Operations Research

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Performance Analysis and Best Implementations of Old and New Algorithms for the Open-Pit Mining Problem

Dorit S. Hochbaum, Anna Chen,

To cite this article:
Dorit S. Hochbaum, Anna Chen, (2000) Performance Analysis and Best Implementations of Old and New Algorithms for the Open-Pit Mining Problem. Operations Research 48(6):894-914. https://doi.org/10.1287/opre.48.6.894.12392

PERFORMANCE ANALYSIS AND BEST IMPLEMENTATIONS OF OLD AND NEW ALGORITHMS FOR THE OPEN-PIT MINING PROBLEM

DORIT S. HOCHBAUM

Department of IE & OR and Haas School of Business, University of California, Berkeley, California 94720, hochbaum@ieor.berkeley.edu

ANNA CHEN

Department of IE & OR, University of California, Berkeley, California 94720

(Received August 1996; revisions received July 1997, April 1998; accepted June 1999)

The open-pit mining problem is to determine the contours of a mine, based on economic data and engineering feasibility requirements, to yield maximum possible net income. This practical problem needs to be solved for very large data sets. In practice, moreover, it is necessary to test multiple scenarios, taking into account a variety of realizations of geological predictions and forecasts of ore value.

The industry is experiencing computational difficulties in solving the problem. Yet, the problem is known to be equivalent to the minimum cut or maximum flow problem. For the maximum flow problem there are a number of very efficient algorithms that have been developed over the last decade. On the other hand, the algorithm that is most commonly used by the mining industry has been devised by Lerchs and Grossmann (LG). This algorithm is used in most commercial software packages for open-pit mining.

This paper describes a detailed study of the LG algorithm as compared to the maximum flow "push-relabel" algorithm. We propose here an efficient implementation of the push-relabel algorithm adapted to the features of the open-pit mining problem. We study issues such as the properties of the mine and ore distribution and how they affect the running time of the algorithm. We also study some features that improve the performance of the LG algorithm. The proposed implementations offer significant improvements compared to existing algorithms and make the related sensitivity analysis problem practically solvable.

1. INTRODUCTION

A basic problem faced by the mining industry is the open-pit contour determination. Open-pit mining is a surface mining operation in which blocks of earth are extracted from the surface to retrieve the ore contained in them. During the mining process, the surface of the land is being continuously excavated, and an increasingly deeper pit is formed until the operation terminates. The optimal contour of this mine pit has to be determined prior to the onset of mining operations. To design an optimal pit the entire volume is subdivided into blocks, and the value of the ore in each block is estimated by using geological information obtained from drill cores. Each block has a weight associated with it representing the value of its ore less the cost of excavating the block. There are constraints that specify the slope requirements of the pit and precedence constraints preventing blocks from being mined before others on top of them. Subject to these constraints, the objective is to mine the set of blocks that provide the maximum net benefits.

The propagation and increased use of computational tools in the mining industry opened up possibilities for better informed decision making. The current standards of planning require the exploration of numerous scenarios and their effect on all aspects of the operations. A typical scenario analysis calls for varying market prices for the commodity/metal to be mined and its affect on the pit design, and thus on the production plan. This type of planning for different scenarios is called *sensitivity analysis*. With large-scale mines and the arising need for sensitivity analysis, the running time of algorithms have become a limiting factor.

The most commonly used algorithm by the industry and commercial software has been devised by Lerchs and Grossmann (1965). That algorithm, which we refer to as the *LG algorithm*, solves optimally the open-pit mining problem. While the industry has been using primarily the LG algorithm, the open-pit mining problem can also be solved via any maximum flow algorithms. Maximum flow algorithms that are exceptionally efficient have been developed over the last decade. Yet, the majority of the approaches used by mining practitioners for solving the problem consist of LG algorithm combined with some variants or heuristics.

The open-pit mining problem can be formally represented as a graph problem defined on a directed graph $G=(V,A)$. Each block corresponds to a node in V with a weight b_i representing the net value of the individual block. That value is the difference between the ore value in the specific block and the cost of excavating that specific block. This weight can be either positive or negative. There is a directed arc in A from node i to node j if block i cannot be extracted before block j. (Note that it suffices to include only arcs between blocks and their immediate successors.) Thus to decide which blocks to extract to maximize profit is equivalent to finding a maximum weight set of nodes in the graph such that all successors of the nodes in the set are included in

Subject classifications: Industries/mining/metals: planning for mining operations. Networks/graphs, applications: application of minimum cut to mining. Networks/graphs, flow algorithms: implementing the preflow and another minimum cut algorithm.
Area of review: ENVIRONMENT, ENERGY, AND NATURAL RESOURCES.

the set. Such a set is called a maximum *closure* of G. The open-pit mining problem is also known in the mining industry as the *ultimate pit problem*. Formally:

Problem Name: *Open-Pit Mining*;

Instance: *Given a directed graph* $G=(V,A)$, *and node weights* (*positive or negative*) b_i *for all* $i \in V$;

Optimization Problem: *find a closed subset of nodes* $V' \subseteq V$ *such that* $\Sigma_{i \in V'} b_i$ *is maximum.*

We refer to the node weighted graph $G=(V,A)$ as the *open-pit graph*. The collection of blocks and their weights is known as the *economic block model*. Determining the precedence constraints, or equivalently the arcs in the graph, is equivalent to imposing *slope requirements*. The choice of specific precedence pattern (or edge pattern) representing slope requirements is usually proprietary to each mining company, and depends on engineering feasibility concerns.

The open-pit mining problem is known in the graph theory literature as the *maximum closure problem*. The maximum closure problem (and therefore also the open pit mining problem) is known to be solvable by the maximum flow that is constructed on a certain related network, as shown by Picard (1976). This makes the entire repertoire of maximum flow algorithm available to solve the open-pit mining problem. Among maximum flow algorithm the current fastest implementations are based on an algorithm called *push-relabel* algorithm.

In another paper, Hochbaum (1996), we provide a detailed analysis of the theoretical complexity of the LG algorithm. This analysis shows that the original algorithm is not polynomial and has substantially higher complexity than the push-relabel and other polynomial maximum flow algorithms. Complexity, however, is a worst-case concept, which means that it is not impossible for LG algorithm to run, in practice, faster than push-relabel. A famous example of such phenomenon is the (exponential) simplex method, which runs for practical problems faster than the polynomial ellipsoid method. Because complexity alone is not a full indication on the usefulness of an algorithm, it is necessary also to examine the practical comparative performance of algorithms in order to come up with recommendations for practical use.

We report here on an extensive empirical study, the purpose of which is to determine the *practical* performance of the Lerchs-Grossmann algorithm that is widely used in the industry, and compare it to the push-relabel algorithm for maximum flow. We evaluate here practical issues of preprocessing, parametric and sensitivity analysis, choice of edge pattern to represent a given mine, and how features of the mine data affect the running times of the different algorithms. This study highlights the advantages and deficiencies of each approach. We use the study to select among all possible implementations of the push-relabel algorithm the one that works best for the problem and study the drawbacks and advantages of the LG algorithm. This study serves also to promote better understanding of the push-relabel algorithm and its performance when used in large-scale graphs. The graphs we use in this study are of sizes that vary between 54,000 nodes to 400,000 nodes and up to 0.7 billion arcs.

We showed in Hochbaum (1996) a parametric version of the LG algorithm that works without an increase in complexity compared to a single run. We demonstrate here how to use this algorithm and the efficient parametric push-relabel algorithm of Gallo et al. (1989) that was devised for the sensitivity analysis of the maximum flow problem. We analyze preprocessing—a technique that was recommended in the literature—and study empirically its effectiveness (§8).

The experiments reported here were all using a 25-Mhz. Next computer with 28 MB of memory. For the real mine data we used another computer as reported in §10. The running times reported do not include input and output times.

The reader may want to access a WWW page for graphical interactive demonstration of the dependence of the optimal pit on the distribution and value of ore in the simpler two-dimensional case (http://www.ieor.berkeley.edu/∼hochbaum).

The paper is organized as follows. We start with a review of the literature on computational studies of algorithms for the open-pit mining problem, preprocessing, and the literature on scheduling and sensitivity analysis. Technical preliminaries are introduced in §3 that include the reduction of the open-pit mining problem to minimum cut problem and introduction of the important features of push-relabel and the LG algorithms. In §4 we describe how the data for our experiments were generated. In §5 we report on the experiments that compare various push-relabel implementations and compare push-relabel to LG. In §6 we investigate the performance of LG and find critical parameters of the input that adversely affect the performance of the LG algorithm. In §7 we study how changes in various mine features in the data affect the push-relabel algorithm and LG. Preprocessing and its validity are reviewed and proved in §8. Empirical results on the performance of preprocessing are reported as well. In §9 we present a comparison of a parametric implementation of both push-relabel and LG algorithms to their corresponding versions in which the runs are repeated for each value of the parameter—a practice commonly used. Finally, in §10 we present the results of runs for real data.

2. LITERATURE REVIEW

A number of algorithms for solving the basic mining problem have been developed over the years. These algorithms include heuristic approaches that are not guaranteed to deliver optimal solutions as well as optimization approaches. Most of the optimization approaches are based on the LG algorithm. These are, however, few that employ a transportation problem based algorithms, or maximum flow algorithms. Each of these approaches will be discussed below.

2.1. Maximum Flow Algorithms

Because the basic mining problem is equivalent to the maximum closure problem (Picard 1976), algorithms that solve maximum flow could be used for the mining problem. There is a vast literature on the maximum flow problem and very efficient versions have been developed recently. Yet, while the LG algorithm was extensively studied in the mining literature, there was no work on other maximum flow algorithms for the open-pit-mining problem until the early 1990s.

Yegulalp and Arias (1992) and Yegulalp et al. (1993) conducted computational experiments to compare the performance of a variant of the push-relabel maximum flow algorithm, the *excess-scaling algorithm* of Ahuja and Orlin (1989), and the LG algorithm. Yegulalp and Arias showed that Whittle Programming Pty. Ltd.'s implementation of the LG algorithm has faster computation times than their own implementation of the excess-scaling algorithm. Their experiments showed that in the majority of 11 runs, the LG algorithm is approximately twice as fast as the excess-scaling algorithm. In a later paper Yegulalp et al. showed that the opposite holds. It is, however, impossible to assess the implication of the computational experiments reported in Yegulalp et al. because they bound the distance label of the nodes by 12; whereas the technical requirement for correctness is to allow distance labels to go up to the number of nodes or more.

Giannini et al. (1991) developed the software package PITOPTIM, which uses Dinic's maximum flow algorithm for computing the optimal pit contour. Computation times for their implementation were given without comparisons with other algorithms.

Huttagosol and Cameron (1992) formulated the open-pit mining problem as a *transportation problem*. The objective of the transportation problem is to assign shipping quantities from a set of suppliers (sources) to a set of consumers (destinations) so that all demands are met, supplies are not exceeded, and the total shipping cost is minimized. This is effectively the bipartite reduction described earlier. Huttagosol and Cameron proposed using the network simplex method for solving the problem. Their computational experiments show that the network simplex is slower than the LG algorithm.

2.2. Preprocessing

Preprocessing was introduced by Barnes and Johnson (1982). The idea is to identify blocks that easily can be eliminated from any optimal pit thus reducing the problem size. For preprocessing to be worthwhile, the preprocessing algorithm has to be effective in reducing the problem size and efficient in terms of computation time so that applying it prior to the open pit (or ultimate pit limit) algorithm will result in a reduction in the overall computation time.

Giannini et al. (1991) conducted computational experiments and showed that preprocessing the input data can lead to a substantial reduction in the number of blocks in the economic block model and in the computation time required to compute the optimal pit contour. The preprocessing algorithm that they used is presented by Caccetta and Giannini (1985). We describe the preprocessing approach and its validity in §8.

2.3. Heuristic Algorithms

Up till the early 1980s, heuristic algorithms were widely used in the mining industry because they execute faster and are conceptually simpler than optimizing algorithms. As Whittle (1989) pointed out, the difference in value between an open-pit design based on an optimal pit and one based on a pit obtained from a heuristic algorithm can be several percent, representing millions of dollars.

Most heuristic algorithms proceed by identifying ore blocks and then checking if one or more of these blocks together have a combined weight that is higher than the total cost of excavating their overlying blocks.

A terminology frequently used in the mining industry is that of *cone* and *base*. Given a block b and a safe slope requirement, the set of blocks that must be excavated before b form a *cone* with b at its *base*. A cone is a block and all its successors. The moving cone method (described, e.g., in Pana 1965) is to search for cones in which the total weight of all the blocks in the cone is positive. These cones are added to the already generated pit. It is easy to devise an example where no positive weight cone exists while still there is an optimal solution of arbitrarily high value, thus demonstrating that this algorithm is not an optimizing algorithm. The heuristic idea lies in the assumption that every cone in the optimal pit is profitable, whereas in fact an optimal pit may consist of a collection of cones none of which alone is of positive value, but together the cones share negative value blocks and have total weight which is positive.

Robinson and Prenn's (1977) algorithm checks each cone that has an ore block as its base. If the total weight of all the nodes in the cone is positive, then the nodes are removed from the graph. All the removed nodes together form the final pit. As noted above, such an algorithm may deliver a nonoptimal solution.

Koborov's algorithm (1974) attempts to improve on the moving cone method by checking cones with an ore block as its base block. The idea is that an ore block generates profit that pays for the cost of removing its overlying waste blocks those of negative weight. So the algorithm "charges" an ore block for its overlying waste blocks. This is done by decreasing the ore block's weight and increasing the waste block's weight by the same amount. All cones that have a base block with a positive weight at the end of the algorithm are included in the final pit.

Dowd and Onur (1992) developed a modified version of Koborov's algorithm that is claimed to find an optimal pit. Their computational experiments show that it is faster than their implementation of the LG algorithm.

Other heuristic algorithms include the dynamic programming methods of Johnson and Sharp (1971) and Koenigsberg (1982). A discussion of additional heuristic algorithms is given by Kim (1978) and Koenigsberg.

2.4. Optimizing Algorithms

One variant of the LG algorithm was developed by Zhao and Kim (1992). It differs from the LG algorithm in the way that the blocks are regrouped after it has been discovered that a block in a profitable set lies beneath a block in a unprofitable set. The complexity of Zhao and Kim's algorithm has never been analyzed, and there is no indication that this algorithm's method of regrouping the blocks is more efficient than the LG algorithm. No direct computational comparison between the two algorithms was provided in the paper.

Vallet (1976) proposed an interesting variant of the LG algorithm. Rather than partitioning the nodes into strong and weak (which are subsets of nodes of total positive weight and negative weight, respectively, see §3.4 for definitions), Vallet classifies the sets based on the average weight—the ratio B/V, where B is the total weight and V the volume of the set. He refers to this ratio as *strength*. The ratio classification does not affect the status of subsets as strong or weak; it affects only the order of processing the merger arcs.

2.5. Sensitivity Analysis and Scheduling Algorithms Literature

Parametric (or sensitivity) analysis is often called for in the process of planning in the mining industry. Typical parameters include the unit sale price of the processed ore, or the unit cost per processing capacity. Parametric and sensitivity analysis have been of major concern and importance for the industry. The parametric implementations reported to date have running times that are considerably greater than that required to solve for a single scenario. In Hochbaum (1996) we report a theoretically efficient parametric implementation of the LG algorithm that has the same complexity as a single-value run of the LG algorithm. Here we report on the practical performance of this implementation. Parametric algorithms for maximum flow were never previously implemented and tested for the open-pit mining problem to the best of our knowledge.

Comprehensive planning is crucially dependent on the ability to conduct sensitivity analysis to explore various scenarios and their effect on aspects of the operations. A common practice in the mining industry is to perform sensitivity analysis by repeated applications of an algorithm that computes a single optimal pit. For instance, a sensitivity analysis is required for variations in b_i, the weight of the ith block. This is typically computed by the following formula:

$$b_i = extract(i) \cdot recovery \cdot price - ore(i) \cdot proc\_cost - volume(i) \cdot mine\_cost, \quad (1)$$

where $extract(i)$ is the weight (in tons) of extract contained in block i, *recovery* is the recovery rate, *price* is the commodity price (per ton) of the extract, $ore(i)$ is the amount of ore contained in block i in tons, *proc_cost* is the cost of processing a ton of ore, $weight(i)$ is the weight of block i in tons (which is typically constant), and *mine_cost* is the cost of mining a ton of rock.

When performing sensitivity analysis, one of the elements in the block value formula, λ, is identified as the parameter of interest, and its effect on the optimal pit is analyzed. The formula can then be expressed as $b_i(\lambda) = c_i + \lambda d_i$, where c_i represents the terms that are independent of λ and d_i represents the terms that are linearly dependent on λ. The problem should then be repeatedly solved and evaluated for a range of values of λ.

An important observation is that for a monotone sequence of parameter values, the sequence of generated pits is *nested*. This means that a pit evaluated for a value of λ that has higher benefits per block, contains the pit that is evaluated for a lower value of λ.

A substantial body of literature was devoted to issues of scheduling for the open-pit mining problem. Because of computational difficulties, scheduling is usually done by employing various methods to generate *nested pits*. This use of nested pits links the scheduling decisions to sensitivity analysis.

Based on the idea that mining the most valuable ore rock as early as possible would maximize the net present value (NPV), several algorithms were devised to generate a series of nested pits $P_1 \subseteq P_2 \subseteq \cdots \subseteq P_\ell$ such that $B_1/V_1 > B_2/V_2 > \cdots > B_\ell/V_\ell$, where B_1 denotes the total benefit of all the blocks in pit P_1, V_1 denotes the volume of pit P_1, and B_i and V_i denote the total benefit and volume of pit $(P_i - P_{i-1})$, $i = 2, \ldots, \ell$, respectively. In other words, pit P_1 has the highest benefit-to-volume ratio while P_ℓ has the lowest benefit-to-volume ratio.

DEFINITION 1. The computation of a series of nested pits $P_1, P_2, \ldots, P_\ell$ such that $B_1/V_1 > B_2/V_2 > \cdots > B_\ell/V_\ell$, is called *parameterization*.

Vallet (1976) developed algorithms that generate the series of nested pits by searching, at each stage, for the pit with the highest benefit-to-volume ratio among all the feasible pits in the graph. Having found this pit, the current stage is complete, and the pit is removed from the graph before the next stage is begun. (This approach is effectively the same as the repeated applications for parameterization of Lerch and Grossman.)

Taking a different approach, Dagdelen and Francois-Bongarcon (1982) and Francois-Bongarcon and Guibal (1982, 1984), proposed to replace the economic parameters by ore content and recoverable metal quantity. They presented an algorithm that generates a series of nested pits $P_1, P_2, \ldots, P_\ell$ such that $Q_1/V_1 > Q_2/V_2 > \cdots > Q_\ell/V_\ell$, where Q_1 and V_1 denote the total metal content and volume of pit P_1, and Q_i and V_i denote the total metal content and volume of pit $(P_i - P_{i-1})$, $i = 2, \ldots, \ell$, respectively. Note that while benefit is any real number, the metal content is always nonnegative.

DEFINITION 2. The computation of a series of nested pits $P_1, P_2, \ldots, P_\ell$ such that $Q_1/V_1 > Q_2/V_2 > \cdots > Q_\ell/V_\ell$, is called *reserve parameterization*.

The rationale for either parameterization is that mining the most valuable ore rock as early as possible would maximize the net present value (NPV). The generation of the series of nested pits is done using heuristic approaches.

Francois-Bongarcon and Guibal's algorithm is implemented in the open-pit software package MULTIPIT. Coleou (1989) states that MULTIPIT is faster in generating a series of pits than repeatedly applying the LG algorithm. Coleou also reviews the methodologies and applications of reserve parameterization.

Recall that the purpose of using a series of nested pits in mine sequencing is to reduce the complexity caused by problem size. However, if the increment in the number of blocks from pit P_i to pit P_{i+1} in the series is large, then the problem size would still be large. To minimize this difficulty, Wang and Sevim (1993) developed a heuristic algorithm for generating a series of pits such that the number of blocks between two consecutive pits is not greater than a predefined value. This problem is NP-hard: The problem of finding a set of blocks to excavate that does not include more than a prescribed number of blocks is solvable by any algorithm that solves minimum cut in a closure graph with bounded source set, which is a known NP-hard problem. (Garey and Johnson 1979 problem [ND17] refers to cuts with bounded sets in general graphs. We have a proof that applies directly to closure graphs, by reducing the maximum clique problem to the bounded maximum closure problem. The details are omitted.)

Wang and Sevim's algorithm identifies the sets of blocks that can be eliminated without violating the contour constraints and computes their metal quantities. Among these blocks, the set containing the predefined number of blocks k that has the least metal quantity is removed from the set of K candidate blocks. The remaining blocks form the pit of size $K-k$ that has heuristically high metal quantity among all feasible pits of size $K-k$. This process is repeated to compute the rest of the pits in the series, where at iteration i the pit of size $K-i\cdot k$ that is likely to have high metal quantity is found.

2.6. Comprehensive Modeling

A comprehensive model of the mining operations scheduling has seldom been addressed. One such effort originated with Johnson (1968), who has formulated the entire mine planning problem including scheduling and capacity issues, as a linear programming problem. This formulation generates a large number of constraints and variables and is very difficult to solve. For example, a problem with 50,000 blocks (which is considered in practice to be a small problem) and 5 scheduling periods can have as many as 500,000 variables and at least as many constraints (Dagdelen and Johnson 1986). Moreover, the linear programming formulation does not take into account the 0-1 integer requirement of some of the variables and thus, if solved, delivers only a fractional solution.

3. PRELIMINARIES

3.1. Notation

A directed graph on a set of nodes V and a set of arcs A is denoted by $G=(V,A)$. The number of arcs $|A|=m$ and the number of nodes $|V|=n$. The capacity bound on arc $(u,v)\in A$ is $c(u,v)$. For $P,Q\subset V$, the set of arcs going from P to Q is denoted by, $(P,Q)=\{(u,v)\in A \mid u\in P \text{ and } v\in Q\}$. For $P,Q\subset V$, the capacity of the cut separating P from Q is $C(P,Q)=\Sigma_{(u,v)\in(P,Q)}\, c(u,v)$.

3.2. The Open-Pit Mining Problem as a Minimum Cut Problem

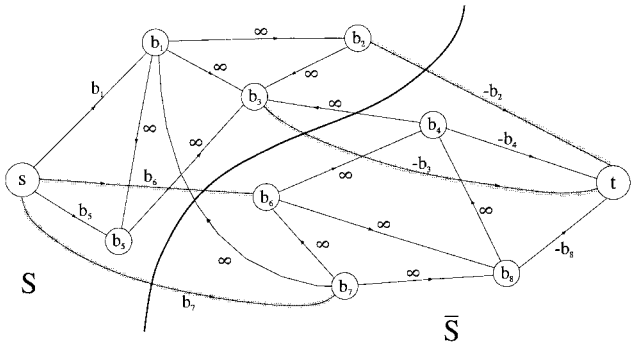
In formulating the open-pit mining problem, or the equivalent maximum closure problem, each block is represented by a node in a graph, and the slope requirements are represented by precedence relationship described by the arcs of A in the graph. Let x_j be a binary variable that is 1 if node j is in the closure, and 0 otherwise. b_j is the weight of the node or the net benefit derived from the corresponding block.

$$\begin{aligned} \text{Max} \quad & \sum_{j\in V} b_j\cdot x_j \\ \text{subject to} \quad & x_j - x_i \geqslant 0 \quad \forall (i,j)\in A \\ & 0\leqslant x_j \leqslant 1 \text{ integer}, \ j\in V. \end{aligned}$$

Each row of the constraint matrix has one 1 and one -1, a structure indicating that the matrix is totally unimodular. More specifically, it is indicating that the problem is a dual of a flow problem—a minimum cut problem. Johnson (1968) seems to have been the first researcher who recognized the relationship between the open-pit mining problem and the maximum flow problem. He did this by reducing the problem to another closure problem on bipartite graphs. This *bipartition reduction* involves placing an arc between two nodes of positive and negative weight if and only if there is a directed path leading from the positive weight node to the negative weight node. The latter problem, frequently referred to as the *selection problem*, he observed, can be solved by the maximum flow problem. That bipartite problem was independently shown to be solved by a minimum cut algorithm by Rhys (1970) and Balinski (1970).

Picard (1976) demonstrated directly that a minimum cut algorithm on a related graph solves the maximum closure problem and thus the open-pit mining problem. The related graph $\tilde{G}=(\tilde{V},\tilde{A})$ is constructed by adding a source and a sink node, s and t to the set of nodes V to create $\tilde{V}=V\cup\{s,t\}$. Let $V^+=\{j\in V \mid b_j>0\}$, and $V^-=\{j\in V \mid b_j<0\}$. The set of arcs in the related graph, $\tilde{A}$, is the set A appended by arcs $\{(s,v) \mid v\in V^+\}\cup\{(v,t) \mid v\in V^-\}$. The capacity of all arcs in A is set to ∞, and the capacity of all arcs adjacent to source or sink is $|b_v|$: That is, $c(s,v)=b_v$ for $v\in V^+$ and $c(v,t)=-b_v$ for $v\in V^-$. In the related graph $\tilde{G}$ the source set of a minimum cut separating s and t is also a maximum closure in the graph. The proof of this fact is repeated here for completeness: The source set containing s is obviously closed as the cut must be finite

Figure 1. Maximum closure, $S - \{s\}$, in the related graph.



and thus cannot include any arcs of A. So there is no arc of A going from the source set to the sink set.

Now let a finite cut be $(S, \bar{S})$ in the graph $\tilde{G} = (\tilde{V}, \tilde{A})$. The capacity of the cut $C(S, \bar{S})$ is

$$\sum_{j \in V^- \cap S} |b_j| + \sum_{j \in V^+ \cap \bar{S}} b_j = \sum_{j \in V^- \cap S} |b_j| + \sum_{j \in V^+} b_j - \sum_{j \in V^+ \cap S} b_j$$

$$= B - \sum_{j \in S} b_j,$$

where B is fixed—the sum of all positive weights. Hence minimizing the cut capacity is equivalent to maximizing the total sum of weights of nodes in the source set S of the cut. It thus follows that the source set of a minimum cut is also a closed set of nodes of maximum weight.

Figure 1 illustrates the correspondence between a minimum cut $(S, \bar{S})$ (the thick arcs) and the maximum closed set—the source set of the cut S.

3.3. The Push-Relabel Algorithm

The push-relabel algorithm is an algorithm solving the maximum flow and thus also the minimum cut problem. (Ford and Fulkerson 1957 established that maximum flow is equal to minimum cut.) The algorithm was first proposed in a generic form by A. Goldberg (1987). Goldberg and Tarjan (1988) described an implementation of the algorithm using dynamic trees that has particularly efficient running time. With the notation $m = |A|$ and $n = |V|$ the complexity of Goldberg and Tarjan's algorithm is $O(nm \log n^2/m)$. We now sketch the algorithm and its important properties that are relevant to our study.

DEFINITION 3. A *preflow* f is a function on the arcs that satisfies the capacity constraint and the *relaxed flow balance constraint* $\Sigma_{w \in V} f(w, v) \geq \Sigma_{w \in V} f(v, w), \ \forall v \in V - \{s, t\}$.

For preflow f let $G_f = (V, A_f)$ be the *residual graph*: For arc $(u, v) \in A$, the arc $(u, v) \in A_f$ if $f(u, v) < c(u, v)$. The residual capacity of the arc is $c_f(u, v) = c(u, v) - f(u, v)$. For $(u, v) \in A$ the arc $(v, u) \in A_f$ if $f(u, v) > 0$. The residual capacity of (v, u) is $c_f(v, u) = f(u, v)$.

DEFINITION 4. The *excess* $e_f(v)$ of a node v with respect to flow f is $e_f(v) = \Sigma_{w \in V} f(w, v) - \Sigma_{w \in V} f(v, w)$.

DEFINITION 5. A vertex $v \notin \{s, t\}$ is *active* if $e_f(v) > 0$.

DEFINITION 6. The *distance labeling* d is a function that assigns the vertices to the non-negative integers such that $d(t) = 0$, $d(s) = n$, and $d(v) \leq d(w) + 1$ for every residual arc $(v, w) \in A_f$.

DEFINITION 7. An arc (u, v) is called *admissible* if $(u, v) \in A_f$ and $d(u) > d(v)$.

THE GENERIC PUSH-RELABEL ALGORITHM.

```
begin /* Initialization */
  d(s) = n, d(i) = 0, ∀i ≠ s
  ∀(s, j) ∈ A f(s, j) = c(s, j)
  e_f(j) = c(s, j)
  while network contains an active node do
  select an active node v
    begin /* Push step */
      if ∃ admissible arcs (v, w)
      then
        Let δ = min_{(v,w)∈A_f} {e_f(v), c_f(v, w)},
          where (v, w) is admissible
        Send δ units of flow from v to w.
      else /* Relabel step (all (v, w) are inadmissible) */
        set d(v) = min_{(v,w)∈A_f} {d(w) + 1}
    end
end
```

An active node is processed by applying to it the push and relabel steps. Removing a node from the list of active nodes is called a *discharge step*.

The push-relabel algorithm can be viewed as a two-phase algorithm. In the first phase, the preflow initialized at the source is *pushed* to the sink. In the second phase, the excess that cannot get to the sink is returned to the source. The end of the first phase, phase I, occurs when the labels of all active nodes are $\geq n$.

The push-relabel algorithm has several nice properties: the label of each node never exceeds $2n + 1$; the distance label, when $< n$, is a lower bound on the distance of the node to the sink; when all active nodes have labels $\geq n$ then the nodes with labels $\geq n$ form the source set of a minimum cut. This latter property can be used when we are interested only in solving the minimum cut problem, which is the case here. We can terminate the algorithm after the first phase without carrying the algorithm to completion. In the implementations of the push-relabel algorithm we indeed run phase I only.

The algorithm can be implemented with different strategies. The **select** step of next active node can be implemented in a FIFO (First In First Out) order, or LIFO (Last In First

Out) order. The **Relabel** step increases the distance label of nodes. The speed of increase of the labels is an important factor determining the running time of the algorithm as the algorithm must terminate by the time all labels exceed $2n$. In practice, two heuristics were found to improve the performance of the push-relabel algorithm by speeding up the process of labels' increase. They are the *global-relabeling* and the *gap-relabeling* heuristics. Additional details on these relabeling heuristics are given next.

3.3.1. Global-Relabeling Heuristic. The global-relabeling heuristic computes the exact distance labels for all the nodes by performing a breadth-first search (BFS) from the sink t. This heuristic is useful because of the "local" nature of the relabeling step: In the relabeling step of the push-relabel algorithm, only one node is relabeled at a time. The distance labels of all the other nodes remain unchanged, including every node whose current path to the sink passes through the relabeled node. After a number of relabeling steps, many of the nodes may have distance labels whose values are much lower than their actual distances to the sink. A large number of relabeling steps would then have to be performed to increase the distance label of each of these nodes to the actual length of the shortest path from the node to the sink. These relabeling steps can be eliminated by performing a global relabeling, which immediately resets the distance labels of all the nodes to their actual distances to the sink.

The origin of global relabeling heuristic is not clear. It was mentioned in Goldberg's dissertation in 1987 and was also used by Grigoriadis about the same time.

Indeed, it appears that global relabeling improves the runtime performance of the push-relabel algorithm: Anderson and Setubal (1993), Nguyen and Venkateswaran (1993), and Badics and Boros (1993) all reported the importance of including a global relabeling procedure. Still, performing the procedure is time-consuming, so it is only executed periodically. To ensure that it is applied at regular intervals, a common strategy is to perform a global relabeling step every $\bar{n}$ discharge steps, where $\bar{n}$ is in the range $[0.4|V|, \ldots, 4|V|]$ for the network $G = (V,E)$. Badics and Boros stated that for different values of $\bar{n}$ within this range, the running times in their computational experiments were almost identical. Anderson and Setubal also stated that for $\bar{n}$ at the values $|V|/2$, $|V|$, and $2|V|$, the variations in the running times were small.

3.3.2. Gap-Relabeling Heuristic. In the generic push-relabel algorithm flow is pushed from a node labeled d to a node labeled $d-1$. A property of the push-relabel algorithm is that an arc exists in the residual graph only if the difference in labels of its endpoints is at most 1. So if there is a node labeled d but no node labeled $d-1$ then there is no path in the residual graph where the first node can push flow to the sink—there is a *gap* between the label d and smaller labels. The label of the node labeled d can then be correctly increased to $|V| = n$. This *gap-relabeling* heuristic was introduced by Derigs and Meier (1989). We now explain this heuristic formally.

DEFINITION 8. Given a network $G = (V,E)$, let g be a preflow and d a valid labeling with respect to g. We call $z \in \{1,2,\ldots,|V|-1\}$ a *gap* if $d(v) \neq z \ \forall v \in V$ and $\exists w \in V$ such that $d(w) > z$.

When a gap z is found, the following gap-relabeling step is performed: $\forall v \in V$,

$$d(v) = \begin{cases} |V|, & \text{if } d(v) > z, \\ d(v), & \text{otherwise.} \end{cases}$$

Derigs and Meier proved that the distance labeling d remains valid after the gap-relabeling step is performed.

3.4. A Sketch of Lerchs and Grossmann's Algorithm

A detailed description and analysis of the LG algorithm is available in Hochbaum (1996). Here we provide only the necessary basics to enable the interpretation of the algorithm's performance.

The LG algorithm maintains at each iteration a collection of node sets, D, with the property that in the related graph $\tilde{G}$, $C(s,D) > C(D,t)$ (sum of capacities of arcs from source to D is greater than the corresponding sum of capacities from D to the sink). Such sets are then candidates to be part of the source set of the cut. The LG algorithm could be viewed as a dual algorithm that works with a superoptimal yet infeasible solution. At each state of the algorithm there is a partition of the set of nodes to a collection of subsets some of which are called strong and others are called weak. The incumbent candidate for maximum closure is the union of the strong subsets, each of which having incapacity larger than outcapacity. This union of subsets is not necessarily a closed set, but its total weight can only be greater than that of the optimal solution.

The LG algorithm works with the graph $G = (V,A)$ to which a root node r is added. That graph does not contain a source and a sink s and t. It will be convenient to refer in this section to a strong node as s which should not be confused with the source set of the related graph discussed in the earlier section.

At each iteration the algorithm creates a spanning forest, called a *normalized tree*, containing a subset A' of the arcs A. The tree is constructed with the property that a maximum closed set in the tree is easily identifiable. That identified set however is not necessarily closed in the graph $G = (V,A)$. However, according to Lemma 1 its value is only greater than that of an optimal closed set in G. Each iteration of the algorithm consists of identifying closure infeasibilities and updating the tree while reducing the gap between the value of the superoptimal solution and the optimal feasible solution.

The normalized tree created is the spanning forest in G appended with a root at a dummy node r (which, according

to acceptable standards for tree data structures, is considered to be at the top of the tree). The normalized tree thus spans all the vertices of V. Let $e_k = [k, \ell]$ be an edge in T such that ℓ is the parent of k.

DEFINITION 9. The edge e_k defines a *branch*, the subtree rooted at k. The branch is denoted by $T_k = (X_k, A_k)$, where X_k is the set of nodes in T_k and A_k is the set of arcs in T_k.

DEFINITION 10. The *mass* M_k of a branch T_k is the sum of the weights of all the nodes in T_k. The edge e_k is said to *support* the mass M_k.

An edge of the tree either points towards the root (upward) or points away from the root (downwards). To distinguish the orientation of the edges with respect to the root in a particular tree, the following definitions are used in Lerchs and Grossmann (1965). They used the notation of p-edge and m-edge, where p and m stand for plus and minus.

DEFINITION 11. An edge pointing away from the root is a *p-edge*, and an edge pointing towards the root is an *m-edge*. Accordingly, the branch defined by a p-edge is a *p-branch* and the branch defined by an m-edge is an *m-branch*.

DEFINITION 12. A p-edge is *strong* if it supports a mass that is *strictly positive*. An m-edge is *strong* if it supports a mass that is *zero or negative*. Edges that are not strong are said to be *weak*.

DEFINITION 13. A branch is *strong* if the edge that defines it is strong, otherwise it is *weak*.

DEFINITION 14. A vertex i is *strong* if there is at least one strong edge on the undirected path in T joining i to the root r. Vertices that are not strong are said to be *weak*.

DEFINITION 15. A tree is *normalized* if the root r is an endpoint of all the strong edges. That is, for every strong edge $[i, j]$, either $i = r$ or $j = r$.

In other words, a tree is normalized if it does not contain any strong edges of A, and the only strong edges are among those adjacent to root.

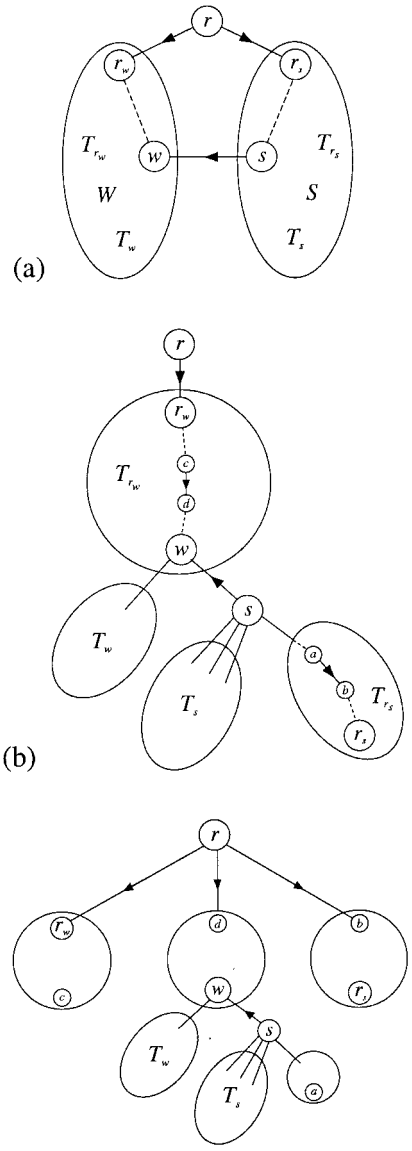
Lerchs and Grossmann showed that for a normalized tree T, the maximum closed set of T is its set of strong nodes, or the union of its strong branches

LEMMA 1. *Given a normalized tree inducing a forest* $F = (V, A')$, $A' \subset A$. *The union of the strong branches is a maximum closed set in* (V, A').

COROLLARY 1. *Any proper subset of the strong nodes is either not maximum, or not closed in* (V, A').

Each iteration of the algorithm consists of identifying an infeasibility in the form of an arc from a strong node to a weak node. The existence of such an arc indicates that the set of strong nodes is not closed and the algorithm may not terminate. The arc is then added in and the tree is updated. The update consists of recomputing the masses of nodes, and removing any strong edges that were created along with the branches they define to become adjacent to the root.

Figure 2. Illustration of the steps of LG algorithm.



An iteration involves three major operations: (1) merger—adding an arc from a strong branch S to a weak branch W and removing the arc from the root of the subtree S to the root; (2) mass updates; (3) renormalization—removal of strong edges and their subtrees/branches and reattaching them to r. These operations are shown in Figures 2 (a), (b), (c), respectively, where in (b) the edges (a, b) and then (c, d) were found strong. Figure 2 is explained in more detail following the description of the algorithm.

Let M_e denote the mass of an edge e. The algorithm gets as input the graph and an initial (normalized) tree. A standard initial normalized tree for a graph G with a vector of node weights b is $T = T_0(G, b) = (V \cup \{r\}, A')$ for $A' = \{(r, j) \mid j \in V\}$, $M_{(r,j)} = b_j$.

PROCEDURE LG. $(G=(V,A), b_j \quad \forall j \in V, T)$.
$V_S^{(T)} = \{v \in V \mid v \text{ is strong in } T\}$.
`While` $(V_S^{(T)}, V \backslash V_S^{(T)}) \neq \emptyset$, `do`
For $(s,w) \in (V_S^{(T)}, V \backslash V_S^{(T)})$, let r_s be the root of the branch containing s and r_w be the root of the branch containing w.
$\{Merger\} T \leftarrow T \backslash (r, r_s) \cup (s,w)$.
$\{Mass\ update\}$
(i) For all edges e on the path $[r_s, \ldots, s]$, $M_e \leftarrow M_{r_s} - M_e$.
(ii) $M_{(s,w)} \leftarrow M_{r_s}$.
(iii) For all edges e on the path $[w, \ldots, r_w]$, $M_e \leftarrow M_{r_s} + M_e$.
$\{Renormalization\}$
`While` there are strong edges on the merger path $[r_s, \ldots, s, w, \ldots, r_w]$,
`repeat`
Let $[a,b]$ be the next strong edge encountered on the path $[r_s, \ldots, s, w, \ldots, r_w]$.
Call **Renormalize edge** $(T, b_j \quad \forall j \in V, [a,b])$.
`end`

Return "$V_S^{(T)}$ is a maximum closed set."

PROCEDURE RENORMALIZE EDGE $(T, b_j \quad \forall j \in V, [a,b])$.
Replace $[a,b]$ by $(r,a): T \leftarrow T \backslash [a,b] \cup (r,a)$.
$\{a$ and the entire subtree rooted at a becomes a separate branch.$\}$
For all edges e on the path $[b, \ldots, r]$, $M_e \leftarrow M_e - M_{[a,b]}$.
Return T.

Figure 2 illustrates the changes in the normalized tree throughout an iteration. In (a), a merger arc (s,w) is identified. In (b), the strong branch S is merged with the weak branch W, and in the process of renormalization the arc (a,b) and then the arc (c,d) are found to support positive weights (masses). Figure 2 (c) illustrates the branches of the tree after the renormalization took place.

4. DATA GENERATION

To compare the performance of the different implementations, we ran the algorithms on a set of real mine data and on sets of generated data. The generated data were used to control the characteristics of the mine (and graph) to demonstrate the measured effect on the performance of the algorithms. The mine characteristics that were controlled and varied include the number of ores, their spatial distribution, mineralization, concentration, size of individual clusters, and mine size.

The characteristics of the generated data emulate those of a copper mine in Canada. That copper mine has 16,800 blocks, with 14 layers and 30×40 blocks per layer. Within the mine, there are clusters of ore rock. The size and shape of the ore clusters vary, and sometimes two or more clusters overlap. Within some ore clusters, there are regions with high concentrations of ore. The weights of the positive blocks in these regions are much higher than the weights of the surrounding positive blocks. For the waste blocks, the negative weights decrease from the top layer down but are constant within a layer. This reflects the fact that excavation costs are higher for blocks that are deeper in the ground but are roughly the same for blocks in the same layer.

In generating our data, all the parameters—the number of ore clusters in the mine, the size and shape of the ore clusters, the weights of the blocks—mimic this type of mine data. In the first set of generated data we have mines on three-dimensional arrays of blocks with 20 layers, 45 rows, and 60 columns. The size of the generated mines is thus 54,000 blocks. That translates to 54,000 nodes in the flow graph. The generated flow graph can be considered with closure in terms of successors, or by reversing the arcs, in terms of predecessors. While these two representations are equivalent, they make a difference in terms of the computational performance. We call the one with the reversed directions of the arcs the *reverse graph*.

There are 10 mines generated for each block value distribution: low, medium, and high. The basic runs provide the mean and the standard deviation for each such set of 10 mines run on the graph and the reverse graphs.

4.1. Generation of Ore Clusters

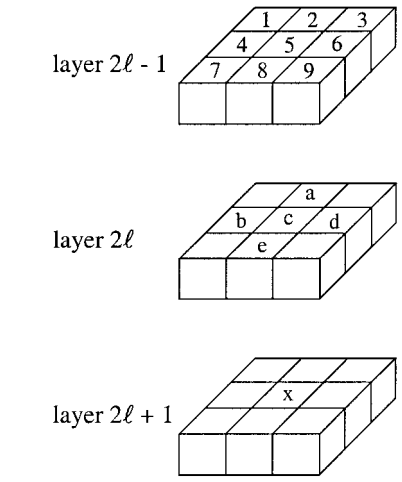
A real ore cluster is a collection of contiguous blocks with positive (nonzero) metal value. To generate such blocks we select randomly the position of a block anchoring each cluster. We then generate randomly one dimensional intervals the endpoints of which are determined by a random shift away from the anchor block. It is necessary to ensure that there is overlap between any pair of spatially adjacent interval to assure contiguity: Each layer can be viewed as a grid with rows and columns. The intervals are within each row and consecutive rows have intervals overlapping in that they use several common columns. This concept is then extended to generate intervals in different layers. The random selection of the translation of blocks in each layer follows a bell-shaped distribution. The distribution functions used are discrete. While we provide here only a sketchy qualitative description of the ore clusters generation, we maintain the detailed distribution tables used and will provide them upon request.

Once the ore clusters have been determined, the ore value in each block of the cluster is also generated randomly so that the concentration is smaller close to the margins of the clusters, and higher towards the center of the ore cluster. Here, too, we used discrete distribution functions. There are three distribution functions, each for high, medium, and low value of ore.

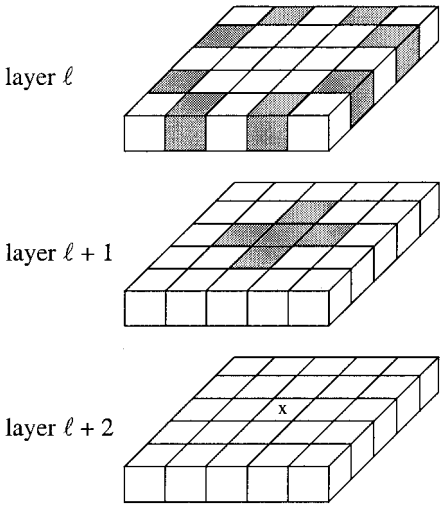
4.2. Edge Pattern Generation

There are several types of edge patterns used by the mining industry to enforce the necessary slope requirement. A common one is called the 1:5:9 pattern. In this pattern the

Figure 3. 1:5:9 pattern and Knight's move pattern.



(a) 1:5:9 pattern



(b) Knight's move pattern

successors of a node follow a different pattern from an odd layer than those from even layer. The block marked "x" in Figure 3 (a) has for successors blocks a through e in the next layer up. Block c in the even layer has 9 successors, 1 through 9, in the odd layer above it.

Another common pattern is the Knights's move. Here each block marked "x" has for successors the shaded nodes in the two layers above it in Figure 3 (b). Notice that because each node that is a successor of x in the first layer above has cross-like set of successors in the layer above it, then in particular the successors of node x, that are two layers above in layer ℓ, include all nodes in the square in the figure except for the four corners. Another pattern involving 22 successors per block will be mentioned later. In the set of basic experiments we use the 1:5:9 pattern.

5. COMPARING LG AND PUSH-RELABEL

5.1. Choosing a Push-Relabel Implementation

Our choice of implementation also took into account the possibility of using publicly available software or our own implementation. We obtained the push-relabel algorithm (the GOLD program) of Badics and Boros available at the DIMACS site. That program admits only FIFO implementation but no LIFO. To test the LIFO implementation we programmed our own push-relabel implementation. To account for the differences in programming style between our LIFO program GOLD's FIFO we also programmed the FIFO push-relabel, which emulates the GOLD program.

We ran the push-relabel algorithm in phase I only, as in the discussion in §3.3. The results for the push-relabel algorithm with the various strategies are shown in Table 1. The notation used to distinguish between the implementations of the push-relabel algorithm is:

Pr: Push-relabel; F: FIFO; L: LIFO; Gl: Global relabeling; Gp: Gap relabeling; D: DIMACS.

We used FIFO GOLD once with the global relabeling heuristic only and another time with the global *and* gap relabeling procedures. The more efficient of the two, with global relabeling only, we programmed to unify our comparison scheme and ensure that the improvements we observed in the version of LIFO that we programmed are not a result of our particularly efficient (or inefficient, as it turns out) software design. Our results for the three FIFO implementation and the two LIFO implementations are reported in Table 1. Each entry is the average of the runs on 10 mines of size 54,000 blocks each.

Among the three FIFO implementations, PrFGID, which is the Global only GOLD program, is the fastest. All implementations perform consistently better on the reverse graphs. This is to be expected because the cut capacity at the sink is less than that at the source. In terms of the economic block model of a mine, that means that the total negative weight of the waste blocks is less in absolute value than the total weight of the positive candidate blocks. This should generally hold true for an economically viable mine.

The push-relabel algorithm pushes first an amount of flow that is equal to the capacity of the arcs adjacent to the source. When this is much larger than the capacity at the sink, then most of the flow contributes to keeping nodes active, and the algorithm does not terminate until the label of these nodes exceeds n. On the other hand, if we start from the sink, then most preflow can be pushed all the way to the source in the reverse graph as there is sufficient capacity to dispose of the excess. An example of this phenomenon is provided in Figure 4.

Indeed, in our experimental results for the high-value ore blocks case, the number of groups of ore blocks that are profitable to mine is highest, and the reduction in the running time when the reverse graphs are used is also highest. For the high-value ore blocks distribution, the average running time of PrFGlD is decreased by 46.35% when the reverse graph is used. For these cases, the average number of discharge steps is decreased by 46.2%.

Table 1. Average running times in seconds (standard deviations) for implementations of the push-relabel algorithm.

Ore value	Graph	PrFGlD	PrFGl	PrFGlGpD	PrLGlGp	PrLGp*
Low	Original	44.4 (6.20)	50.8 (4.69)	47.5 (4.46)	67.1 (12.42)	44.1 (13.89)
	Reverse	32.0 (7.62)	43.9 (5.09)	34.5 (8.19)	31.6 (8.09)	25.6 (9.38)
Medium	Original	44.2 (3.57)	50.1 (4.50)	47.4 (3.26)	60.7 (8.71)	36.6 (6.64)
	Reverse	28.8 (4.02)	44.9 (5.45)	31.2 (5.07)	29.7 (6.65)	24.6 (7.70)
High	Original	46.6 (4.86)	53.2 (3.84)	48.9 (4.91)	64.5 (11.09)	37.1 (9.31)
	Reverse	25.0 (2.61)	39.2 (4.14)	25.7 (2.49)	23.9 (4.28)	17.1 (3.42)

* The first three are FIFO implementations: FIFO Global of GOLD, our FIFO Global, and FIFO Global and Gap of GOLD; the next two are LIFO implementations: with Global and Gap and with Gap only.
Pr: Push-relabel; F: FIFO; L: LIFO; Gl: global-relabeling; Gp: gap-relabeling; D: DIMACS.

We now compare the performances of the best FIFO and LIFO strategies, PrFGl and PrLGp. On the original graphs, the average running time of PrLGp is virtually the same as that of PrFGl for the low value distribution, but is better—17% and 20% faster, respectively—for the medium- and high-value distributions. The better performance for the medium- and high-value distributions is linked directly to the occurrence of gaps. Without global relabeling, gap relabeling is the only step in which a set of distance labels is increased; and a gap relabeling step is only applied when a gap is discovered. When the positive block weights are increased in the medium- and high-value distributions while the negative block weights remain constant, the paths to the sink in the corresponding open-pit networks are more likely to be saturated, making the occurrence of gaps more likely. In the reverse graphs the effect is the opposite—fewer gaps for the medium and high distributions compared to the low distribution.

Note that the Gap heuristic performs well with the LIFO strategy but not with FIFO. Indeed, the FIFO strategy selects each of the active nodes in the queue in turn and performs a push/relabel step. If the number of active nodes is large, a large set of nodes might be relabeled. On the other hand, in the LIFO strategy, the push-relabel actions are repeatedly applied to the node at the top of the stack. No action is performed on the nodes in the rest of stack until they rise to the top. Thus, when the LIFO strategy is used the values of the distance labels may extend over a wider range and may be more scattered than when the FIFO strategy is used. This makes the occurrence of gaps more likely with a LIFO vertex selection strategy, thus the gap relabeling pushes up labels faster and speeds up the algorithm.

Global relabeling also makes the occurrence of gaps less likely because nodes that are disconnected from the sink are identified in the global relabeling process. These observations are confirmed in the experiments where we count the number of gaps found in each run of the FIFO strategy. The averages on the number of gaps found are reported in Table 2.

Since it turns out that our own program for the same implementation of push-relabel with global (PrGl) is inferior to the DIMACS version (compare PrFGlD to PrFGl in Table 1), our claim that the LIFO strategy with the Gap heuristic (which we programmed) is the better approach to use is only strengthened.

5.2. Comparing Push-Relabel and LG Algorithms

We checked two possible implementations of the LG algorithm. In one we chose not to update the status of nodes until they are used in a merger. In the other the status is updated at every iteration. The latter version was faster than the delayed update version by a factor of more than 2. We thus consider the LG algorithm and report only on the node update version. The results of LG algorithm are compared to the best push-relabel algorithm PrLGp. It is evident that push-relabel is significantly more efficient than the LG algorithm as reported in Table 3. Further runs, in which we

Figure 4. The graph and reverse graph.

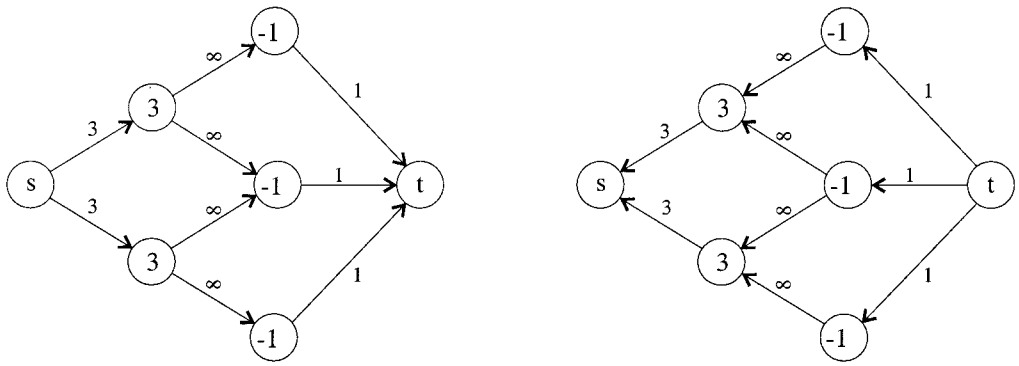


Table 2. Average number of gaps found (standard deviations).

Ore value	Graph	PrFGlGpD	PrLGlGp	PrLGp
Low	Original	4.7 (2.76)	917.4 (256.2)	920.1 (256.9)
	Reverse	3.6 (1.91)	478.9 (106.0)	568.2 (118.5)
Medium	Original	4.2 (2.18)	1428.9 (222.2)	1430.1 (220.4)
	Reverse	4.7 (1.85)	355.3 (78.2)	445.7 (69.9)
High	Original	3.8 (1.89)	1741.6 (346.1)	1744.3 (341.1)
	Reverse	4.1 (1.81)	282.8 (63.4)	410.0 (81.9)

vary features of the data, all indicate that not only is push-relabel more efficient, but also LG algorithm is not robust and tends to exhibit variable and unstable performance with changes in the input data. The table includes a column for w^+/w^-, the purpose of which is to be explained in the coming section.

We analyze in the next section the major factors in the input data that affect the performance of the LG algorithm.

6. INPUT FEATURES AFFECTING LG ALGORITHM

The complexity analysis of LG algorithm in Hochbaum (1996) indicates that the number of iterations depends on the total weight of the positive nodes, $O(\Sigma_{i\in V^+} b_i)$, where $V^+ = \{j \in V \mid b_j > 0\}$. The experimental results in Table 3 appear to indicate that the running time of LG decreases as the positive block weights increase, which is contradictory to the theoretical complexity of the LG algorithm. In this section we discuss the causes of this phenomenon.

Recall that in each iteration of the LG algorithm, a merger arc (s, w) is found that connects a strong branch to a weak one. To renormalize the tree, the masses of nodes (or arcs) are scanned along the merger path that traverses from the root of the strong branch to s and then from w to the root of the weak branch. The amount of work per iteration thus depends on the sizes of the weak and strong branches.

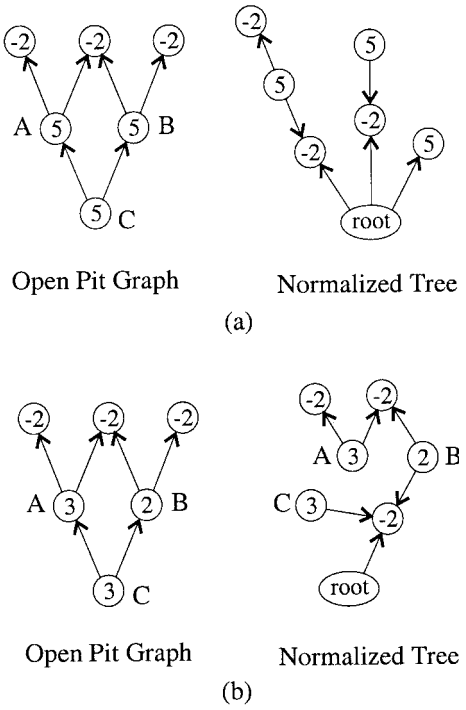
Branch size is also related to the number of iterations of the algorithm. At the start of the LG algorithm, each node constitutes a separate branch. At each iteration, the LG algorithm adds at most one edge to the normalized tree. Hence, branch sizes can become large only after a large number of iterations.

A major factor affecting a strong branch size is the number of negative nodes that a positive node can "support" and still remain strong. That means that the weight of the node is positive and larger than the absolute value of the sum of weights of the negative weight nodes together with it in the same branch. Similarly, a weak branch is large if the weight of the negative nodes it contains can support many positive weight nodes and still remain nonpositive. We illustrate this with examples in Figure 5.

Table 3. Average running times in seconds (standard deviations) for the LG algorithm and push-relabel/LIFO with Gap.

Ore value	Graph	$\frac{w^+}{w^-}$	LG	PrLGp
Low	Original	2.72	158.8 (98.80)	44.1 (13.89)
	Reverse		123.3 (63.82)	25.6 (9.38)
Medium	Original	3.35	94.8 (33.78)	36.6 (6.64)
	Reverse		113.4 (45.98)	24.6 (7.70)
High	Original	4.05	63.2 (30.70)	37.1 (9.31)
	Reverse		71.3 (23.77)	17.1 (3.42)

Figure 5. Effect of weights on branch size.

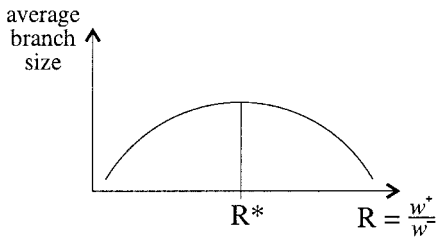


In Figure 5 (a), each of the positive nodes A and B has a weight that is higher than the sum of the weights of its descendants. Therefore, in the normalized tree, the branch containing A and the branch containing B are able to support these weak nodes and remain strong, keeping the sizes of the branches small, as shown in Figure 5 (a). In Figure 5 (b), the reverse is true: Each of the nodes A and B has a weight that is lower than the sum of the weights of its descendants, and the branches containing these nodes become weak. As a result, the branch containing node C must combine with these two weak branches to form a strong branch, creating a much larger branch.

To approximate the number of negative nodes that a positive node can support and still remain strong, we use the ratio $R = w^+/w^-$, where w^+ is the average weight of all the positive nodes in the graph and w^- is the absolute value of the average weight of all the negative nodes.

The branch sizes are also affected by the outdegree of each node and the ratio of positive nodes to the total number of nodes in the graph. To illustrate this, consider the following scenario. Suppose the ratio $R = q$, the average number of outgoing edges from each node is less than q, and there is an equal number of positive nodes and negative nodes. (This is the case in Figure 5 (a).) Then we estimate that each positive

Figure 6. The schematic trade-off between the positive-to-negative weight ratio and the branch size.



node should be able to support the negative nodes that it has arcs to, and branch sizes would remain small. On the other hand, when $R = q$ and there is an equal number of positive nodes and negative nodes, but the average outdegree is at least $2 \cdot q$, then each positive node is not able to support on average the negative nodes that it has arcs to, and branch sizes tend to become large.

To see the effect of R on branch sizes, let the set of edges and the number of positive nodes be fixed. As the value of R increases from 0, the branch sizes would increase, leading to an increase in the computation time. At some value of R, R^*, the branch sizes would start to decrease, leading to a decrease in the computation time, as in Figure 6.

Looking back at Table 3 we note that the performance improves with increased values of $R = w^+/w^-$. We call R^* the *critical value* of $R = w^+/w^-$ if it is the value at which the branch sizes are at their maximum.

The example in Figure 5 shows a decrease in branch sizes as R is increased from 4/3 in Figure 5 (b) to 5/2 in Figure 5 (a). So for values of R less than the critical value for the graph, branch sizes are increasing with increasing R. But for values of R greater than the critical value, branch sizes are decreasing with increasing R.

For values of R close to but less than the critical value, branch sizes tend to be large because positive nodes have to be combined to support negative nodes. But as R increases to values greater than the critical value, branch sizes would decrease because fewer positive nodes have to be combined to form strong branches. This critical value of R is dependent on other characteristics of the graph. For the same set of arcs, if a graph has a larger number of positive nodes, then a larger number of a node's descendants may be positive nodes, so the critical value of R would be lower. For the same set of nodes, if a graph has a larger set of arcs, then the critical value of R would be higher.

To confirm that the change in performance is indeed caused by an increase in the branch size we recorded the maximum number of nodes that were updated for each of the 30 runs. This corresponds to the maximum branch size throughout the algorithm. The results are reported in Table 4.

This effect of the ratio R on the running time of LG has been observed in all of our experiments. To demonstrate the full range of dependence we generated additional random graphs to cover the full possible range of R.

As before, the total number of nodes in these graphs is 54,000 and the average number of positive nodes is 8,000. The negative node weights are uniformly distributed between −160 and −170. To generate graphs with R values 4.7, 5.3, 6.0, 6.6, 7.8, and 9.1, respectively, the positive weights are uniformly distributed between 700 and 800, between 800 and 900, between 900 and 1000, between 1000 and 1100, between 1200 and 1300, and between 1400 and 1500, respectively. For each value of R, 10 random graphs are generated. Each node has 6 outgoing edges, and the endpoints of the edges are randomly chosen among all the nodes in the graph. Because there are only 8,000 positive nodes out of a total of 54,000 nodes, on the average less than 1 of the outgoing edges for each node would be pointing to a positive node. The results are listed in Table 5.

Table 4. Update statistics for different ore value distributions.

Maximum length of merger path	No. of cases		
	Low value	Medium value	High value
1001–2000	0	1	1
2001–3000	1	0	1
3001–4000	0	2	4
4001–5000	1	1	1
5001–6000	3	2	1
6001–7000	2	1	1
7001–8000	0	1	1
8001–9000	0	0	0
9001–10000	1	1	0
10001–11000	1	1	0
11001–12000	1	0	0

As R increases, there is initially an increase in the average running time, followed by a decrease in the average running time. The results show that the critical value of R for these graphs is around 6. At $R = 5.3$, the number of nodes in the maximum closure is 0, but at $R = 6.0$, the number is increased to an average of 53885. The average number of iterations and the average amount of time spent per iteration are higher than in the open-pit graphs because of the higher average number of descendants per node.

7. EFFECT OF MINE FEATURES ON LG AND PUSH-RELABEL

We modify here several characteristics of the mine to measure the effect of these features on the algorithms'

Table 5. Effect of R on the LG algorithm in random graphs.

R	Average running time in seconds (standard deviation)	Average no. of iterations	Average time spent per iteration in milliseconds
4.7	77.9 (20.2)	57619	1.4
5.3	849.9 (366.1)	78600	10.8
6.0	2381.9 (437.6)	92773	25.7
6.6	951.1 (184.9)	74242	12.8
7.8	199.9 (57.5)	58990	3.4
9.1	52.2 (25.4)	53023	1.0

Table 6. Average running times in seconds (standard deviation) for graphs with small and large size of ore clusters.

Ore Cluster Size	Ore Block Value	Graph	LG	PrFGlD	PrLGp
Small	Low	Original	84.3 (32.15)	43.3 (4.41)	42.0 (9.13)
		Reverse	93.9 (60.77)	25.0 (5.69)	16.3 (3.23)
	Medium	Original	49.5 (23.85)	42.9 (2.71)	34.8 (8.81)
		Reverse	57.3 (28.43)	25.4 (3.67)	14.7 (3.10)
	High	Original	50.0 (21.80)	43.4 (2.71)	32.5 (5.37)
		Reverse	61.0 (20.66)	23.2 (4.26)	16.0 (4.73)
Large	Low	Original	117.0 (62.38)	40.9 (5.59)	35.7 (12.67)
		Reverse	122.6 (75.27)	34.9 (8.62)	21.2 (4.62)
	Medium	Original	134.2 (76.83)	44.8 (5.10)	42.0 (10.69)
		Reverse	116.0 (48.72)	31.4 (7.42)	20.8 (7.49)
	High	Original	132.8 (115.53)	48.3 (5.29)	38.3 (9.24)
		Reverse	116.8 (66.44)	27.1 (5.58)	18.9 (6.20)

performance. We change the size and number of ore clusters, the mine size in terms of number of blocks, and the edge pattern in terms of the outdegree of each node.

7.1. Size and Number of Ore Clusters

We refer to the size of the ore clusters in the first set of experiments as *regular*. For the next 10 sets of data, the size of the ore clusters is halved. Then in the following 10 sets of data, the size of the ore clusters is doubled. We will refer to these two sizes as small and large, respectively. For the small ore clusters, 20 ore clusters are generated for each data set, whereas for the large ore clusters, 5 are generated. This keeps the total number of positive blocks in each data set close to constant. The rest of the parameters remain unchanged. Hence, compared to the initial data sets, the only change is in the distribution of the positive blocks. n is 54,000, m is 356,350, the lower bound on the number of positive nodes is 8,000, and the values of R for the low-, medium-, and high-value distributions remain at 2.72, 3.35, and 4.05, respectively, on the original graphs and the reciprocals of these values for the reverse graphs.

The results are listed in Table 6. The push-relabel algorithm is not greatly affected by large ore clusters, but its performance is improved compared to the regular ore clusters when the ore clusters are small. When the ore clusters are small, the average number of arcs from a positive node to a negative node in the open-pit network is also small. Because the negative nodes are the only nodes with arcs to the sink, this means that the average distance from positive nodes to the sink is also short. Hence, the computation time of the push-relabel algorithm is reduced. The average times spent per iteration remain at around 70 microseconds for PrFGlD and around 40 microseconds for PrLGp, the same as for the experiments in Table 3.

Small ore clusters do benefit the LG algorithm, while large clusters adversely affect it. Because for each value distribution R is the same for the small and large ore cluster data sets, the difference in running time can be attributed to the distribution of the positive nodes. When the ore clusters are small, the average number of overlying blocks would be smaller than when the ore clusters are large. Accordingly, the average branch size in the normalized tree would be smaller. Our empirical results show that when the ore clusters are small, the CPU times spent per iteration are 2.0, 1.3, and 1.4 milliseconds (2.3, 1.5, and 1.6 milliseconds for the reverse graphs) for the low-, medium-, and high-value distributions, respectively. When the ore clusters are large, the times spent per iteration are increased to 2.9, 3.2, and 3.1 milliseconds (3.1, 2.9, and 2.9 milliseconds for the reverse graphs) for the three distributions. Notice that for the large ore clusters, the average computation time is increased from the low value distribution to the medium-value distribution. This is because the number of nodes in the maximum closure increased from 17795 to 20432.

To get data sets with twice the number of ore clusters we generate 10 data sets with 20 ore clusters, and the lower bound on the number of positive blocks is doubled to 16,000.

For graphs with double the number of (regular-sized) ore clusters, the results are listed in Table 7. On the reverse graphs, the performance of the push-relabel algorithms is improved across the board. When the number of ore clusters is doubled, the number of feasible pits that are profitable to mine is also increased; this makes reversing the network even more beneficial. On the original graphs, the performance of the push-relabel algorithms remains

Table 7. Average running times in seconds (standard deviation) for graphs with double the number of ore clusters.

Ore Block Value	Graph	LG	PrFGlD	PrLGp
Low	Original	69.6 (16.11)	62.8 (5.17)	47.2 (5.25)
	Reverse	72.8 (15.81)	24.7 (3.35)	14.2 (2.75)
Medium	Original	48.0 (18.20)	57.8 (6.27)	39.9 (8.18)
	Reverse	51.3 (24.44)	19.0 (2.72)	10.7 (2.64)
High	Original	54.4 (36.49)	54.0 (6.83)	40.2 (8.00)
	Reverse	51.4 (28.27)	19.2 (1.99)	10.1 (2.26)

relatively stable, except that PrFGlD slows down. As the number of positive blocks is increased, the amount of flow sent from the source is also increased, so the algorithm has more excess flow to process and push. The average times spent per iteration remain at approximately 70 microseconds for PrFGlD and approximately 40 microseconds for PrLGp.

LG runs about twice as fast for the low- and medium-value distributions on these graphs as on the set of graphs with approximately half the number of positive blocks in Table 3. This is attributed to the sizes of the branches in the normalized tree: With approximately twice as many positive blocks, and a corresponding reduction in the number of negative blocks, the branches of the normalized tree are kept small. The low average times spent per iteration, 1.5, 1.2, and 1.3 milliseconds (1.6, 1.3, and 1.3 milliseconds for the reverse graphs) for the low, medium, and high distributions, respectively, confirms this explanation.

7.2. Mine Size

There is a concern in the mining industry that as the number of nodes in the open-pit graph increases, the algorithms will not be able to produce the optimal pit in reasonable amount of time. To test for the rate of increase in the running time as a function of mine size, we increase the dimensions of the economic block models for the next set of experiments.

For the next 10 sets of data, the dimensions of the economic block model are increased from $20 \times 45 \times 60$ to $20 \times 64 \times 85$. Each of these models has 108,800 blocks, approximately doubling the number of blocks in all the previous models, each of which has 54,000 blocks. Accordingly, we double the number of ore clusters to 20 and double the lower bound on the total number of positive blocks to 16,000. The weights of the positive blocks are generated according to the medium-value distribution. The rest of the parameters are the same as in the first set of experiments in §5. In these data sets, $n = 108{,}800$, $m = 356{,}350$, $R = 3.35$ for the original graphs, and the lower bound on the number of positive nodes is 16,000.

To test the algorithms on yet larger mines, the dimensions are next increased to $25 \times 70 \times 93$, creating 162,750 blocks, tripling the number of blocks in the models in the previous sections. The number of ore clusters is increased to 25, and the lower bound on the total number of positive blocks is set to be 24,000. As before, positive block weights are generated according to the medium-value distribution, and the other parameters are the same as in the first set of experiments in §5. Here, $n = 162{,}750$, $m = 356{,}350$, $R = 3.35$ for the original graphs, and the lower bound on the number of positive nodes is 24,000.

Table 8. Average running times in seconds (standard deviation) for large mines.

Dimensions	Graph	LG	PrFGlD	PrLGp
$20 \times 64 \times 85$	Original	301.8 (153.62)	107.4 (11.38)	101.8 (31.44)
	Reverse	314.3 (162.75)	69.7 (9.79)	45.1 (10.65)
$25 \times 70 \times 93$	Original	568.5 (244.37)	139.8 (5.43)	173.6 (55.58)
	Reverse	691.2 (224.00)	106.9 (16.19)	76.4 (13.37)

The results are listed in Table 8. We compare these results to the computation times for the medium value ore blocks distribution (without preprocessing) for the $20 \times 45 \times 60$ mines in Table 3. For the $20 \times 64 \times 85$ mines, LG's average running time more than triples, and PrLGp and PrFGlD's average running times more than double. For the $25 \times 70 \times 93$ mines, the average running time for LG is about six times that for the $20 \times 45 \times 60$ mines, while the average running times for PrFGlD and PrLGp are approximately tripled. This performance is graphed in Figure 7.

The average times spent per iteration remain stable at approximately 60 to 70 microseconds for PrFGlD and at approximately 40 microseconds for PrLGp. For the $20 \times 64 \times 85$ mines, the average time spent per iteration of the LG algorithm is increased from 2.25 milliseconds to 3.3 milliseconds (3.5 milliseconds for the reverse graphs). For the $25 \times 70 \times 93$ mines, it is further increased to 4.3 milliseconds (5.2 milliseconds for the reverse graphs). The average number of iterations is also increased from 42,150 to 92,718 for the $20 \times 64 \times 85$ mines and to 133,520 for the $25 \times 70 \times 93$ mines.

One feature in which the LG algorithm has an advantage is that the push-relabel algorithm has a much larger memory requirement than the LG algorithm. For each edge in the open-pit graph, it is sufficient for the LG algorithm to store the edge and the mass that the edge supports. However, for the push-relabel algorithm, each arc (i, j) has to be on node i's edge list as well as node j's, and the capacities of both arc (i, j) and arc (j, i) have to be stored. As a result, when the data set is large and cannot be stored entirely in the main memory, the push-relabel algorithm will consume more system time.

Figure 7. Running time as a function of mine size.

Mine Size	LG	PfFGlD	PfLGp
54,000	94.8	28.8	24.6
108,800	301.8	69.7	45.1
162,750	568.5	106.9	76.4

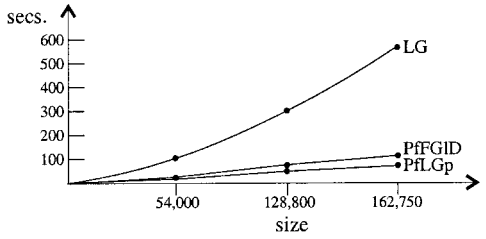


Table 9. Average system times (standard deviation) for large mines.

Dimensions	Graph	LG	PrFGlD	PrLGp
$25 \times 70 \times 93$	Original	5.3 (1.55)	468.2 (178.27)	22.9 (6.62)
	Reverse	5.4 (1.28)	720.0 (156.40)	34.22 (4.34)

The system times for the set of experiments on large mines are listed in Table 9. The difference is dramatic. PrFGlD requires a large amount of system time since for each global-relabeling step, the edge lists of the nodes have to be scanned, and this data has to be brought into the main memory in turn. Thus, when the amount of memory is small but the data set is large, the efficiency of PrFGlD is substantially decreased.

7.3. Choice of Edge Pattern

We ran the experiments for three types of edge patterns. In addition to the 1:5:9 pattern used in the basic experiments, we ran also the Knight's move and the 22-edge pattern. The conclusion is that the increase in the node outdegree and thus the total number of edges degrades the performances of the algorithms.

For this set of experiments, we use the economic block models from §5, where $n = 54{,}000$, the values of R for the low-, medium-, and high-value distributions are 2.72, 3.35, and 4.05, respectively, on the original graphs, and the lower bound on the number of positive nodes is 8,000. The difference is in the edge pattern that is used to generate the edges in the open-pit graphs.

In the first set of experiments, the *Knight's move pattern*, shown in Figure 3 (b), is used to generate the edges. This pattern generates 13 edges for each node. The graphs created using this pattern have approximately twice the number of edges as those in §5 as for the 1:5:9 pattern the average node degree is 7. We will refer to these graphs as *Knight's move graphs*. For these graphs the number of edges, $m = 618{,}918$.

In the second set of experiments, the 22-edge pattern described by Caccetta and Giannini (1988) is used. Compared to the 1:5:9 pattern the number of edges is more than tripled. We refer to these graphs as *22-edge graphs*. For these graphs, $m = 980{,}324$.

The average computation times and the standard deviations are listed in Table 10. The preprocessing times are included in the computation times in Table 10 (a). We use this data in the next section to evaluate the effectiveness of the preprocessing technique.

Table 10. Effect of preprocessing and comparing edge patterns.

(a) Average running times in seconds (standard deviation) for Knight's move graphs.

		LG	PrFGlD	PrLGp
	Without preprocessing			
Low	Original Graph	205.6 (140.54)	64.7 (7.47)	64.7 (18.52)
value	Reverse Graph	130.6 (75.13)	44.0 (8.64)	34.1 (10.91)
ore	*With preprocessing*			
blocks	Original Graph	211.2 (140.04)	60.4 (6.04)	74.0 (20.26)
	Reverse Graph	132.6 (73.38)	43.4 (9.33)	37.7 (10.46)
	Without preprocessing			
Medium	Original Graph	115.7 (45.17)	66.0 (5.29)	49.8 (9.37)
value	Reverse Graph	126.2 (55.40)	40.3 (6.29)	33.7 (10.84)
ore	*With preprocessing*			
blocks	Original Graph	124.4 (44.65)	61.7 (5.76)	59.1 (9.17)
	Reverse Graph	127.5 (56.42)	42.8 (6.57)	38.7 (11.04)
	Without preprocessing			
High	Original Graph	73.3 (38.23)	68.9 (6.39)	51.4 (12.42)
value	Reverse Graph	72.0 (27.93)	34.0 (4.37)	23.2 (4.98)
ore	*With preprocessing*			
blocks	Original Graph	80.7 (36.25)	61.4 (7.23)	61.0 (12.59)
	Reverse Graph	72.4 (27.44)	36.9 (4.06)	29.1 (5.41)

(b) Average running times in seconds (standard deviation) for 22-edge graphs.

Ore Block Value		LG	PrFGlD	PrLGp
Low	Original Graph	325.8 (211.73)	107.0 (10.56)	101.0 (31.92)
	Reverse Graph	274.0 (154.36)	71.0 (11.38)	49.7 (14.72)
Medium	Original Graph	195.5 (86.45)	107.4 (9.34)	83.9 (16.43)
	Reverse Graph	290.7 (126.18)	71.7 (14.50)	60.3 (24.54)
High	Original Graph	143.2 (104.32)	112.8 (10.05)	94.2 (31.62)
	Reverse Graph	164.4 (93.92)	57.2 (9.94)	39.3 (7.64)

The results in Table 10 show that the push-relabel algorithms still outperform LG by a wide margin. We compare these running times with the corresponding running times reported in Table 3, which are running times for graphs corresponding to the same economic block models but with approximately half the number of edges. The push-relabel algorithms show an increase in the average running time of approximately 35% to 50%. The average time spent per discharge step is increased from around 70 microseconds to approximately 110 microseconds for PrFGlD and increased from around 40 microseconds to approximately 70 microseconds for PrLGp. The number of discharge steps remains relatively stable. The average times per iteration for LG are slightly higher than before at 3.89, 2.46, and 1.70 microseconds (2.60, 2.60, and 1.69 microseconds for the reverse graphs) for the low-, medium-, and high-value distributions, respectively.

Comparing the average running times in Tables 10 (a) and 10 (b), the push-relabel algorithms demonstrate approximately 50% increase in computation time as the number of edges in the edge generation pattern is increased from 13 to 22. The average time spent per discharge step is further increased from 110 microseconds to approximately 175 microseconds for PrFGlD, and further increased from around 70 microseconds to approximately 105 microseconds for PrLGp. The number of discharge steps remains relatively stable. The average times per iteration for LG are increased considerably at 5.03, 3.48, and 2.72 microseconds (4.50, 4.60, and 3.10 microseconds for the reverse graphs) for the low-, medium-, and high-value distributions, respectively. This shows the importance of carefully choosing an edge pattern to generate the edges in the open-pit graph and preferring, if possible, patterns with small node outdegrees.

We compare the optimal solutions derived for the same mines but different edge patterns. The less restricting edge pattern—the Knight's move—gives a higher optimal value. The difference in the number of blocks in the optimal pits for the two types of graphs is less than 1%. The optimal values for the 1:5:9 graphs are 1% to 2% lower than that of the Knight's move graphs. Since each node on an even layer in the 1:5:9 graphs has more descendants than its corresponding node in the Knight's move graphs, more negative blocks will have to be excavated to uncover an ore cluster. This can render an ore cluster unprofitable to mine, or decrease its profits. The lesson is to not include slope restrictions that are not essential because these tend to reduce the set of feasible mines and have an adverse effect on operational profits.

8. PREPROCESSING

In this section, we review the concept of preprocessing and its validity. We then test the effectiveness of preprocessing the input data.

8.1. The Preprocessing Approach

The preprocessing approach is based on *relaxing* the slope requirements in a problem. This relaxation is equivalent to the relaxation of some of the precedence constraints in the formulation. Using an algorithm that is computationally very efficient (typically linear time) to compute the optimal solution with the relaxed slope requirements, the goal is to derive a solution (pit) that contains the optimal pit.

We say that the slope requirement of 60° is more restrictive than the slope requirement of 45°, or $45° \preccurlyeq 60\%$. We call a mining problem *relaxed* if it is solved for a less restrictive slope requirement than prescribed.

Solving the relaxed problem for a less restrictive slope s_1 results in a pit p_1 that does not necessarily contain the optimal pit for s_0, P_0. To guarantee that it does, let the closure of P_1 with respect to s_0 be denoted by $P_1(s_0)$. While it is not necessarily the case that $P_0 \subseteq P_1(s_0)$, one can prove that if $P_1 = \emptyset$ then $P_0 = \emptyset$.

LEMMA 2. *Let $s_1 \preccurlyeq s_0$. If the solution to the relaxed problem for slope s_1, P_1, is empty for a block model V, then the optimal solution for s_0, P_0, also satisfies, $P_0 = \emptyset$.*

PROOF. If P_0 is not empty, then it is a feasible pit also for s_1. Thus, P_0 must have a negative value, else P_1 would not be optimal for s_1. But then a nonempty optimal pit is never negative. □

The claim of the lemma is used to generate a pit that is guaranteed to contain an optimal pit by calling repeatedly for the solution of the relaxed problem on a set of blocks from which we remove with each call the pit found for the less restrictive slope requirement and its closure with respect to s_0. When in the remaining set of blocks the solution set to the relaxed problem is empty, then the process terminates. The following algorithm is a formal statement of this approach. Algorithm Preprocess produces a pit contour $\bar{V}$ that *contains* an optimal solution pit (for s_0).

Algorithm **Preprocess** ($s_0 \preccurlyeq s_1$, V)
Set $\bar{V} = \emptyset$.
`repeat`
Solve relaxed problem on V with s_1 with an optimal solution P_1.
`if` $P_1 = \emptyset$, then `STOP:` output $\bar{V}$.
`else`
For $P_1(s_0)$ the closure of P_1 w.r.t. s_0, $\bar{V} \leftarrow \bar{V} \cup P_1(s_0)$.
$V \leftarrow V - \bar{V}$.
`end`

When the algorithm terminates, $\bar{V}$ is the union of the sets with the relaxed requirements closed with respect to s_0. It is thus a feasible solution for s_0 containing an optimal solution with respect to s_0.

Other fast ultimate pit limit algorithms that can be used for preprocessing include Johnson and Sharp's (1971) dynamic programming method, the best valued cross-section bound and the modified best valued cross-section bound of Barnes and Johnson (1982), and Koenigsberg's dynamic programming method. The first three algorithms are extensions of Lerchs and Grossmann's two-dimensional pit optimization algorithm.

Table 11. Average running times in seconds (standard deviations) with preprocessing.

		LG	PrFGlD	PrLGp
Low value ore blocks	Original Graph	164.6 (93.79)	46.1 (3.83)	53.2 (13.65)
	Reverse Graph	126.0 (61.49)	36.4 (6.70)	32.3 (8.92)
Medium value ore blocks	Original Graph	102.0 (32.83)	46.1 (5.56)	44.9 (6.96)
	Reverse Graph	112.0 (42.65)	33.4 (4.57)	30.0 (6.96)
High value ore blocks	Original Graph	69.1 (31.26)	45.0 (6.40)	44.7 (10.30)
	Reverse Graph	74.7 (23.86)	28.6 (3.77)	22.6 (4.18)

For an algorithm solving a relaxed problem, consider for instance Lerchs and Grossmann's two-dimensional pit optimization algorithm. The algorithm works for a two-dimensional rectangular grid with slope requirements of 45°, i.e., from each block to the three blocks directly above and diagonally above it.

Algorithm **Two-Dimensional Pit** ($B_{m\times n} = \{b_{ij}\}$)
$M_{ij} = \Sigma_{k=1}^{i} b_{kj}$, for $i = 1,\ldots,m$, for $j = 1,\ldots,n$,
{the value of the single column above the block in row i and column j}.
$P_{0j} = 0$ for $j = 1,\ldots,n$.
For $i = 1,\ldots,m$, for $j = 1,\ldots,n$, do
$P_{ij} = M_{ij} + \max_{k=-1,0,1}\{P_{i+k,j-1}\}$
enddo
Return $P_{\max} = \max_{k=0,\ldots,m}\{P_{kn}\}$.
end

The value of P_{ij} computed is the value of the optimal pit restricted to columns $1,\ldots,j$ with the constraint that it includes block i. The value returned, $P_{\max}$, indicates the value of the optimal pit. The optimal pit contour can be traced by backtracking from the block where the value of $P_{\max}$ is attained. This algorithm was used in our experiments for preprocessing.

The average running times with preprocessing are shown in Table 11. Comparing these with the corresponding computation times in Table 3 without preprocessing reveals that preprocessing is not effective on our data sets. The reduction in the number of blocks in the input data varies from 27% to 45%, and the optimal pits contain 35% to 73% of the blocks in the reduced models, with an average of 59%. The increase in the overall running times is attributed to the time required to run the preprocessing algorithm.

The results given for the Knight's move graphs with and without preprocessing, in Table 10 (a), further confirm the conclusions from Table 11.

We should point out that Giannini et al. found preprocessing to be a very effective strategy. Their experiments were run on data sets from two gold mines. For the first data set, a series of experiments was run in which first the block size is varied and then the slope requirement is varied. In these experiments, preprocessing reduced the number of blocks by an average of 85%, and the optimal pits contain an average of 87% of the blocks in the reduced block models. For the second set of data, preprocessing reduced the number of blocks by 96%, and the optimal pit contains 59% of the blocks in the reduced block model.

One noticeable difference between our data sets and Giannini et al.'s data sets is the percentage of blocks contained in the optimal pit. In our data sets, for the low value ore blocks distribution, an average of 35% of the blocks is contained in the optimal pits, and for the medium and high value ore blocks distributions, the averages are 37% and 43%, respectively. In Giannini et al.'s first data set an average of 13% of the blocks is contained in the optimal pits, and in the second data set only 2% of the blocks is contained in the optimal pit. This explains their success in preprocessing because their data sets contain a huge number of blocks that are not profitable to mine, and most of these are removed by preprocessing.

9. PARAMETRIC ANALYSIS

The current practice for conducting parametric analysis is by *repeated* applications of the optimization procedure while taking advantage of the fact that for a monotone sequence of parameter values the pits are nested. In contrast, our parametric approach is to retain the state of the network, i.e., the distance labels, the flow values on the edges for push-relabel, and the normalized tree for LG. This information is then used as a starting state for the next application of the parameter value.

The parametric push-relabel algorithm retains distance labels and flows as the parameter is increased from one value to the next. This algorithm was proved to have the same complexity as a single run of push-relabel by Gallo et al. Parametric LG retains the same branches in the normalized tree from one parameter value to the next; only that some branches are then identified as strong whereas with the previous parameter they were weak. This parametric algorithm also has the same complexity as a single run of the LG algorithm as proved in Hochbaum (1996).

The two versions of the parametric push-relabel algorithm are denoted with the prefix P. When push-relabel is implemented so that the computation restarts for every new parameter value (while deleting the blocks in the previously selected pit), the distance labels are reinitialized and these push-relabel implementations are *repeated* applications and have the prefix of R. The same notation is used for LG with

Table 12. Average running times in seconds (standard deviation) for parametric algorithms versus repeated algorithms.

Ore clusters	RLG	PLG	RPrLGp	PPrLGp
Small	392.1 (149.4)	271.7 (88.04)	97.2 (14.34)	75.9 (9.79)
Medium	673.9 (184.13)	434.1 (107.27)	114.9 (17.47)	91.6 (16.75)
Large	878 (438.68)	495.6 (227.87)	115.8 (27.91)	90.5 (23.56)
Double no. of small clusters	631.6 (239.84)	488 (202.56)	119.3 (23.11)	105.4 (18.31)

the notation of PLG and RLG for the parametric and repeated versions, respectively.

We compared the performance of PLG and PPrLGp with repeated applications of LG and PrLGp for sensitivity analysis. By repeated applications, we mean that we still take advantage of the fact that the series of pits generated through sensitivity analysis, for monotone increasing parameter values, is nested, but reinitialize the normalized tree for LG and the distance labels for push-relabel.

9.1. Data Generation for Parametric Runs

LG and PrLGp are first modified so that they accept the same set of input data as PLG and PPrLGp. Instead of having a single block weight for each node, these implementations now accept two values, $cost(i)$ and $metal(i)$, for each node i. The input also includes a monotonically increasing sequence of parameter values. We will refer to these revised implementations as RLG (repeated applications of LG) and RPrLGp (repeated applications of PrLGp).

For simplicity we assume that the weight of block i is a function of the form $cost(i) + \lambda * metal(i)$, where $cost(i)$ is the cost (negative quantity) of excavating block i, $metal(i)$ is the amount of metal contained in block i, and λ is the unit price of the metal. The value of $cost(i)$ for a block i is derived with the same distribution as was used for generating the negative block weights in the previous experiments. As for *metal*, we use the positive block weights generated according to the medium value ore blocks distribution to compute them. Suppose the positive weight $m(i)$ is generated for block i. Then $metal(i)$ is computed as $metal(i) = (m(i) - cost(i))/\alpha$, where $cost(i)$ is assigned according to the same cost distribution as previously, and α is arbitrarily set to 30.

Similarly, data sets with small and large ore clusters, and with double the number of small ore clusters, are generated based on the corresponding data sets in §5. For each different ore cluster size and quantity, 10 data sets are generated. The sequence of parameter values for all of the experiments is set to be $10, 15, 20, 25, 30, 35, 40, 45$, and 50.

9.2. Empirical Results of Parametric Runs

To avoid having to start RLG and RPrLGp manually for each parameter value, we have also modified these implementations so that they re-start automatically. RLG (RPrLGp) starts by computing the first set of node weights (arc capacities) according to *cost*, *metal*, and the first parameter value. Then it computes the optimal pit in this graph in the same way that LG (PrLGp) computes the optimal pit in a graph. After the optimal pit is obtained, all the nodes contained in the optimal pit are removed from the graph. In the remaining graph, the node weights (arc capacities) are updated according to the next parameter value. Then RLG (RPrLGp) re-initializes the graph according to the initialization step of LG (PrLGp) and computes the next optimal pit. This process is repeated until the optimal pit for each parameter value has been computed.

All four implementations generate as output a series of nested pits, where each pit is optimal with respect to the corresponding parameter value.

The average computation times, excluding input and output times, and the standard deviations, are shown in Table 12.

PPrLGp performs the best in all cases and is much more efficient than PLG. The results also show that the parametric algorithms are more efficient than applying the nonparametric algorithms repeatedly for sensitivity analysis. PLG shows a 23% to 44% improvement over RLG, and the improvement is most significant for the cases where the previous sections' experiments show that LG is slow. PPrLGp shows a 12% to 22% speed up over RPrLGp. This improvement is less significant than that of the parametric LG algorithm over the nonparametric LG algorithm. We discuss this further later in this section.

The reverse graphs are not used in these experiments. Reversing a network is beneficial when the number of groups of ore blocks that are profitable to mine is high. This is not expected to be the case in these experiments. In these experiments, the sequence of parameter values starts at a low value and increases gradually. Hence, the number of groups of ore blocks that are profitable to mine is increased gradually. Furthermore, the optimal pit corresponding to each parameter value is removed from the graph after it is computed. Therefore, for each application of the algorithms, the number of groups of ore blocks that are profitable to mine is expected to be small.

Notice that the running times of push-relabel are larger than those for a single parameter application although the complexity bound for single parameter and multiple parameters is the same. One reason for the degraded performance is that in the parametric network, the capacities of the arcs (v, t) are nonincreasing functions of λ. When the arc capacities are updated some arc capacities (v, t) decrease and become lower than the flow value on the arc. In that case

the extra flow amount becomes excess at node v, and the algorithm has to perform discharge steps to remove the excess. The more often that this happens, the more work the parametric push-relabel algorithm would have to perform.

In the parametric open-pit network, for every node v with $metal(v) > 0$, the capacity of the arc (v, t) is decreasing toward zero. So potentially, PPrLGp has to deal with excess created at each of these nodes v because of a decrease in the capacity of the arc (v, t). This explains why in our experiments PPrLGp does not show as significant an advantage over RPrLGp (a 12% to 22% speed up) as PLG does over RLG (a 23% to 44% speed up).

10. EXPERIMENT ON REAL DATA FROM A GOLD DEPOSIT

The implementations LG and PrLGp were tested on data obtained from Newmont Gold Company. The data represent a gold mine given as a set of blocks and an edge pattern.

The format of the input data differs from the standard input format required for the GOLD program, which needs the full description of the complete set of edges as input. While it is possible in principle to generate the network, this network corresponding to the gold mine has close to one billion arcs, so storage becomes a difficulty. Our implementations LG and PrLGp were modified to accept an economic block model and an edge pattern as input. Parts of the open-pit graph are generated as needed by the algorithms. The GOLD program PrFGlD was not modified to accept implicitly the network because it is not our own implementation and we treat it as a black box.

The dimensions of the mine, in terms of blocks, are $56 \times 101 \times 131$, with 12,573 positive blocks, 369,140 negative blocks, and 359,223 air blocks. Each block has dimensions of 50 ft $\times$ 50 ft $\times$ 20 ft. The slope requirement is set at a 45° angle, and the edge pattern consists of 1849 edges, i.e., the number of successors of each node is 1849. Because the data are proprietary, we are not permitted to describe its characteristics. The experiments confirm, however, the fact that the push-relabel algorithm is much more efficient than the LG algorithm.

The experiments were run on a 100-Mhz SGI challenge (IP19) computer system with 640 MB of memory and 1 MB secondary cache. The running time (excluding input and output times) for LG is 19,558 seconds, while that for PrLGp is 2,391 seconds on the original network and 1,858 seconds on the reverse network. The running time of PrLGp on the reverse network is better than that of LG by a factor of more than 10. This is consistent with our experiments on generated data, where the push-relabel algorithm outperforms the LG algorithm by a wide margin.

11. CONCLUDING REMARKS

We study here two algorithms for the open-pit mining problem, or equivalently, minimum cut on closure graphs. We demonstrate that the push-relabel algorithm has a superior performance compared to the algorithm of Lerchs and Grossmann for all types of mine data we tested. The LG algorithm, however, has an advantage with its more economical use of space and memory, which reduces overhead.

We provide an analysis that explains the reasons for the lack of robustness of the LG algorithm and predicts its performance on various mine data, depending on the features of the data. Specifically, we demonstrate that LG is sensitive to the weight distribution in the given input, and we provide the ratio quantity between the total positive and negative weights as the one parameter that predicts the performance of the LG algorithm. Both LG algorithm and the push-relabel algorithm benefit from applying the algorithm to the reverse graph. For the push-relabel algorithm this benefit is more pronounced and consistent. The LG algorithm not only has larger running times than any of the push-relabel variants, but it also has a rate of growth with input size that is substantially worse than that of the push-relabel algorithm.

We tested a parametric implementation of both the push-relabel and the LG algorithm and demonstrated that the concept of maintaining distance labels for push-relabel and the normalized tree structure for the LG algorithm between consecutive applications of different parameter values reduces the overall running time compared to the commonly used approach.

Finally, we demonstrate that the choice of edge pattern affects the running time of all the algorithms we tested, where patterns that have higher node degrees require greater amounts of run time.

While the performance of LG demonstrated here is inferior to that of the push-relabel, this does not preclude the possibility that variants of the algorithm could perform well. This is so if the variants have features built in to address the current weaknesses in the LG's performance. Indeed, we are undertaking now the implementation of the strongly polynomial variant of the LG algorithm which has theoretical complexity similar to push-relabel. Preliminary results of that variant implementation (reported in Anderson and Hochbaum 1999) demonstrate performance that for several classes of graphs is superior to the push-relabel algorithm.

Note added in proofs: Inspired by the LG algorithm, we devised a new, strongly polynomial algorithm for the maximum flow problem in Hochbaum (1997). This algorithm runs in practice faster than all known implementations of the push-relabel algorithm.

ACKNOWLEDGMENTS

Research was supported in part by ONR contract N00014-91-J-1241, by NSF award No. DMI-9713482, by NSF award No. DMI-0084857, and by SUN Microsystems.

REFERENCES

Ahuja, R. K., J. B., Orlin. 1989. A fast and simple algorithm for the maximum flow problem. *Oper. Res.* **37**(5) 748–759.

Alford, C. G., J. Whittle. 1986. Application of Lerchs-Grossmann pit optimization to the design of open pit mines. *AusIMM/IE Aust Newman Combined Group*, *Large Open Pit Mining Conference*. Oct., 201–207.

Anderson, R. J., J. C. Setubal. 1993. Goldberg's algorithm for maximum flow in perspective: A computational study. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **12** 1–18.

Badics, T., E. Boros. 1993. Implementing a maximum flow algorithm: experiments with dynamic trees. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science* **12** 43–63.

Balinski, M. L. 1970. On a selection problem. *Management Sci.* **17**(3) 230–231.

Barnes, R. J., T. B. Johnson. 1982. Bounding techniques for the ultimate pit limit problem. *Proc. 17th Internat. APCOM Symposium*, 263–273.

Caccetta, L., L. M. Giannini. 1988. The generation of minimum search patterns in the optimum design of open pit mines. *AusIMM Bull. Proc.* **293**(5) 57–61.

——, ——. 1985. On bounding techniques for the optimum pit limit problem. *Proc. AusIMM* **290**(4) 87–89.

Coleou, T. 1989. Technical parameterization of reserves for open pit design and mine planning. *Proc. 21st Internat. APCOM Symposium*, 485–494.

Dagdelen, K., D. Francois-Bongarcon. 1982. Towards the complete double parameterization of recovered reserves in open pit mining. *Proc. 17th Internat. APCOM Symposium*, 288–296.

——, T. B. Johnson. 1986. Optimum open pit mine production scheduling by Lagrangian parameterization. *Proc. 19th Internat. APCOM Symposium*, 127–142.

Derigs, U., W. Meier. 1989. Implementing Goldberg's max-flow algorithm—A computational investigation. *ZOR—Methods Model Oper. Res.* **33** 383–403.

Dowd, P. A., A. H. Onur. 1992. Optimising open pit design and sequencing. *Proc. 23rd Internat. APCOM Symposium*, 411–422.

Ford, L. R., Jr., D. R. Fulkerson. 1957. A simple algorithm for finding maximal network flows and an application to the Hitchcock problem. *Canad. J. Math*. **9** 210–218.

Francois-Bongarcon, D., D. Guibal. 1984. Parameterization of optimal designs of an open pit—Beginning of a new phase of research. *Trans. Soc. Mining Engineers of AIME* **274** 1801–1805.

——, ——. 1982. Algorithms for parameterizing reserves under different geometrical constraints. *Proc. 17th Internat. APCOM Symp*., 297–309.

Gallo, G., M. D. Grigoriadis, R. E. Tarjan. 1989. A fast parametric maximum flow algorithm and applications. *SIAM J. Comput.* **18**(1) 30–55.

Garey, M. R., D. S. Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA.

——, ——, L. Stockmeyer. 1976. Some simplified NP-complete graph problems. *Theoret. Comput. Sci.* **1** 237–267.

Giannini, L. M., L. Caccetta, P. Kelsey, S. Carras. 1991. PITOPTIM: A new high speed network flow technique for optimum pit design facilitating rapid sensitivity analysis. *AusIMM Proc.* **2** 57–62.

Goldberg, A. V., R. E. Tarjan. 1988. A new approach to the maximum flow problem. *J. Assoc. Comput. Mach.* **35** 921–940.

——. 1987. Efficient graph algorithms for sequential and parallel computers. Ph.D. Thesis, MIT, Cambridge, MA.

Hochbaum, D. S. 1996. A new-old algorithm for minimum cut on closure graphs. Working paper.

Hochbaum, D. S. 1997. The pseudoflow algorithm: A new algorithm for the maximum flow problem. Working paper, May.

Huttagosol, P., R. Cameron. 1992. A computer design of ultimate pit limit by using transportation algorithm. *Proc. 23rd Internat. APCOM Symp.*, 443–460.

Johnson, T. B. 1968. Optimum open pit mine production scheduling. Ph.D. Thesis, Dept. of IEOR, University of California, Berkeley, CA.

——, W. R. Sharp. 1971. A three-dimensional dynamic programming method for optimal ultimate open pit design. U.S. Bureau of Mines, Report of Investigation 7553.

Kim, Y. C. 1978. Ultimate pit limit design methodologies using computer models—The state of the art. *Mining Engrg.* **30** 1454–1459.

Koborov, S. 1974. Method for determining optimal open pit limits. Rapport Technique ED 74-R-4, Dept. of Mineral Engineering, Ecole Polytechnique de Montreal, Canada, February.

Koenigsberg, E. 1982. The optimum contours of an open pit mine: An application of dynamic programming. *Proc. 17th Internat. APCOM Symp.* 274–287.

Lerchs, H., I. F. Grossmann. 1965. Optimum design of open-pit mines. *Transactions*, *C.I.M.* **LXVIII** 17–24.

Nguyen, Q. C., V. Venkateswaran. 1993. Implementations of the Goldberg-Tarjan maximum flow algorithm. *DIMACS Series in Discrete Math. Theoret. Comput. Sci.* **12** 19–41.

Pana, M. T. 1965. The simulation approach to open pit design. *Proc. 5th APCOM Symp.* Tucson, AZ.

Picard, J. C. 1976. Maximal closure of a graph and applications to combinatorial problems. *Management Sci.* **22** 1268–1272.

Rhys, J. M. W. 1970. A selection problem of shared fixed costs and network flows. *Management Sci.* **17**(3) 200–207.

Robinson, R. H., N. B. Prenn. 1977. An open pit design model. *Proc. 15th Internat. APCOM Symp.* 1–9.

Seymour, F. 1994a. Pit limit parameterization from modified 3D Lerchs-Grossmann algorithm. Working paper.

——. 1994b. Finding the mining sequence and cutoff grade schedule that maximizes net present value. Working paper.

Vallet, R. 1976. Optimisation mathematique de l'exploitation d'une mine a ciel ouvert ou le problem de l'enveloppe. *Ann. Mine de Belgique* February 113–135.

Wang, Q., H. Sevim. 1993. An alternative to parameterization in finding a series of maximum-metal pits for production planning. *Proc. 24th Internat. APCOM Symp.* 168–175.

Whittle, J. 1989. The facts and fallacies of open pit optimization. Working paper.

——, L. I. Rozman. 1992. Open pit design in the 90s. Working paper.

Yegulalp, T. M., A. J. Arias. 1992. A fast algorithm to solve the ultimate pit limit problem. *Proc. 23rd Internat. APCOM Symp.* 391–397.

——, A. J. Arias, P. Muduli, Y. Xi. 1993. New development in ultimate pit limit problem solution methods. *SME Trans.* **294** 1853–1857.

Zhao, Y., Y. C. Kim. 1992. A new optimum pit limit design algorithm. *Proc. 23rd Internat. APCOM Symp.*, 423–434.