cut theorem which states that the maximum flow value is equal to the minimum capacity of the cuts between the sink and the source. In particular, the push-relabel algorithm (Cherkassky and Goldberg [9]) is applied. Only the minimum cut for all pairs of vertices (r, i), with $i \in R, y_i > 0$ are concerned, since we only care about the connectivity between the profitable vertices and the root. In every search, the root r is the source and the profitable vertex i is the sink.

Denote the maximum flow algorithm to be $(f, S_r, S_i) = \text{MaxFlow}(G_d, x, r, i)$ where G_d is the input directed graph and x is the edge-variable solution vector which is viewed as a capacity vector. This function returns the maximum flow value f and two sets of vertices:

- a subset $S_r \subset V_d$ containing root vertex r and the vertices on the source side with $x(\delta^+(S_r)) = f$.
- a subset $S_i \subset V_d$ containing vertex i and the vertices on the sink side with $x(\delta^-(S_i)) = f$.

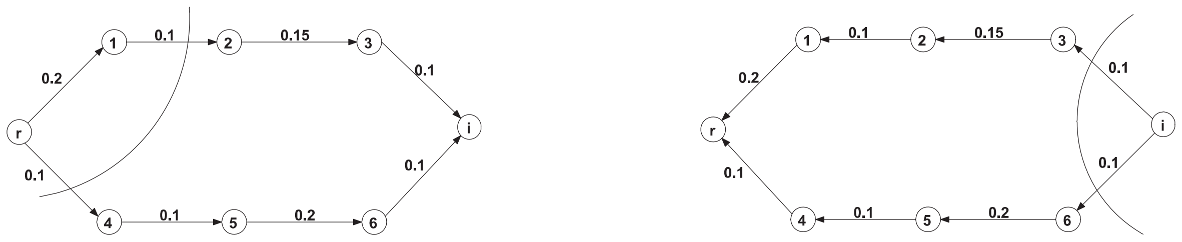
If $f < y_i$, we insert the violated constraint $x(\delta^+(S_r)) \geq y_i$ into the IP2 problem. We repeat this procedure for every profitable vertices i to find the violated constraints and then re-solve IP2 problem with branch-and-cut algorithm.

In order to speed up the process of detecting more violated cuts within the same *separation phase* (separation phase means checking the connectivity and finding the violated constraints for one pair $(r, i), i \in R$), we implement the *back cuts* and *nested cuts* methods which are also used in Ljubić et al. [31].

Back Cuts

As mentioned before, we use the maximum flow algorithm to find the minimum cut for every pair of (r, i). If the capacity of the minimum cut does not satisfy the connectivity constraint $x(\delta^-(S)) \geq y_i$, then this minimum cut is viewed as a violated cut and the corresponding violated constraint is added into IP2. Considering the fact that the minimum cuts in a directed graph are not always unique, more minimum cuts between r and i are expected to be found within one separation phase. In order to find another minimum cut which is closest to the profitable vertex i, say S_i $(i \in S_i)$, aside from the one already found which is closest to the root, say S_r $(r \in S_r)$, within the same separation phase, we applied the back cuts method.

First, we obtain the dual graph $\hat{G}_d$ by reversing the direction of the arcs in G_d while keeping the arc capacities the same, i.e., arc (i, j) is in $\hat{G}_d$ if and only if (j, i) is an arc of G_d, and the capacities of arc (i, j) in $\hat{G}_d$ and (j, i) in G_d are the same. Then we perform the maximum flow algorithm on $\hat{G}_d$. The vertex $i, i \in R, y_i > 0$ is viewed as the source



(a) Original graph and its minimum cut (b) Dual graph and its minimum cut

Figure 4.6: An example of the back cuts. Here, r is the root and i is the profitable vertex while all other vertices are non-profitable. The numbers on the edges are the capacities and y_i for the profitable vertex i is 0.5. The curved lines indicate the minimum cut set.

node and the root vertex r is viewed as the sink. By doing this, We change the maximum flow algorithm from $(f, S_r, S_i) = \text{MaxFlow}(G_d, x, r, i)$ to $(f, S_i, S_r) = \text{MaxFlow}(G_d, x, i, r)$. Finally, the output is obviously changed to the minimum cut closest to the profitable vertex i, $S_i, i \in S_i$.

For example, Figure 4.7a shows a network containing the root r and a profitable vertex i, together with non-profitable vertices 1–6. According to the maximum flow algorithm, $(f, S_r, S_i) = \text{MaxFlow}(G_d, x, r, i)$, the minimum cut set $\delta^+(S_r) = \{12, r4\}$, which is "closest" to r, is found first. It is obvious that the capacity of the minimum cut between r and i is $f = 0.2$. Since $f < y_i (y_i = 0.5)$, the corresponding constraint, $x(\delta^+(S_r)) \geq y_i$, is a violated constraint. Then we apply the back cuts within this separation phase: reverse the direction of the arcs while keeping the arc capacities the same. The dual graph is shown in Figure 4.6b. In the dual graph, the source is changed to i while the sink is changed to r. Using maximum flow algorithm $(f, S_i, S_r) = \text{MaxFlow}(G_d, x, i, r)$ we obtain the minimum cut set $\delta^+(S_i) = \{i3, i6\}$ whose corresponding constraint, $x(\delta^-(S_i)) \geq y_i$, is also a violated constraint in this separation phase.

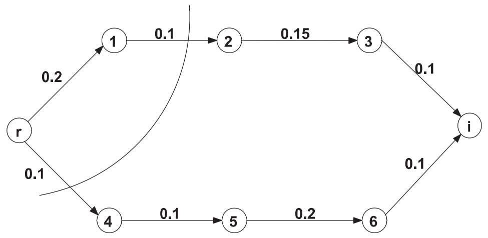
Nested Cuts

If r and i are not connected, as mentioned before, there may exist a bunch of violated cuts (recall that these are cuts whose capacities do not satisfy the connectivity constraint of r and i, $x(\delta^-(S)) \geq y_i$). Using the back cuts method can find two minimum cuts which may be the violated cuts, one of which is closest to the root r, the other one is closest to i. However, there may exist more violated cuts which we expect to find in the same separation phase. The nested cuts method is used to find further violated cuts with the maximum flow algorithm, by "removing" the current minimum cuts. It is described as follows.

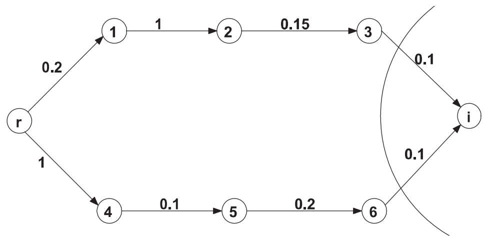
After finding the minimum cut S_r between r and i, we temporarily set the capacities of all the arcs $(u, v) \in \delta^+(S_r)$ to 1. Since $0 \leq y_i \leq 1$, the capacity of the current minimum cut must satisfy the connectivity constraint so that we "remove" this current minimum cut from the network. In fact, we can temporarily set the capacities of the current minimum cut to be any number as long as it ensures that the capacity of this minimum cut satisfies the connectivity constraint. We then apply the maximum flow algorithm for r and i on this resulting network again to find the minimum cut for the new network. Finally, if this new minimum cut is a violated cut, we add the corresponding violated constraint into IP2. We repeat this procedure until the maximum flow value is larger than or equal to y_i.

For example, as in Figure 4.7, after we get the minimum cuts $\delta^+(S_r) = \{12, r4\}$ and add the corresponding violated constraint $x(\delta^+(S_r)) \geq y_i$ into IP2, set the capacities on the edges in the minimum cuts to be 1. Figure 4.8e shows the graph after temporarily fixing the capacities on the edges in the minimum cuts to be 1. By applying the maximum flow algorithm in the modified graph, we find the minimum cut $\delta^+(S'_r) = \{23, 45\}$ with flow value $f = 0.25$ which is smaller than y_i. Thus we obtain the further violated cut and the corresponding violated constraint $x(\delta^+(S'_r)) \geq y_i$ in the same iteration. We repeat this procedure, until there is no such thing as a violated cut can be found.

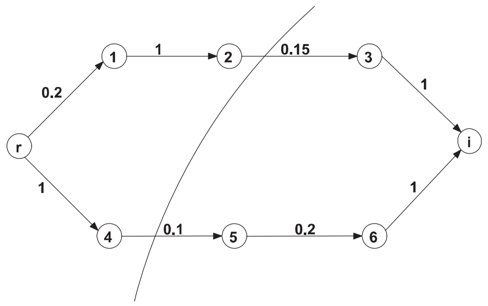
In all, the separation algorithm is as follows:



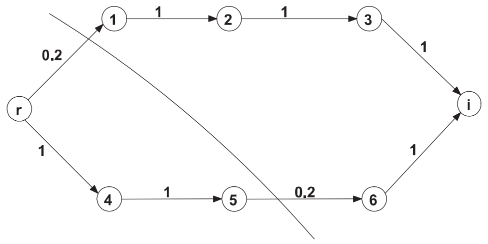
(a) Step 1 of nested cuts. The minimum cut $\{12, r4\}$ which is a violated cut is found. The capacities of the edges on this minimum cut is temporarily set to be 1.



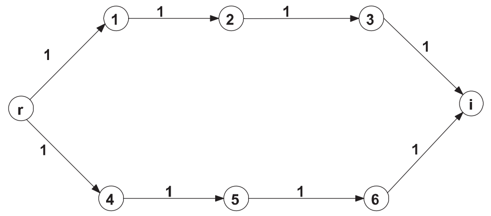
(b) Step 2 of nested cuts. The minimum cut $\{3i, 6i\}$ which is a violated cut is found. The capacities of the edges on this minimum cut are temporarily set to be 1.



(c) Step 3 of nested cuts. The minimum cut $\{23, 45\}$ which is a violated cut is found. The capacities of the edges on this minimum cut are temporarily set to be 1.



(d) Step 4 of nested cuts. The minimum cut $\{r1, 56\}$ which is a violated cut is found. The capacities of the edges on this minimum cut are temporarily set to be 1.



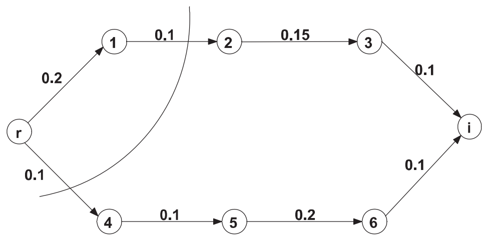
(e) Step 5 of nested cuts. The capacity of the minimum cut is larger than y_i, stop.

Figure 4.7: An example of nested cuts. r is the root and i is the profitable vertex while all other vertices are non-profitable vertices. The numbers on the edges are the capacities of these edges and y_i for the profitable vertex i is 0.5. The curved lines indicate the minimum cut set.

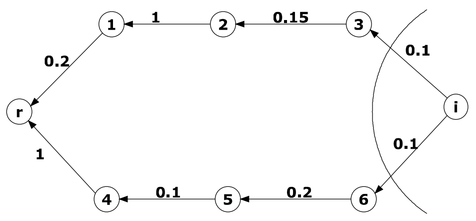
Algorithm 3 Separation algorithm

```
Require: A graph $G_d = (V_d, E_d, x)$. The input $x$ is the capacity vector.
Ensure: A set of violated inequalities included into IP2.
  for all $i \in R, y_i > 0$ do
    $x' = x$
    {Comment: introduce $x' = x$, then change $x'$ later but keep $x$ unchanged}
    repeat
      $f = MaxFlow(G, x', r, i)$
      if $f < y_i$ then
        Detect the cut $\delta^+(S_r)$ s.t. $x'(\delta^+(S_r)) = f, r \in S_r$
        Insert $x(\delta^+(S_r)) \geq y_i$ into the IP
        $x'_{ij} = 1, \forall (i,j) \in \delta^+(S_r)$
        {Comment: This is to find nested cuts.}
        if BACKCUTS then
          $f' = MaxFlow(G, x', i, r)$
          Detect the cut $\delta^-(S_i)$ s.t. $x'(\delta^-(S_i)) = f', i \in S_i$
          {Comment: If BACKCUTS= 1, back cuts method is used; otherwise, back cuts
          method is not used.}
          if $f' < y_i$ then
            Insert $x(\delta^-(S_i)) \geq y_i$ into the IP
            $x'_{ij} = 1, \forall (i,j) \in \delta^-(S_i)$
          end if
        end if
      end if
    until $f \geq y_i$
  end for
```

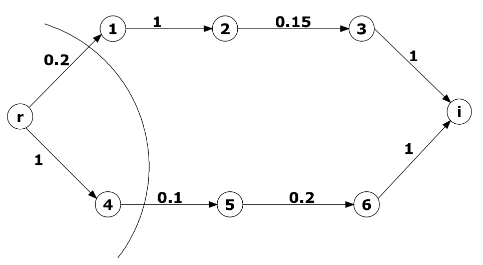
The back cuts and nested cuts can be combined together in the implementation. Figure 4.8 is an example of the combination of back cuts and nested cuts. Through the two methods, a set of violated constraints can be found within one separation iteration.



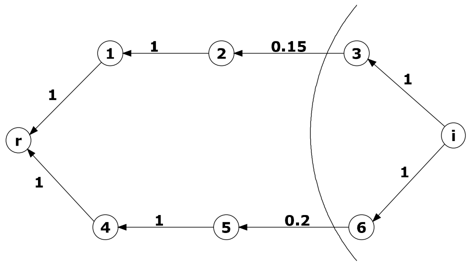
(a) Step 1: nested cuts. The minimum cut $\{12, r4\}$ which is a violated cut is found. The capacities of the edges on this minimum cut is temporarily set to be 1.



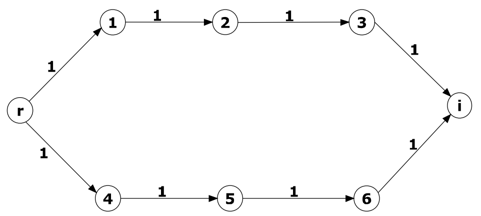
(b) Step 2: back cuts and nested cuts. The minimum cut $\{i3, i6\}$ which is a violated cut is found. The capacities of the edges on this minimum cut is temporarily set to be 1.



(c) Step 3: nested cuts. The minimum cut $\{r1, 45\}$ which is a violated cut is found. The capacities of the edges on this minimum cut is temporarily set to be 1.



(d) Step 4: back cuts and nested cuts. The minimum cut $\{23, 56\}$ which is a violated cut is found. The capacities of the edges on this minimum cut is temporarily set to be 1.



(e) Step 5. There is no such thing as a violated cut can be found. Stop.

Figure 4.8: An example of the combination of back cuts and nested cuts. r is the root and i is the profitable vertex while all other vertices are non-profitable vertices. The numbers on the edges are the capacities of these edges and y_i for the profitable vertex i is 0.5. The curved lines indicates the minimum cut set.

Chapter 5

Computational Results

5.1 Implementation Method and Environment

The algorithm explained in Chapter 4 was implemented in C++ in conjunction with IBM ILOG CPLEX version 12.1 and ILOG Concert Technology version 2.9. We mainly implemented the branch-and-cut algorithm and the separation algorithm without the heuristic method. The implementation of the heuristic method could be a a possible item for the future work. The test environment was a desktop PC with 64-bit AMD Phenom(tm) 9950 Quad-Core Processor 2.20GHz and 6.00GB RAM.

5.2 Experimental Results and Discussions

In order to test our algorithm on more realistic instances, we apply it to a set of grid graphs, which have a similar structure to that of a simplified real-word mine input. Figure 4.2a is a typical example of our test instance: a square matrix with profitable and non-profitable vertices. As the results we will show, we can solve all the testing instances to optimality in short time. For these instances, up to 81 vertices seem to be the largest size we can solve under our test environment.

In fact, some mining graphs may be larger, which might have more than thousands of vertices. Moreover, comparing our results with those in Ljubić et al. [31] which we follows at most, the size of the problems we can solve is much smaller. According to our observation and analysis, the possible reasons are as follows:

1. Our problem has an extra constraint which requires the exactly K profitable vertices in the solution. This constraint increases the difficulty of a standard PCST which is considered in Ljubić et al. [31] because there may be much more choices to be tried.

2. There is presumably a way around this problem because Ljubić et al. [31] applied the separation algorithm at each node of the branch-and-cut tree and used the branch-and-cut method once, but that due to limited time we were not able to find it. So in our algorithm, as described in Chapter 4, not only one branch-and-cut procedure is needed. In some cases, even a wealth of branch-and-cut procedures is required to solve the problem. Branch-and-cut consumes the most memory and it is easy to cause the memory problem in CPLEX, which will be discussed in details in Section 5.3 of this Chapter. In addition, this could affect the run time of the algorithms.

3. There is no preprocessing method in our algorithm. As described before, the preprocessing method provided in Ljubić et al. [31] is not suitable for KPCST which has an extra cardinality constraint. While the preprocessing method plays a crucial role in the problem of Ljubić et al. [31], which can significantly reduce the size of the problem.

Table 5.1 shows the experimental results for the grid graph instances. All of them are based on grid graphs built by generating the vertices in an $n \times n$ $(n \leq 9)$ square matrix. The prizes of the profitable vertices are set uniformly to be 100 while the edge costs are random integral numbers ranging from 1 to 10. We chose the profitable vertices in the square matrix randomly. We tested the instances with different numbers of vertices, numbers of profitable vertices and the values of K (number of profitable vertices specified for the solution). These combinations lead to thousands of instances. We tested about 200 instances and all these instances are solved to optimality. As a matter of fact, in a realistic problem, the prizes may vary randomly. So setting the prizes of profitable vertices to be the random number from 1 to 100 and setting the costs of the edges to be uniformly 1, or setting both the prizes and the costs to be random numbers are some of the other variations we would try if time permitted.

For Table 5.1, in the first six columns we report the problem parameters: the problem ID, the number of vertices $|V|$ in the problem, the number of profitable vertices $|N|$, the number of edges $|E|$, the number of edges $|E'|$ in the transformed directed graph and the value of K. In the following six columns we report the optimal objective value we get, the total number of the violated constraints we found by the algorithm, the number of the branch-and-cut nodes in the last iteration (i.e., the last time to run the branch-and-cut method), the total CPU time of the computation in seconds, the number of *zero-half cuts* and the number of the *Gomory cuts* added automatically by CPLEX. zero-half cuts and Gomory cuts are constraints added to the model to restrict non-integer solutions that would otherwise be solutions of the continuous relaxation. The addition of cuts usually reduces the number of branches needed to solve an IP problem. In fact, CPLEX often adds several kinds of cuts in the branch-and-cut algorithm, such as clique cuts, cover cuts, zero-half

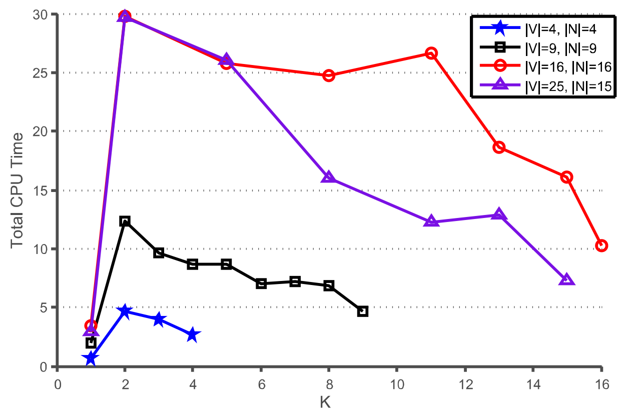


Figure 5.1: The relationship between CPU time and the cardinality of the profitable vertices K, while fixing $|V|$ and $|N|$. The CPU time gets shorter as K increases. The CPU time is shortest and the number of the violated constraints is 0 when $K = 1$. The CPU time is the longest when the difference between $|N|$ and K is the largest. As K increases, i.e., the difference gets smaller, the CPU time gets shorter.

cuts, Gomory cuts and so on. Zero-half cuts and Gomory cuts are two cuts applied in our IP problem.

All the instances have been solved to optimality. 6 instances require more than 10 branching nodes whereas the other ones are solved in less than 10 nodes. In fact, according to our observation, we can only solve the instances which require less than 15 branching nodes under our test environment. This is a major memory problem which restrains the size of the KPCST problem we can solve. As mentioned before, this problem is probably due to the reason that several times of the branch-and-cut algorithm are used in the algorithm. This memory issue will be discussed in detail in the next section.

In order to find the relationship between $|V|$, $|N|$ and K, we plot the following several figures.

Figure 5.1 shows that, in general, the CPU time gets shorter as K increases while fixing a certain $|V|$ and $|N|$. The CPU time is shortest and the number of the violated constraints is 0 when $K = 1$, this is because that the problem is trivial in this case: we only select the vertex with the largest prize; the costs of the edges need not to be concerned. Therefore, as can be seen that the CPU time jumps high when K grows from 1 to 2. Another thing one can notice is that the CPU time is the longest when the difference between $|N|$ and K is the largest. As K increases, i.e., the difference gets smaller, the CPU time gets shorter. The reason seems to be the increased difficulty generated by more choices and combinations of the selected profitable vertices when $|N|$ is big and K is small. Finally, the instances with

Table 5.1: Computational results of the algorithm for the mining problem

Problem Parameters						Optimal	No. of violated	No. of	Total	No. of	No. of
ID	$\|V\|$	$\|N\|$	$\|E\|$	$\|E'\|$	K	value	constraints	B&C Nodes	CPU time [a]	Zero-half Cuts	Gomory Cuts
5.11	25	1	40	81	1	-100	0	1	2.091	0	0
5.21	25	2	40	82	1	-100	0	1	2.612	0	0
5.22	25	2	40	82	2	-197	10	1	7.654	0	0
5.52	25	5	40	85	2	-190	15	3	10.873	0	0
5.54	25	5	40	85	4	-372	31	1	13.812	0	0
5.55	25	5	40	85	5	-462	43	1	13.865	0	0
5.81	25	8	40	88	1	-100	0	1	2.812	0	0
5.84	25	8	40	88	4	-392	32	4	21.814	2	1
5.86	25	8	40	88	6	-582	83	4	18.466	2	1
5.88	25	8	40	88	6	-748	56	3	16.082	3	1
5.111	25	11	40	91	1	-100	0	1	2.831	0	0
5.112	25	11	40	91	2	-189	67	3	14.871	0	0
5.113	25	11	40	91	3	-297	63	4	16.912	0	0
5.117	25	11	40	91	7	-684	53	5	10.522	2	1
5.119	25	11	40	91	9	-864	91	5	8.813	0	0
5.1111	25	11	40	91	11	-1058	88	1	6.101	0	0
5.151	25	15	40	95	1	-100	0	1	2.912	0	0
5.152	25	15	40	95	2	-198	45	1	29.733	0	0
5.155	25	15	40	95	5	-478	80	2	26.007	0	0
5.158	25	15	40	95	8	-782	98	5	16.032	1	1
5.1511	25	15	40	95	11	-1073	145	6	12.257	3	1
5.1513	25	15	40	95	13	-1244	108	3	12.881	0	0
5.1515	25	15	40	95	15	-1446	99	1	7.312	0	0
5.181	25	18	40	98	1	-100	0	1	2.744	0	0
5.182	25	18	40	98	2	-194	112	6	38.028	0	0
5.186	25	18	40	98	6	-583	165	5	31.214	4	2
5.1811	25	18	40	98	11	-1073	154	5	27.856	1	0
5.1815	25	18	40	98	15	-1446	152	7	16.776	3	1
5.1818	25	18	40	98	18	-1735	211	3	14.265	1	1
5.211	25	21	40	101	1	-100	0	1	3.430	0	0
5.212	25	21	40	101	2	-197	57	3	40.980	0	0
5.215	25	21	40	101	5	-488	142	4	36.765	6	1
5.2111	25	21	40	101	11	-1066	189	3	37.010	2	1
5.2116	25	21	40	101	16	-1554	121	6	25.984	3	2
5.2121	25	21	40	101	21	-2031	187	1	19.776	0	0
5.251	25	25	40	105	1	-100	0	1	3.859	0	0
5.252	25	25	40	105	2	-199	70	4	81.294	0	0
5.255	25	25	40	105	5	-496	302	8	69.644	6	2
5.2510	25	25	40	105	10	-969	248	3	60.169	1	2
5.2516	25	25	40	105	16	-1559	230	11	44.850	11	0

[a] The time is in seconds and measured by a function with a high resolution in the range of 10 milliseconds to 16 milliseconds provided by Windows

Continued Table 5.1: Computational results of the algorithm for the mining problem

Problem Parameters						Optimal	No. of violated	No. of	Total	No. of	No. of
ID	$\|V\|$	$\|N\|$	$\|E\|$	$\|E'\|$	K	objective value	constraints	B&C Nodes	CPU time	Zero-half Cuts	Gomory Cuts
5.2520	25	25	40	105	20	-1944	296	12	45.246	10	1
5.2525	25	25	40	105	25	-2421	150	1	25.506	0	0
6.11	36	1	60	121	1	-100	0	1	2.644	0	0
6.22	36	2	60	122	2	-180	10	1	7.656	0	0
6.54	36	5	60	125	4	-381	156	6	37.890	0	0
6.55	36	5	60	125	5	-478	167	5	35.764	0	0
6.81	36	8	60	128	1	-100	0	1	2.955	0	0
6.82	36	8	60	128	2	-194	132	5	53.901	6	0
6.86	36	8	60	128	6	-581	109	6	33.575	2	0
6.151	36	15	60	134	1	-100	0	1	3.890	0	0
6.155	36	15	60	134	5	-490	135	1	30.111	0	0
6.158	36	15	60	134	8	-781	143	5	29.012	5	2
6.1515	36	15	60	134	15	-1430	289	4	25.777	3	1
6.202	36	20	60	140	2	-194	188	5	68.954	1	1
6.205	36	20	60	140	5	-487	201	5	69.113	8	2
6.209	36	20	60	140	8	-878	112	5	53.342	0	0
6.2015	36	20	60	140	14	-1453	232	4	35.901	2	0
6.2018	36	20	60	140	18	-1733	256	4	25.589	0	0
6.2020	36	20	60	140	20	-1913	256	5	19.782	3	0
6.252	36	25	60	145	2	-195	134	4	72.198	0	0
6.259	36	25	60	145	9	-863	399	5	60.489	7	4
6.2515	36	25	60	145	13	-1476	255	5	45.643	0	0
6.2520	36	25	60	145	20	-1945	345	4	37.894	0	0
6.2525	36	25	60	145	25	-2401	356	3	28.964	0	0
6.302	36	30	60	150	2	-198	165	2	92.301	0	0
6.308	36	30	60	155	8	-747	789	4	167.432	12	5
6.3015	36	30	60	155	15	-1466	543	5	57.876	0	0
6.3025	36	30	60	155	25	-2423	543	5	42.765	4	1
6.3030	36	30	60	155	30	-2890	454	2	26.765	0	0
6.361	36	36	60	161	1	-100	0	1	4.765	0	0
6.364	36	36	60	161	4	-396	112	1	48.454	0	0
6.368	36	36	60	161	8	-786	236	7	72.634	1	1
6.3620	36	36	60	161	20	-1950	428	7	111.588	4	2
6.3625	36	36	60	161	25	-2433	330	3	77.439	0	0
6.3630	36	36	60	161	30	-2890	262	2	57.321	0	0
6.3636	36	36	60	161	36	-3485	248	1	53.571	0	0
7.22	49	2	84	170	2	-185	14	1	16.987	0	0
7.54	49	5	84	173	4	-385	56	4	45.764	1	0
7.55	49	5	84	173	5	-478	89	6	37.222	0	0
7.82	49	8	84	176	2	-196	87	8	65.453	5	2
7.88	49	8	84	175	8	-763	210	5	45.875	0	0

Continued Table 5.1: Computational results of the algorithm for the mining problem

Problem Parameters						Optimal	No. of violated	No. of	Total	No. of	No. of
ID	$\|V\|$	$\|N\|$	$\|E\|$	$\|E'\|$	K	objective value	constraints	B&C Nodes	CPU time	Zero-half Cuts	Gomory Cuts
7.158	49	15	84	182	8	-762	289	7	86.433	10	1
7.1515	49	15	84	182	15	-1422	654	9	78.658	1	1
7.252	49	25	84	192	2	-194	123	9	81.099	4	0
7.2525	49	25	84	192	25	-2422	323	5	64.765	0	0
7.341	49	34	84	201	1	-100	0	1	6.054	0	0
7.345	49	34	84	201	5	-489	214	7	68.755	0	0
7.3425	49	34	84	201	25	-2418	354	8	74.754	5	1
7.491	49	49	84	216	1	-100	0	1	6.785	0	0
7.495	49	49	84	216	5	-495	198	4	87.407	1	1
7.4927	49	49	84	216	27	-2634	879	8	143.875	4	2
7.4949	49	49	84	216	49	-4766	192	1	55.396	0	0
8.22	64	2	112	226	2	-176	25	3	65.342	0	2
8.54	64	5	112	229	4	-396	54	6	68.644	2	0
8.55	64	5	112	229	5	-475	186	10	73.421	0	0
8.82	64	8	112	232	2	-198	87	4	69.075	0	0
8.88	64	8	112	232	8	-732	298	14	164.975	0	0
8.6445	64	64	112	286	45	-4389	836	11	271.223	1	2
9.22	81	2	144	290	2	-195	32	8	198.875	5	0
9.54	81	5	144	293	4	-391	208	3	165.754	0	0
9.55	81	5	144	293	5	-491	245	1	234.865	0	0
9.88	81	8	144	296	8	-756	456	13	287.553	5	4

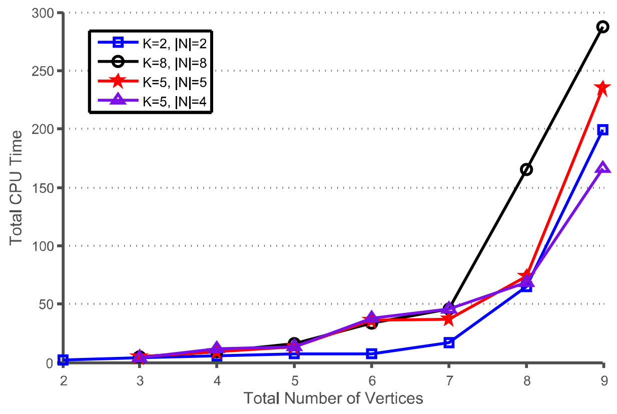


Figure 5.2: The relationship between CPU time and the number of the vertices $|V|$, while fixing K and $|N|$. The CPU time increases almost exponentially while $|V|$ increases.

more vertices are harder to solve according to the comparison of the general CPU time of the four lines in Figure 5.1. This is demonstrated in Figure 5.2.

From Figure 5.2, we can see that the CPU time increases almost exponentially while $|V|$ increases.

Figure 5.3 shows the relationship between the CPU time and the number of profitable vertices $|N|$. Apparently the CPU time increases as $|N|$ increases.

A lot of related problems in other papers are solved with edge costs in a small range. For example, in Aneja [1], they used integer edge costs in the range of [1,10]; and in Wong [40], they used real edge cost in the range of [0,1]. As mentioned earlier, we use integer edge costs between 1 and 10. We also want to see how a wider range of edge costs might affect the problem difficulty.

Table 5.2 presents the results for the experiments with edge costs ranging from 1 to 10 and edge costs ranging from 500 to 10500. As before, the problem parameters: problem ID, the number of vertices $|V|$, the number of profitable vertices $|N|$ and the value of K are listed. We mainly compare the CPU time and the violated constraints in the two cases. Figure 5.4 and 5.5 clearly show the comparison.

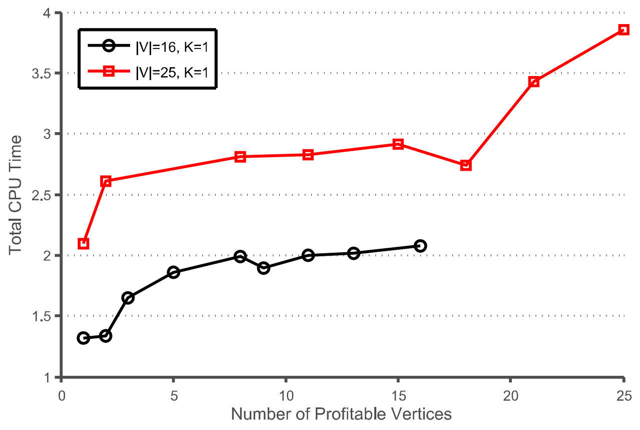


Figure 5.3: The relationship between CPU time and the number of the profitable vertices $|N|$, while fixing K and $|V|$. The CPU time increases as $|N|$ increases.

Table 5.2: Experimental results for the edge costs from 1 to 10 and the edge costs from 500 to 1500

Problem Parameters				Random edge cost $(1 - 10)$		Random edge cost $(500 - 10500)$	
ID	$\|V\|$	$\|N\|$	K	CPU time [a]	No. of violated constraints	CPU time	No. of violated constraints
2.41	4	4	1	0.654	0	0.754	0
3.51	9	5	1	2.003	0	1.987	0
3.92	9	9	2	12.334	76	13.754	65
3.94	9	9	4	8.645	65	7.854	51
4.162	16	16	2	29.754	187	68.765	123
4.165	16	16	5	25.765	45	35.432	56
4.168	16	16	8	24.766	231	87.432	254
5.252	25	25	2	81.294	70	234.321	87
5.2510	25	25	10	60.169	248	63.532	256
5.2520	25	25	20	45.246	296	36.643	213
6.368	36	36	8	72.634	236	432.442	254
6.3630	36	36	30	57.321	262	189.321	221
6.3636	36	36	36	53.571	248	91.645	243
7.495	49	49	5	87.467	198	125.566	143
7.4927	49	49	27	143.875	879	265.654	902
7.4949	49	49	49	55.396	192	201.454	234
8.22	64	2	2	65.342	25	45.298	20
8.55	64	5	5	73.421	186	101.892	90
8.6445	64	64	45	271.223	836	503.432	989

[a]The time is in seconds

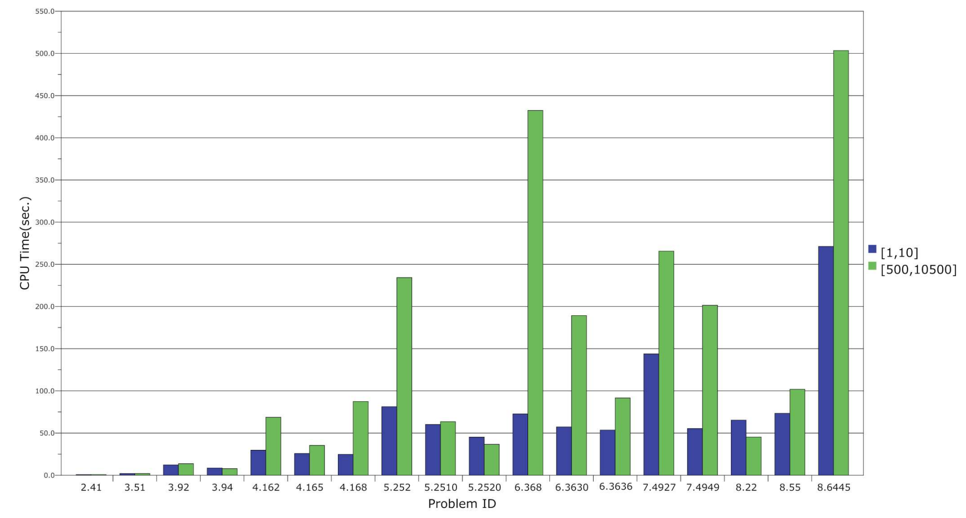


Figure 5.4: The comparison of the CPU time between the edge costs in the range of [1,10] and the edge costs in the range of [500,10500]

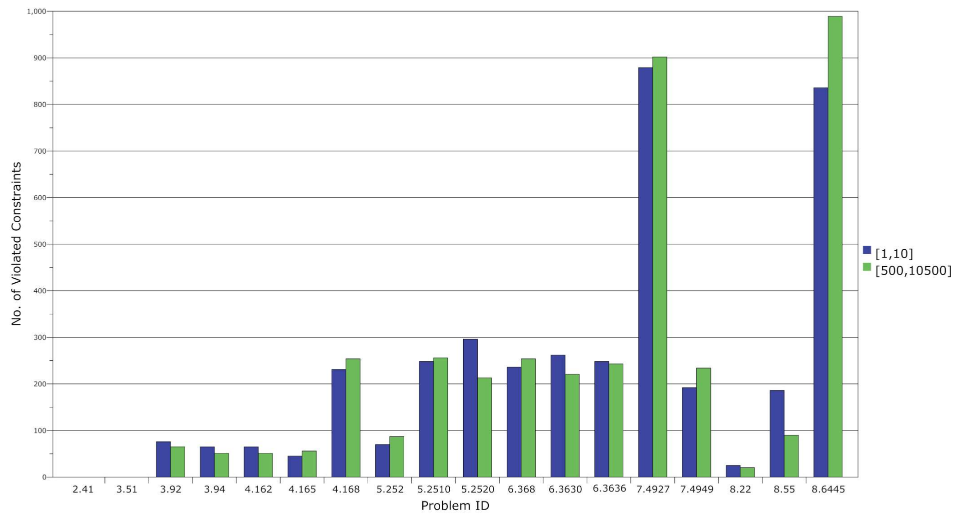


Figure 5.5: The comparison of the number of the violated constraints between the edge costs in the range of [1,10] and the edge costs in the range of [500,10500]

In Figure 5.4, the CPU time grows when the edge costs range changes. We may conclude that the edge costs range may affect problem difficulty for IP2 problem. Moreover, it can be observed that the increased CPU time is two to several times of that for the smaller edge costs problem, while almost all the increased violated constraints are less than twice. Even for instance 6.3630 and 8.55, the CPU time grows but the violated constraints drops. The reason seems to be more time consumed in the IP solver with the larger range edge costs.

We also applied our algorithm to the instances with real edge costs ranging from 1 to 10 to see whether the change affects the problem difficulty by comparing the changed CPU time and the number of violated constraints. Table 5.3 lists the results for our experiments with the real edge costs instances.

Table 5.3: Experimental results for the integer edge costs and the real edge costs

Problem Parameters				Integer edge costs		Real edge costs	
ID	$\|V\|$	$\|N\|$	K	CPU time [a]	No. of violated constraints	CPU time	No. of violated constraints
2.41	4	4	1	0.654	0	0.568	0
3.51	9	5	1	2.003	0	1.872	0
3.92	9	9	2	12.334	76	19.897	86
3.94	9	9	4	8.645	65	5.435	78
4.162	16	16	2	29.754	187	19.765	167
4.165	16	16	5	25.765	45	23.515	45
4.168	16	16	8	24.766	231	31.543	289
5.252	25	25	2	81.294	70	79.765	78
5.2510	25	25	10	60.169	248	78.765	234
6.368	36	36	8	72.634	236	87.543	246
6.3630	36	36	30	57.321	262	59.608	269
6.3636	36	36	36	53.571	248	37.041	267
7.4927	49	49	27	143.875	879	176.764	912
7.4949	49	49	49	55.396	192	53.765	196
8.22	64	2	2	65.342	25	61.654	37
8.55	64	5	5	73.421	186	109.709	124
8.6445	64	64	45	271.223	836	253.986	901

[a]The time is in seconds

Figure (5.6) and (5.7) shows the comparison of the results.

From the above two figures, we can see that the difficulty does not seem strictly correlated with the data type of the edge costs since CPU time sometimes grows (for example, instances 3.92, 4.168, 5.2510 and 7.4927) and sometimes drops (for example, instances 4.162, 4.165, 6.3636 and 8.22). And the CPU time does not increase or decrease too much. The comparison of the number of the violated constraints is similar.

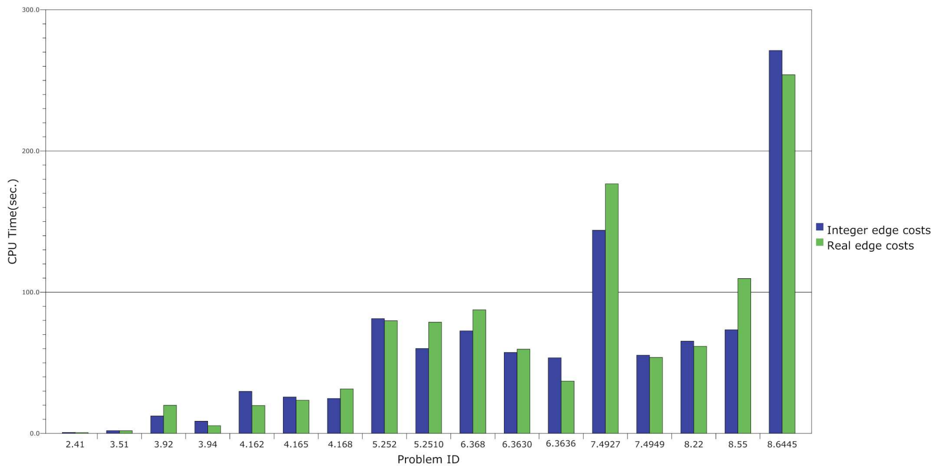


Figure 5.6: The comparison of the CPU time between the integer edge costs and the real edge costs

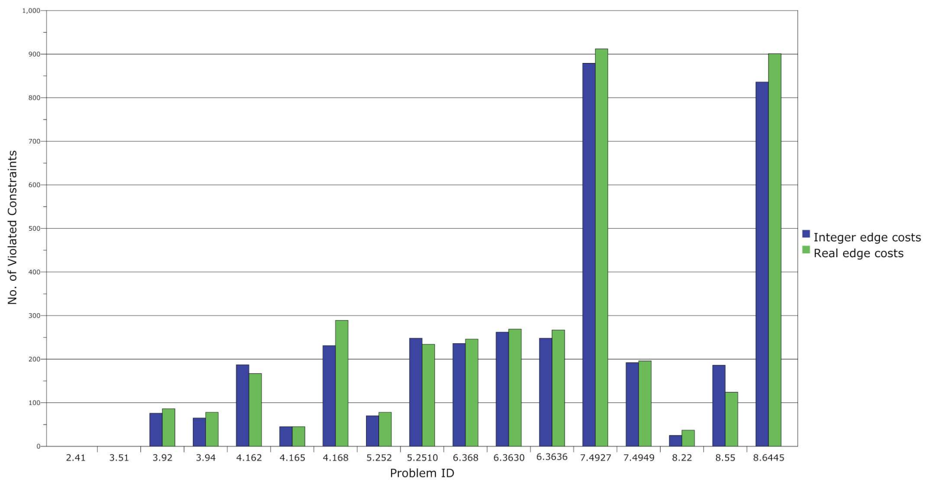
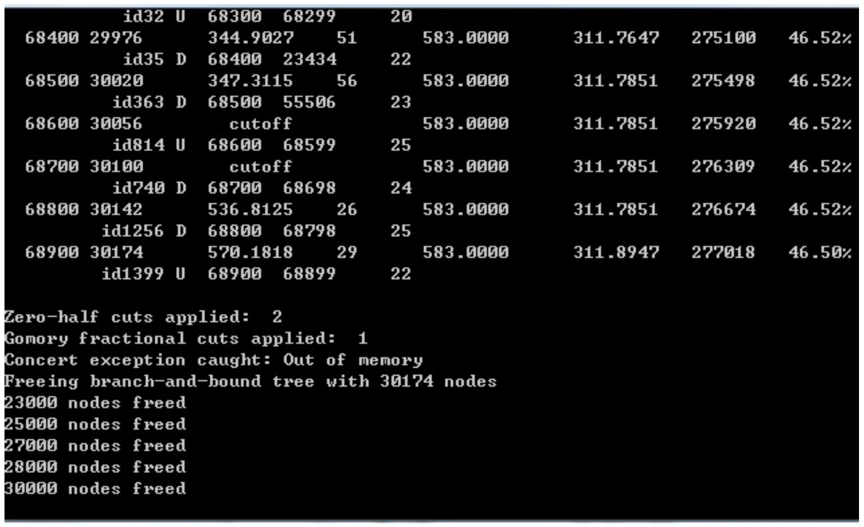


Figure 5.7: The comparison of the number of the violated constraints between the integer edge costs and the real edge costs

5.3 Out of Memory

As mentioned before, we tried to find ways to incorporate separation in each node of branch-and-cut but did not manage in the time available. Through our algorithm, the instances with up to 81 vertices are solved to optimality under our test environment. But for some larger size instances, we may get the result like Figure 5.8



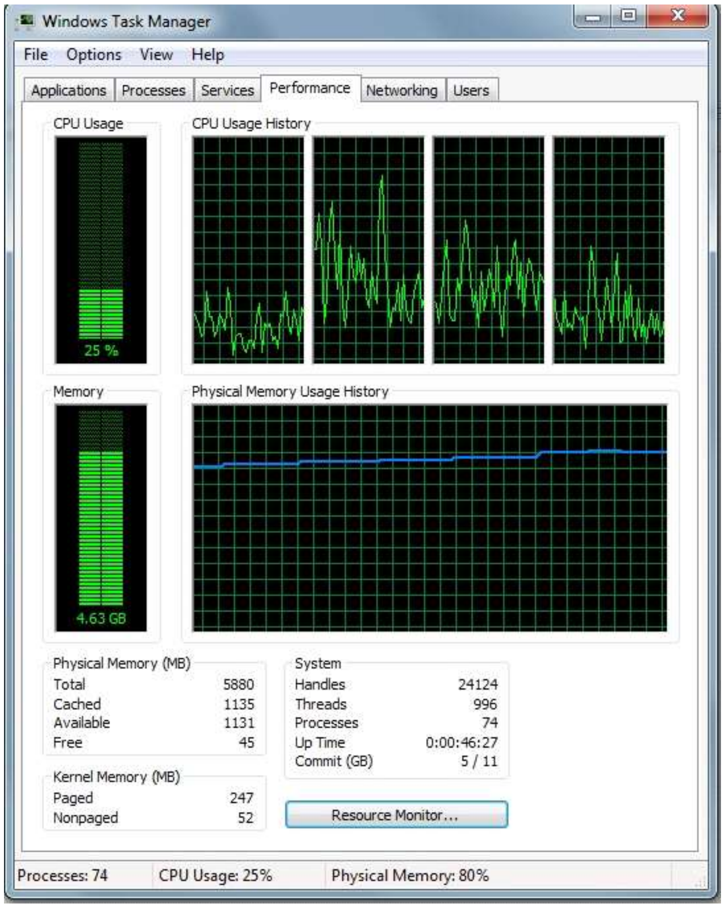
```
          id32 U  68300  68299      20
 68400 29976      344.9027    51      583.0000      311.7647   275100   46.52%
          id35 D  68400  23434      22
 68500 30020      347.3115    56      583.0000      311.7851   275498   46.52%
         id363 D  68500  55506      23
 68600 30056        cutoff            583.0000      311.7851   275920   46.52%
         id814 U  68600  68599      25
 68700 30100        cutoff            583.0000      311.7851   276309   46.52%
         id740 D  68700  68698      24
 68800 30142      536.8125    26      583.0000      311.7851   276674   46.52%
        id1256 D  68800  68798      25
 68900 30174      570.1818    29      583.0000      311.8947   277018   46.50%
        id1399 U  68900  68899      22

Zero-half cuts applied:  2
Gomory fractional cuts applied:  1
Concert exception caught: Out of memory
Freeing branch-and-bound tree with 30174 nodes
23000 nodes freed
25000 nodes freed
27000 nodes freed
28000 nodes freed
30000 nodes freed
```

Figure 5.8: The result of running out of memory

As a matter of fact, running out of memory is a very common difficulty with IP problems using CPLEX. This problem almost always occurs when the branch-and-cut tree becomes so large that insufficient memory remains to solve a continuous LP. The information about a tree that CPLEX accumulates in memory can be substantial. In particular, CPLEX saves a basis for every unexplored node. The list of unexplored nodes itself can become very long for large or difficult problems. How large the unexplored node list can depends on the actual amount of memory available, the size of the problem, and the algorithm selected. Therefore, if we increase the amount of available memory, we extend the problem-solving capability of CPLEX. Unfortunately, when a problem fails because of insufficient memory, it is difficult to project how much further the process needed to go and how much more memory is needed to solve the problem. Memory failure problem in CPLEX can be avoided by some strategies suggested by CPLEX. For example, we can reset the tree memory parameter, use node files for storage and change the selection strategy in branch-and-cut tree.

Another reason is that not only one branch-and-cut procedure is needed in our algorithm. In some cases, even a wealth of branch-and-cut procedures is required to solve the problem. It may run out of memory even though every single branch-and-cut tree can be solved since the memory cannot be released after every single branch-and-cut procedure according to CPLEX mechanism. The memory consumption accumulates and finally leads to excessive memory consumption. Figure 5.9 shows what happens to the physical memory in computer.



Physical Memory (MB)	
Total	5880
Cached	1135
Available	1131
Free	45

Kernel Memory (MB)	
Paged	247
Nonpaged	52

System	
Handles	24124
Threads	996
Processes	74
Up Time	0:00:46:27
Commit (GB)	5 / 11

Processes: 74 CPU Usage: 25% Physical Memory: 80%

Figure 5.9: The change of the physical memory

It can be observed that the physical memory consumption jumps after every branch-and-cut procedure and finally reaches the maximum memory.

5.4 Back Cuts and Nested Cuts

The back cuts method is used in our default implementation, while the nested cuts method is not used. In this section, we compare the performance of the algorithm with and without back cuts and nested cuts.

As before, we list the problem parameters and, the CPU time and the number of the violated constraints. We also list the number of the major iterations in order to give a clearer explanation.

Table 5.4: Comparison over the instances for separation with or without back cuts

Problem Parameters				with back cuts (without nested cuts)			without back cuts (without nested cuts)		
ID	$\|V\|$	$\|N\|$	K	CPU time [a]	No. of violated constraints	No. of major iterations	CPU time	No. of violated constraints	No. of major iterations
2.41	4	4	1	0.639	0	1	0.646	0	1
3.33	9	3	3	4.672	11	5	22.771	89	45
3.51	9	5	1	1.045	0	1	0.905	0	1
3.92	9	9	2	7.722	42	22	99.872	155	142
3.94	9	9	4	5.288	30	6	141.227	389	135
4.162	16	16	2	17.020	36	19	409.707	628	474
4.165	16	16	5	18.5640	78	14	512.321	721	356
4.168	16	16	8	20.920	100	10	$>$ 4 hours	n/a	n/a
5.252	25	25	2	81.294	70	36	out of memory	n/a	n/a
5.2510	25	25	10	60.169	248	21	$>$ 4 hours	n/a	n/a
6.368	36	36	8	72.634	236	25	$>$ 4 hours	n/a	n/a
6.3615	36	36	15	66.332	264	13	out of memory	n/a	n/a
6.3630	36	36	30	57.321	262	17	out of memory	n/a	n/a
6.3636	36	36	36	53.571	248	5	out of memory	n/a	n/a

[a]The time is in seconds

Table 5.4 documents the crucial role of using the back cuts method. The CPU time and the number of major iterations increase dramatically without the back cuts, especially, when the problem size grows. As in table 5.4, some of instances (for example, instances 4.168, 5.2510 and 6.368) without back cuts cannot be solved in a reasonable time limit (4 hours) to optimality; and some of instances (for example, instances 5.252, 6.3615 and 6.3636) cannot be solved due to the memory problem mentioned before. Thus, the introduction of the back cuts method significantly reduces the running time and increases the problem-solving capability.

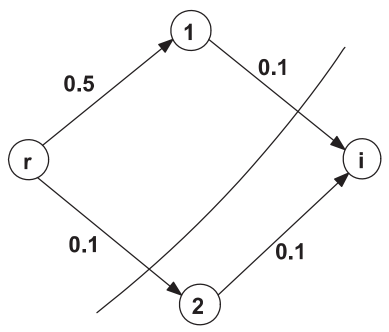
Table 5.5: Comparison over the instances for separation with or without nested cuts

Problem Parameters				without nested cuts (with back cuts)			with nested cuts (with back cuts)		
ID	$\|V\|$	$\|N\|$	K	CPU time [a]	No. of violated constraints	No. of major iterations	CPU time	No. of violated constraints	No. of major iterations
2.41	4	4	1	0.639	0	1	1.871	0	1
3.33	9	3	3	4.672	11	5	10.621	8	19
3.51	9	5	1	1.924	0	1	1.933	0	1
3.92	9	9	2	7.722	42	22	22.187	35	36
3.94	9	9	4	5.288	30	6	220.123	35	26
4.77	16	7	7	3.479	23	3	10.782	13	32
4.162	16	16	2	17.020	36	19	25.876	21	58
4.165	16	16	5	18.564	78	14	101.176	68	82
4.168	16	16	8	20.920	100	10	987.113	178	781
5.252	25	25	2	81.294	70	36	out of memory	n/a	n/a
5.2510	25	25	10	60.169	248	21	2981.212	172	2671
6.368	36	36	8	72.634	236	25	out of memory	n/a	n/a
6.3615	36	36	15	66.332	264	13	out of memory	n/a	n/a
6.3630	36	36	30	57.321	262	17	out of memory	n/a	n/a
6.3636	36	36	36	53.571	248	5	out of memory	n/a	n/a

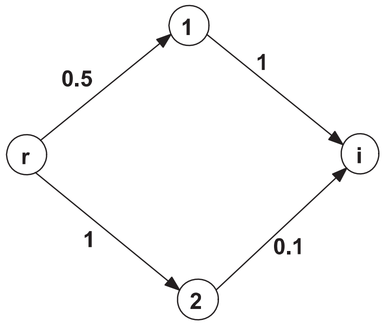
[a] The time is in seconds

Table 5.5 shows that, on the contrary, the problem with the nested cuts method takes much longer CPU time to solve, even for the small size problems, although the reason to use nested cuts method is to reduce the running time. In addition, by using the nested cuts, many larger instances could not be solved to optimality while the same problem without nested cuts can be solved in several seconds.

As explained in Section 4.2.3, the nested cuts method is supposed to find more violated constraints in one separation phase and it indeed helps find more violated constraints to reduce the running time in some problems (see the example in Figure 4.7). However, in our problem, it is not like this. It can be observed from table 5.5 that sometimes less violated constraints are found, therefore, much more major iterations are required. The reason seems to be that setting the capacities of edges in the current minimum cuts to be 1 might miss more violated cuts which are supposed to be found. An example is shown in Figure 5.10. Two violated cuts $\{1i, r2\}$ and $\{1i, 2i\}$ are supposed to be found in one separation phase by back cuts and without nested cuts. If the nested cuts method is applied, however, the violated cut $\{1i, 2i\}$ cannot be found within one separation phase. In addition, the violated cut $\{1i, 2i\}$ is important for the connectedness of i. Hence, in this case, the nested cuts method increases the running time.



(a) The minimum cut $\{1i, r2\}$ which is a violated cut is found. Then according to the nested cuts, the capacities of the edges on the minimum cut is set to be 1.



(b) No more violated cut can be found.

Figure 5.10: An example of the problem caused by nested cuts. Here, r is the root and i is the profitable vertex while all other vertices are non-profitable vertices. The numbers on the edges are the capacities of these edges and y_i for the profitable vertex i is 0.5. The curved lines are the minimum cuts. Due to the nested cuts, only one violated cut is found in one separation phase.

Chapter 6

Conclusions

In this paper, we model the planning of the mining networks to the K-cardinality Prize-Collecting Steiner Tree problem (KPCST): finding the best tree connecting K profitable vertices. KPCST problem is a PCST problem with restrictions, whereas PCST is a generalization of the Steiner tree problem, where the terminal vertices in the solution are fixed.

The target of this paper is to construct a formulation and give an algorithm to solve this KPCST problem to optimality within a reasonable running time and physical memory. The algorithm which is a variant of the one proposed by Ljubić et al. [31] uses branch-and-cut method and takes advantage of the connectivity property of the solution. We also introduce a heuristic method to reduce the running time. Two methods: the back cuts method and the nested cuts method, are also presented in this paper.

When implementing our algorithm on a desktop PC with 64-bit AMD Phenom(tm) 9950 Quad-Core Processor 2.20GHz and 6.00 GB RAM, we successfully solved the problem with up to 81 vertices to optimality in several seconds. We tested about 200 instances out of thousands of instances depending on different number of vertices, number of the profitable vertices and number of the selected profitable vertices in the solution. All the instances were solved to optimality in several seconds. We also compared the results with different ranges ([1,10] and [500,10500]) and data types (integer and real number) of edge costs. According to the computational result, we may conclude that the larger edge costs range makes the problem harder to solve, while the data type of the edge costs does not affect the problem difficulty. Finally, the experimental result shows that back cuts method is crucial, but nested cuts method is not suitable for this problem.

According to our computational results, lack of memory is a main barrier to solve large size problem. Thus, possible future work includes:

1. Improve the algorithm by the following possible methods:

- Embedding the separation procedure into the branch-and-cut method, i.e., using the separation procedure at every node of the branch-and-cut tree.
- Adding reduction tests and preprocessing to extend the problem-solving capability of the algorithm.
- Finding some other ways to get CPLEX to release its memory when we start another branch-and-cut algorithm.
- Adding the heuristic method during implementation. Doing experiments for the heuristic method to check whether and how the heuristic method could improve the algorithm.

2. Do experiments on the variants of the planning of the mining problem described in section 3.4.

APPENDICES

Appendix A

Major parts of C++ Code

A.1 Separation Algorithm

This part is to implement the separation algorithm. This version contains back cuts and nested cuts.

```
int cnt_cons = 0; //the no. of the violated constraints
int cnt_iter = 0; //the no. of the main iterations
do{
    nostop = 0;
    cplex.solve(MyBranchGoal(env, y, x0, x, nodesno));
    cnt_iter ++;
    cplex.getValues(valsy, y);
    cplex.getValues(valsx0, x0);
    cplex.getValues(valsprofx, profx);
    for (IloInt j = 0; j < nodesno-1; j++) {
        IloNumArray   valsxtmp(env);
        for (IloInt p = 0; p < nodesno-1; p++){
            if (p==j+1 || p==j-1 || p==j+nosqr|| p==j-nosqr){
                valsxtmp.add(cplex.getValue(x[j][p]));
            }
            else valsxtmp.add(0);
        }
        valsx[j] = IloNumArray(valsxtmp);
    }
    for (g=nodesno; g<nodesno+profno; g++){
        if (valsy[g]>0){
        NumMatrix        valsx1=valsx;
        IloNumArray      valsprofx1 = valsprofx;
        IloNumArray      valsx01 = valsx0;
        do{
        cnt_text = writefile(g, valsx1, valsx01, prize, valsprofx1, cplex, ptr_text);
        max_flow(&vv, ptr_text, cnt_text);
        IloExpr addcons1(env);
        if ( vv.flow_val < valsy[g]){
        nostop = 1;
        //to find the arcs in the cut
        IloInt s=0, l=0;
        IloBool check1, check2, check3;

        for (IloInt t=1; t<nodesno; t++){
            check1 = 0;
            for (IloInt k=0; k<vv.cnt1; k++){
                if (t == vv.max_flow_set1[k]){
                    check1 = 1;}
                }
            for (IloInt j=1; j<nodesno; j++){
                check2 = 0;
                if (j == t+1 || j==t-1 || j==t+nosqr || j==t-nosqr){
                    for (IloInt k=0; k<vv.cnt1; k++){
                        if (j == vv.max_flow_set1[k]){
                            check2 = 1;
                        }
                                        }
                                }
                if (check1 ==0 && check2 == 1 ){
                    addcons1 += x[t-1][j-1];
                    valsx1[t-1][j-1] = 1;
                }
            }
            check3=0;
            if (prize[t]>0){
                for (IloInt k=0; k<vv.cnt1; k++){
                    if (nodesno+s == vv.max_flow_set1[k])
                        check3 =1;
                    }
                    if (check1 == 1 && check3 == 0) {
                        addcons1 += profx[l];
                        valsprofx1[l] = 1;
                    }
                    if (check1 == 0 && check3 == 1){
                        addcons1 += profx[l+1];
                        valsprofx1[l+1] = 1;
                    }
                    s++; l += 2;
                }
}
if (BACKCUTS){
        cnt_text = writefilereverse(g, valsx1, valsx01, prize, valsprofx1, cplex, ptr_text);
        cnt_tmp++;
        max_flow(&vv_rev, ptr_text, cnt_text);
        IloExpr addcons2(env);
        s = 0; l = 0;
        if (vv_rev.flow_val < valsy[g]){
            IloBool check1, check2;
            for (IloInt t=1; t<nodesno; t++){
                check1 = 0;
                for (IloInt k=0; k<vv_rev.cnt1; k++){
                    if (t == vv_rev.max_flow_set1[k]){
                    check1 = 1;
                    }
                                }
                for (IloInt j=1; j<nodesno; j++){
                    check2 = 1;
                    if (j == t+1 || j==t-1 || j==t+nosqr || j==t-nosqr){
                        for (IloInt k=0; k<vv_rev.cnt2; k++){
                            if (j == vv_rev.max_flow_set2[k]){
                            check2 = 0;
                            }
                                                }
                                        }
                                        if (check1 ==1 && check2 == 0 ){
                        addcons2 += x[t-1][j-1];
                        valsx1[t-1][j-1] = 1;
                                        }
                                }
                check3=1;
                if (prize[t]>0){
                    for (IloInt k=0; k<vv_rev.cnt2; k++){
                        if (nodesno+s == vv_rev.max_flow_set2[k])
                            check3 =0;
                        }
                        if (check1 == 0 && check3 == 1) {
                            addcons2 += profx[l];
                            valsprofx1 [l] = 1;
                        }
                        if (check1 == 1 && check3 == 0){
                            addcons2 += profx[l+1];
                            valsprofx1 [l+1] = 1;
                        }
                    s++; l += 2;
                                    }
                }
                model.add(addcons2 >= y[g]);
                cnt_cons++;
            }
        }
        model.add(addcons1 >= y[g]);
        cnt_cons++;
}}while(vv.flow_val < valsy[g] && vv_rev.flow_val < valsy[g]);
}}}while(nostop);
```

A.2 Branch-and-cut Algorithm

This part is to implement the branch-and-cut algorithm.

```
ILOCPLEXGOAL4(MyBranchGoal, IloNumVarArray, vars1, IloNumVarArray, vars2, NumVarMatrix, varsmatr, IloInt, nodesno) {
        IloNumArray x1, x2;
   NumMatrix   xmatr;
   IloNumArray objx0;
   NumMatrix   objx;
   IntegerFeasibilityArray feasy, feasx0;
   IntegerFeasibilityMatrix feasx;
   int nosqr = sqrt(nodesno-1.0);
   x1 = IloNumArray(getEnv());
   x2 = IloNumArray(getEnv());
   xmatr = NumMatrix(getEnv(),nodesno-1);
   objx0 = IloNumArray(getEnv());
   objx  = NumMatrix(getEnv(), nodesno-1);
   feasy = IntegerFeasibilityArray(getEnv());
   feasx0 = IntegerFeasibilityArray(getEnv());
   feasx  = IntegerFeasibilityMatrix(getEnv(),nodesno-1);
   getValues(x1, vars1);
   getValues(x2, vars2);
   for (IloInt n=0; n<nodesno-1; n++){
                IloNumArray xtmp;
           xtmp = IloNumArray(getEnv());
           for (IloInt j=0; j<nodesno-1; j++){
                   if (j==n+1 || j == n-1 || j == n+nosqr || j == n-nosqr){
                           xtmp.add(getValue(varsmatr[n][j]));
                   }
                   else xtmp.add(0);
           }
                xmatr[n] = IloNumArray(xtmp);
   }
   getObjCoefs(objx0, vars2);
   for (IloInt n=0; n<nodesno-1; n++){
           IloNumArray xtmp;
           xtmp = IloNumArray(getEnv());
           for (IloInt j=0; j<nodesno-1; j++){
                   if (j==n+1 || j == n-1 || j == n+nosqr || j == n-nosqr){
                           xtmp.add(getObjCoef(varsmatr[n][j]));
                   }
                   else xtmp.add(0);
           }
                objx[n] = IloNumArray(xtmp);
   }
   getFeasibilities(feasy, vars1);
   getFeasibilities(feasx0, vars2);
   for (IloInt n=0; n<nodesno-1; n++){
           IntegerFeasibilityArray xtmp;
           xtmp = IntegerFeasibilityArray(getEnv());
           for (IloInt j=0; j<nodesno-1; j++){
                   if (j==n+1 || j == n-1 || j == n+nosqr || j == n-nosqr){
                           xtmp.add(getFeasibility(varsmatr[n][j]));
                   }
                   else xtmp.add(Feasible);
           }
           feasx[n] = IntegerFeasibilityArray(xtmp);
   }
   IloInt bestj  = -1;
   IloInt bestn  = -1;
   IloNum maxinf = 0.0;
   IloNum maxobj = 0.0;
   IloCplex::Goal res;
   //branch variable y
   for (IloInt j = 0; j < nodesno; j++) {
      if ( feasy[j] == Infeasible ) {
         IloNum xj_inf = x1[j] - IloFloor (x1[j]);
         if ( xj_inf > 0.5 )
            xj_inf = 1.0 - xj_inf;
         if ( xj_inf > maxinf ) {
            bestj  = j;
            maxinf = xj_inf;
         }
      }
   }
   if ( bestj >= 0 ) {
      res = AndGoal(OrGoal(vars1[bestj] >= IloFloor(x1[bestj])+1,
            vars1[bestj] <= IloFloor(x1[bestj])),
            this);
   }
   else{
 // if all the variables of y are integral, branch x0
            IloInt cols2 = vars2.getSize();
        for (IloInt j = 0; j < cols2; j++) {
                        if ( feasx0[j] == Infeasible ) {
                              IloNum xj_inf = x2[j] - IloFloor (x2[j]);
                              if ( xj_inf > 0.5 )
                                    xj_inf = 1.0 - xj_inf;
                              if ( xj_inf >= maxinf                              &&
                                    (xj_inf > maxinf || IloAbs (objx0[j]) >= maxobj)  ) {
                                    bestj  = j;
                                    maxinf = xj_inf;
                                    maxobj = IloAbs (objx0[j]);
                              }
                        }
        }
                  if ( bestj >= 0 ) {
                         res = AndGoal(OrGoal(vars2[bestj] >= IloFloor(x2[bestj])+1,
                  vars2[bestj] <= IloFloor(x2[bestj])),
                  this);
        }
                  else{
//otherwise, branch x
                        IloInt cols3 = varsmatr.getSize();
                        for (IloInt j = 0; j < nodesno-1; j++) {
                              for (IloInt n = 0; n < nodesno-1; n++) {
                                    if ( feasx[j][n] == Infeasible ) {
                                          IloNum xj_inf = xmatr[j][n] - IloFloor (xmatr[j][n]);
                                    if ( xj_inf > 0.5 )
                                          xj_inf = 1.0 - xj_inf;
                                    if ( xj_inf >= maxinf                              &&
                                          (xj_inf > maxinf || IloAbs (objx[j][n]) >= maxobj)  ) {
                                          bestj  = j;
                                          bestn  = n;
                                          maxinf = xj_inf;
                                          maxobj = IloAbs (objx[j][n]);
                                          }
                                    }
                              }
                        if ( bestj >= 0 && bestn >= 0 ) {
                        res = AndGoal(OrGoal(varsmatr[bestj][bestn] >= IloFloor(xmatr[bestj][bestn])+1,
                        \varsmatr[bestj][bestn] <= IloFloor(xmatr[bestj][bestn])),this);
                              }
                        }
                  }
   }
   x1.end();
   x2.end();
   xmatr.end();
   objx0.end();
   objx.end();
   feasy.end();
   feasx0.end();
   feasx.end();
   return res;
}
```

Bibliography

[1] Y.P. Aneja. An integer linear programming approach to the steiner problem in graphs. *Networks*, 10(2):167–178, 1980. 47

[2] A. Archer, M.h. Bateni, M.T. Hajiaghayi, and H. Karloff. Improved approximation algorithms for prize-collecting steiner tree and tsp. *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 427–436. 8

[3] E. Balas. The prize collecting traveling salesman problem. *Networks*, 19(6):621–636, 1989. 6

[4] D. Bienstock, M.X. Goemans, D. Simchi-Levi, and D. Williamson. A note on the prize collecting traveling salesman problem. *Mathematical Programming*, 59(1):413–420, 1993. 6, 8

[5] J. Byrka, F. Grandoni, T. Rothvoß, and L. Sanità. An improved lp-based approximation for steiner tree. *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010*, pages 583–592, 2010. 6

[6] S.A. Canuto, M.G.C. Resende, and C.C. Ribeiro. Local search with perturbations for the prize-collecting steiner tree problem in graphs. *Networks*, 38(1):50–58, 2001. 3, 8

[7] M. Charikar, C. Chekuri, T. Cheung, Z. Dai, A. Goel, S. Guha, and M. Li. Approximation algorithms for directed steiner problems. *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 192–200, 1998. 6

[8] S. Chawla, D. Kitchin, U. Rajan, R. Ravi, and A. Sinha. Profit maximizing mechanisms for the extended multicasting game. *CMU CS Technical Report CMU-CS-02-164*, 2002. 10

[9] B.V. Cherkassky and A.V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19(4):390–410, 1997. 35

[10] M. Chlebik and J. Chlebikova. Approximation hardness of the steiner tree problem on graphs. *Algorithm Theory, SWAT 2002*, pages 95–99, 2002. 6

[11] S. Chopra, E.R. Gorres, and M.R. Rao. Solving the steiner tree problem on a graph using branch and cut. *ORSA Journal on Computing*, 4(3):320–335, 1992. 6, 11

[12] S. Chopra and M.R. Rao. The steiner tree problem i: formulations, compositions and extension of facets. *Mathematical Programming*, 64(1):209–229, 1994. 11

[13] N. Christofides. Worst-case analysis of a new heuristic for the travelling salesman problem. *Technical Report 388-Graduate School of Industrial Administration, Carnegie Mellon University*, 1976. 8

[14] R. Cole, R. Hariharan, M. Lewenstein, and E. Porat. A faster implementation of the goemans-williamson clustering algorithm. *Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 17–25, 2001. 8

[15] R. Cordone and M. Trubian. A relax-and-cut algorithm for the knapsack node weighted steiner tree problem. *Asia-Pacific Journal of Operational Research*, 25(3):373–391, 2008. 7, 9

[16] A.S. Cunha, A. Lucena, N. Maculan, and M.G.C. Resende. A relax-and-cut algorithm for the prize-collecting steiner problem in graphs. *Discrete Applied Mathematics*, 157(6):1198–1217, 2009. 9

[17] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 2(4):393–410, 1954.

[18] C.W. Duin and A. Volgenant. Some generalizations of the steiner problem in graphs. *Networks*, 17(3):353–364, 1987. 7, 9, 10

[19] S. Engevall, M. Göthe-Lundgren, and P. Värbrand. A strong lower bound for the node weighted steiner tree problem. *Networks*, 31(1):11–17, 1998. 9

[20] J. Feigenbaum, C.H. Papadimitriou, and S. Shenker. Sharing the cost of multicast transmissions. *Journal of Computer and System Sciences*, 63(1):21–41, 2001. 3

[21] P. Feofiloff, C.G. Fernandes, C.E. Ferreira, and J.C. de Pina. Primal-dual approximation algorithms for the prize-collecting steiner tree problem. *Information Processing Letters*, 103(5):195–202, 2007. 8

[22] M.R. Garey, R.L. Graham, and D.S. Johnson. The complexity of computing steiner minimal trees. *SIAM Journal on Applied Mathematics*, pages 835–859, 1977. 6

[23] M.X. Goemans and D.P. Williamson. A general approximation technique for constrained forest problems. *Mathematical Programming*, 59(1):413–420, 1993. 3, 6

[24] M.X. Goemans and D.P. Williamson. The primal-dual method for approximation algorithms and its application to network design problems. *Approximation Algorithms for NP-hard Problems*, 4(60):144–191, 1996. 3, 8

[25] S. Guha and S. Khuller. Improved methods for approximating node weighted steiner trees and connected dominating sets. *Foundations of Software Technology and Theoretical Computer Science*, pages 1056–1056, 1998. 8

[26] M.T. Hajiaghayi and K. Jain. The prize-collecting generalized steiner tree problem via a new approach of primal-dual schema. *Proceedings of the 17th Annual ACM-SIAM Symposium on Discrete Algorithm, SODA 2006*, pages 631–640, 2006. 7

[27] D.S. Johnson, M. Minkoff, and S. Phillips. The prize collecting steiner tree problem: theory and practice. *Proceedings of the 11th Annual ACM-SIAM Symposium On Discrete Algorithms*, pages 760–769, 2000. 3, 8

[28] G.W. Klau, I. Ljubić, A. Moser, P. Mutzel, P. Neuner, U. Pferschy, G. Raidl, and R. Weiskircher. Combining a memetic algorithm with integer programming to solve the prize-collecting steiner tree problem. *Genetic and Evolutionary Computation, GECCO 2004*, pages 1304–1315, 2004. 8

[29] P. Klein and R. Ravi. A nearly best-possible approximation algorithm for node-weighted steiner trees. *Journal of Algorithms*, 19(1):104–115, 1995. 8

[30] T. Koch and A. Martin. Solving steiner tree problems in graphs to optimality. *Networks*, 32(3):207–232, 1998. 16

[31] I. Ljubić, R. Weiskircher, U. Pferschy, G. Klau, P. Mutzel, and M. Fischetti. Solving the prize-collecting steiner tree problem to optimality. *Technical Report TR-186-1-04-01, Vienna University of Technology*, 2004. iii, 3, 4, 9, 14, 16, 20, 34, 35, 41, 42, 57

[32] A. Lucena and J.E. Beasley. A branch and cut algorithm for the steiner problem in graphs. *Networks*, 31(1):39–59, 1998. 6

[33] A. Lucena and M.G.C. Resende. Strong lower bounds for the prize collecting steiner problem in graphs. *Discrete Applied Mathematics*, 141(1-3):277–294, 2004. 3, 4, 9

[34] J.E. Mitchell. Branch-and-cut algorithms for combinatorial optimization problems. *Handbook of Applied Optimization*, pages 65–77, 1999. 32

[35] A.B. Philpott and N.C. Wormald. On the optimal extraction of ore from an open-cast mine. *Technical Report - University of Auckland, New Zealand*, 1997. 3, 4, 20, 22, 26, 27, 28, 29

[36] S.K. Rao, P. Sadayappan, F.K. Hwang, and P.W. Shor. The rectilinear steiner arborescence problem. *Algorithmica*, 7(1):277–288, 1992. 14

[37] G. Robins and A. Zelikovsky. Improved steiner tree approximation in graphs. *Proceedings of the 11th Annual ACM-SIAM Symposium On Discrete Algorithms, SODA 2010*, pages 770–779, 2000. 6

[38] A. Segev. The node-weighted steiner tree problem. *Networks*, 17(1):1–17, 1987. 7, 8, 9

[39] D.A. Thomas, M. Brazil1, D.H. Lee, and N.C. Wormald. Decline design in underground mines using constrained path optimisation. *International Transactions in Operational Research*, 14(2):143–158, 2007. 1

[40] R.T. Wong. A dual ascent approach for steiner tree problems on a directed graph. *Mathematical Programming*, 28(3):271–287, 1984. 47