NETWORKS
An International Journal
30th Anniversary Special Paper

A New–Old Algorithm for Minimum-cut and Maximum-flow in Closure Graphs

Dorit S. Hochbaum*
Department of Industrial Engineering and Operations Research and Walter A. Haas School of Business, University of California, Berkeley, Berkeley, California 94720-1777

We present an algorithm for solving the minimum-cut problem on closure graphs without maintaining flow values. The algorithm is based on an optimization algorithm for the open-pit mining problem that was presented in 1964 (and published in 1965) by Lerchs and Grossmann. The Lerchs–Grossmann algorithm (LG algorithm) solves the maximum closure which is equivalent to the minimum-cut problem. Yet, it appears substantially different from other algorithms known for solving the minimum-cut problem and does not employ any concept of flow. Instead, it works with sets of nodes that have a natural interpretation in the context of maximum closure in that they have positive total weight and are closed with respect to some subgraph. We describe the LG algorithm and study its features and the new insights it reveals for the maximum-closure problem and the maximum- flow problem. Specifically, we devise a linear time procedure that evaluates a feasible flow corresponding to any iteration of the algorithm. We show that while the LG algorithm is pseudopolynomial, our variant algorithms have complexity of $O(mn \log n)$, where n is the number of nodes and m is the number of arcs in the graph . Modifications of the algorithm allow for efficient sensitivity and parametric analysis also running in time $O(mn \log n)$. © 2001 John Wiley & Sons, Inc.

Keywords: open-pit mining; maximum-flow; maximum-closure; minimum-cut; sensitivity analysis

1. INTRODUCTION

We introduce here a new algorithm for solving the maximum-closure problem, which solves also the minimum-cut and the maximum-flow problems on the associated "closure graph." Closure graphs are a special class of digraphs that include a source and a sink in which only source adjacent arcs and sink adjacent arcs have finite capacities. All other arcs have infinite capacity.

The new algorithm is based on an algorithm devised in 1964 by Lerchs and Grossmann (LG) [49]. The LG algorithm is of great deal of interest for a number of reasons: It has been used by the mining industry for the past three decades; it does not employ any concept of flow, but solves a problem that is identical to the minimum s,t-cut problem[1]; the complexity of the LG algorithm has never been analyzed (although there is a convergence, or finiteness, proof in [49]), and the ideas employed in the LG algorithm are dramatically different from those used for any known minimum-cut (or maximum-closure) algorithms. Indeed, our investigation presented here indicates that the LG algorithm is fundamentally different from existing minimum-cut algorithms and that it leads to new approaches for solving the maximum-flow problem on closure graphs. Our more recent work that evolved from this research demonstrated a new algorithm that solves maximum flow on general graphs and which has a practical run time substantially faster than any other known algorithm for several classes of graphs.

This paper has a dual purpose: It introduces a new approach for solving maximum flow on closure graphs and it addresses the need of the mining industry to solve effectively and efficiently the open-pit mining problem, using an algorithm that is widely familiar and in common use in the industry.

Received May 2000; accepted February 2001
Correspondence to: D.S. Hochbaum; (e-mail: hochbaum@ieor.berkeley.edu
Contract grant sponsor: NSF; grant nos. DMI-9713482 and DMI-0084857
*UC Regents. Research supported in part by NSF awards No. DMI-9713482 and DMI-0084857.
© 2001 John Wiley & Sons, Inc.

Our study of the mining problem was motivated by being consulted in 1990 about the mining industry's computational difficulties with planning operations in terms of the running time of commercial software. This sentiment was, and still is, manifested in the considerable attention to computational efficiency in the mining literature. Although heuristic approaches and artificial intelligence techniques are usually reserved for NP-hard problems, such approaches are commonplace in promotional materials for mining commercial software [12, 61]. One typical statement [6] reads

> A number of optimization techniques are available for determining this optimal contour. Unfortunately, direct applications of these techniques to large ore bodies cause considerable computational difficulties.

Still, the mining problem is known to be equivalent to the minimum-cut problem which is known to be solvable in polynomial time. (For the proof of this equivalence, the reader is referred to Section 2.2.)

Another aspect of concern to the mining industry is that planning calls for sensitivity analysis of different scenarios under different parameters values such as the value of the commodity. Yet, known efficient parametric analysis network flow techniques are not utilized. Instead, the common approach for addressing the sensitivity analysis, as described in the literature, is to call repeatedly for the optimization algorithm.

We now describe briefly the mining problem addressed here: 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 a deeper and deeper pit is formed until the operation terminates. The final contour of this pit mine is determined before the mining operation begins. To design the optimal pit—one that maximizes profit—the entire area is divided into a 3-dimensional grid of blocks and the value of the ore in each block is estimated using geological information obtained from drill cores. Each block has a weight associated with it, representing the value of the ore in it, minus the cost involved in removing the block. While trying to maximize the total weight of the blocks to be extracted, there are also contour constraints that have to be observed. These constraints specify the slope requirements of the pit and precedence constraints that prevent blocks from being mined before others in a layer on top of them. Subject to these constraints, the objective is to mine the most profitable set of blocks.

The mining problem can be modeled on a directed graph $G = (V, A)$. Each block i corresponds to a node with a weight b_i representing the net value of the individual block. This net value is the assessed value of the ore in that block, from which the cost of extracting that block (alone) is deducted. There is a directed arc from node i to node j if block i cannot be extracted before block j, which is in a layer right above block i. This precedence relationship (which is the reverse of the standard direction for precedence constraints) is determined by the "engineering slope requirements." Suppose that block i cannot be extracted before block j and block j cannot be extracted before block k. By transitivity, this implies that block i cannot be extracted before block k. We chose in this presentation not to include the arc from i to k in the graph and the existence of a directed path from i to k implies the precedence relation. Including only arcs from immediate predecessors reduces the total number of arcs in the graph compared to the alternative approach. This issue is further discussed in Section 2.5.

The decision on which blocks to extract to maximize profit is equivalent to finding a maximum-weight closed set of nodes, where a set of nodes is closed if it contains all successors of the nodes in the set. Such a set is called a maximum *closure* of G. The mining problem is also known as the open-pit mining, the optimal contour, optimal pit problem, or the *maximum-closure problem*.

Problem Name: <u>*Maximum-closure Problem*</u>
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 $\sum_{i \in V'} b_i$ is maximum.*

As noted earlier (and shown in detail in the next section), the open-pit mining problem is solvable by maximum flow on an associated graph. The most efficient maximum-flow algorithms known to date are variants of the push-relabel algorithm described in [24] and [45], of complexities $O(mn \log \frac{n^2}{m})$ and $O(mn + n^{2+\epsilon})$ for any $\epsilon > 0$, respectively, for m and n representing the number of arcs and nodes in the graph. In another paper [34], we verified empirically that for mining problem data the push-relabel algorithm is substantially more efficient than the (original) LG algorithm.

Our contributions here include: A presentation of the LG algorithm with insights into the mechanisms that make it work; linking the algorithm's certificate of optimality to a feasible flow; a complexity analysis of the LG algorithm demonstrating it is pseudopolynomial (presented as a convergence or finiteness proof in [49]); applying scaling techniques and other variants that improve the algorithm's performance; a strongly polynomial time variant of the algorithm; a parametric version of the LG algorithm that has the same complexity as a single run; and practical implementation suggestions.

The paper is organized as follows: We start with preliminaries that include the formulation of the open-pit mining problem as a maximum-closure problem and then, equivalently, as a minimum-cut problem. We describe the relationship between maximum flow and minimum cut and between flow in closure graphs and in general graphs. A literature review focusing on the LG algorithm and the open-pit mining problem is given in Section 3. We then describe the LG algorithm and analyze its correctness, complexity, and some distinguishing properties. In Section 5, we describe an algorithm that takes the output of the LG algorithm and assigns flows to the arcs of the closure graph in linear time. We also comment on the relationship between the LG algorithm and the push-relabel algorithm for maximum flow.

Next, we propose in Section 6 complexity improvements of the LG algorithm and two variants. One variant runs in polynomial time that depends on the value of the largest node weight, $||b||$, and the other is strongly polynomial, of complexities $O(mn \log_2 ||b||)$ and $O(mn \log n)$, respectively. Section 7 describes several variants of the LG algorithm for parametric analysis of the maximum-flow problem on closure graphs.

2. THE MAXIMUM-CLOSURE AND MINIMUM-CUT PROBLEMS

2.1. Notation

For $P, Q \subset V$, the set of arcs going from P to Q is denoted by, $(P, Q) = \{(u, v) \in A | u \in P \text{ and } v \in Q\}$. Let the capacity of arc (u, v) be denoted by c_{uv} or $c(u, v)$. For $P, Q \subset V$, $P \cap Q = \emptyset$, the capacity of the cut separating P from Q is, $C(P, Q) = \sum_{(u,v)\in(P,Q)} c_{uv}$. For $S \subseteq V$, let $\bar{S} = V \setminus S$.

For a graph $G = (V, A)$, we denote the number of arcs by $m = |A|$ and the number of nodes by $n = |V|$.

A graph that includes distinguished source and sink nodes is referred to as an s, t graph.

An arc (u, v) of an unspecified direction is referred to as *edge* $[u, v]$. Let $(v_1, v_2, \ldots, v_k)$ denote a *directed* path from v_1 to v_k, that is, $(v_1, v_2), \ldots, (v_{k-1}, v_k) \in A$. Let $[v_1, v_2, \ldots, v_k]$ denote an *undirected* path from v_1 to v_k, that is, $[v_1, v_2], \ldots, [v_{k-1}, v_k] \in A$.

Each node $v \in V$ has a weight b_v (benefit, mass) associated with it, which could be any integer (possibly negative). (Our strongly polynomial algorithms apply also when the node weights are real.) For $P \subseteq V$, the total weight of the set P is $b(P) = \sum_{v\in P} b_v$. The set of nodes with positive weights is denoted by V^+, and the set of nodes with negative weights, by V^-. The total sum of positive weights in the graph is $M^+ = b(V^+)$ and the total sum of negative weights is $M^- = b(V^-)$.

A *successor* of a node v is a node u such that there is a directed path from v to u. An *immediate successor* of a node v is a node u such that $(v, u) \in A$.

2.2. Reduction of Open-pit Mining to the Minimum-cut Problem

In formulating the open-pit mining problem, each block is represented by a node in a graph and the slope requirements are represented by precedence relationships described by the set of arcs A in the graph. The integer programming formulation of the problem reveals its minimum-cut structure: Let x_j be a binary variable that is 1 if node j is in the closure and 0 otherwise. Let b_j be the weight of the node or the net benefit derived from the corresponding block.

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

Each row of the constraint matrix has one 1 and one -1, a structure indicating that the matrix is totally unimodular. More specifically, it indicates that the problem is a dual of a flow problem.

LG noted in passing that the maximum-closure problem "can be transformed into a network flow problem" (p. 19). Johnson [40] was the first to recognize formally the relationship between the maximum-closure problem and the maximum-flow problem. This he did by reducing the closure problem to another closure problem on bipartite graphs which was, in turn, solved as a transportation problem. 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. Johnson observed that this latter problem, frequently referred to as the *selection problem*, can be solved as a transportation problem. In fact, the problem is equivalent to deciding the *feasibility* of a transportation problem, which is an easier problem than is optimally solving the transportation problem. The bipartite closure problem was independently shown to be solvable by a minimum-cut algorithm by Rhys [56] and Balinski [3]. Other aspects of the bipartition reduction are reviewed in Section 2.5.

Picard [54] demonstrated formally that a minimum-cut algorithm on a related s, t graph solves the maximum-closure problem. The *related graph* G_{st} is constructed by adding source and sink nodes, s and t, $V_{st} = V \cup \{s, t\}$. Let $V^+ = \{j \in V | b_j > 0\}$, and $V^- = \{j \in V | b_j < 0\}$. The set of arcs in the related graph, A_{st}, is the set A appended by arcs $\{(s, v) | v \in V^+\} \cup \{(v, t) | v \in V^-\}$. The capacity of all arcs in A is set to infinity, and the capacity of all arcs adjacent to the source or sink is $|b_v|$:

$$c(u, v) = \begin{cases} \infty & (u, v) \in A \\ b_v & u = s, \ v \in V^+ \\ -b_u & u \in V^-, v = t. \end{cases}$$

In the related graph, the source set of a minimum-cut separating s from t, excluding s, is also a maximum closure in the graph G. The proof of this fact is repeated

here for completeness. The source set is obviously closed as the cut must be finite and thus cannot include any arcs of A. Now, let $(\{s\} \cup S, \{t\} \cup \bar{S})$ be a finite cut in the graph $G_{st} = (V_{st}, A_{st})$. The capacity of the cut $C(\{s\} \cup S, \{t\} \cup \bar{S})$ is

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

Thus, the capacity of the cut is equal to a constant, $b(V^+)$ —the sum of all positive weights—minus the sum of weights of the nodes in the set S, which is closed. Hence, minimizing the cut capacity is equivalent to maximizing the total sum of weights of nodes in the source set of the cut. An example of a closure graph related to a maximum-closure problem, where the source set of the minimum-cut defines a maximum closed set, is depicted in Figure 1.

2.3. *Max Flow Versus Min Cut*

In 1957, Ford and Fulkerson [17] established that the value of the maximum flow in a graph is equal to the value of the minimum s, t-cut. These two problems are linear programming duals of each other, yet there are differences in terms of "information content." Whereas the minimum cut specifies only a partition of the nodes into two subsets, the maximum flow assigns flow to each arc. (For another perspective on the same issue, note that the minimum-cut formulation as the dual of maximum flow is highly degenerate.) The only method known to date for solving the minimum-cut problem requires finding a maximum flow first and then recovering the cut partition by finding the set of nodes reachable from the source in the residual graph (or reachable from the sink in the reverse residual graph). That set is the source set of the cut, and the recovery can be done in linear time in the number of arcs.

On the other hand, given a minimum-cut, there is no efficient way of recovering the flow values on each arc

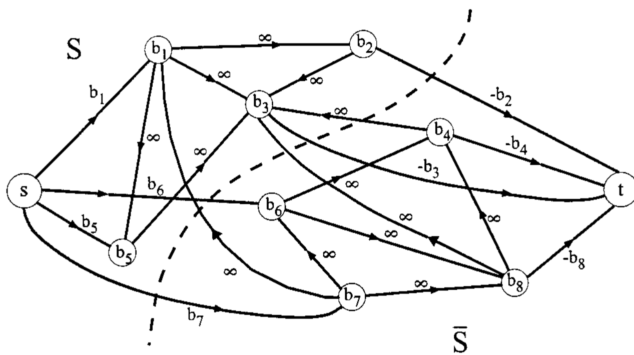

FIG. 1. A maximum closure and minimum cut in closure graphs.

other than utilizing a process equivalent to solving the maximum-flow problem from scratch. The only information given by the minimum cut is the value of the maximum flow and the fact that the arcs on the cut are saturated.

Remark. *The problem that we refer to here as the minimum-cut problem is the minimum s, t-cut problem. In terms of the close link to maximum flow, the minimum s, t-cut problem differs from the minimum 2-cut problem. In fact, the latter problem is only "easier" than the former. To see that, observe that given an oracle solving the minimum s, t-cut problem a solution to the minimum 2-cut is obtained by making a polynomial number of calls to the oracle. The opposite is not known to be the case. For further evidence, consider the problem of finding a partition into three (3) nonempty subsets so as to minimize the cut separating the 3 subsets, the minimum 3-cut problem, versus the minimum t_1, t_2, t_3-cut problem where each subset in the partition must contain one of the specified nodes t_1, t_2, or t_3. Whereas the minimum 3-cut problem is solvable in polynomial time [27], the minimum t_1, t_2, t_3-cut problem is NP-hard [11].*

As evidence that the 2-cut problem may be easier than is the s, t-cut problem, there are algorithms that solve the minimum 2-cut problem on undirected graphs more efficiently than do any maximum-flow algorithm. Such algorithms were devised by Nagamochi and Ibaraki [51], by Stoer and Wagner [60], and by Karger and Stein [42]. In this paper, the minimum-cut problem is the minimum s, t-cut problem.

The asymmetry between solving the maximum-flow and the minimum-cut problems implies that it may be easier to solve the minimum-cut problem than to solve the maximum-flow problem. Yet, known minimum-cut algorithms either compute first the maximum flow or, more rarely, compute a certificate of optimality different from the flow that can be used for both the minimum-cut and the maximum flow. An example of the latter is the minimum-cut in s, t-planar graphs. These are planar graphs where both source node s and sink node t lie on the exterior face. For this problem, the minimum-cut can be derived by finding the shortest paths in the dual graph from the exterior face to itself. The shortest path labels serve as a certificate of optimality for the cut. Hassin showed, in [30], that the shortest path labels can also be used to compute the maximum flow, in linear time. Thus, the shortest path distance labels form a certificate of optimality for both maximum flow and minimum cut.

This discussion raises an interesting question about the LG algorithm, which solves only the minimum-cut problem: As we show, the structure used at each iteration (called a *normalized tree* and defined in Section 4.3), contains sufficient information to make it possible to derive the flow in linear time (Section 5). Conversely, given a feasible flow, we found in [32] that it is possible

to construct a corresponding normalized tree in linear time. (The sense in which the tree corresponds to the flow is not entirely evident from the discussion in this paper, although it is fully explained in [32].)

Yet another still outstanding question is whether, when given a cut, one can construct the corresponding normalized tree? This would result in the construction of flow from cut information with the running time required to construct the tree.

2.4. *Applications of Maximum Flow Minimum Cut on Closure Graphs*

There are numerous applications of maximum closure in addition to the open-pit mining problem. Among the obvious applications are the determination of the value of a maximum-flow problem and, thus, the feasibility as well. Another application is the *feasibility* of a transportation problem. That condition for feasibility was stated, for instance, by Gale [19] for transportation problems with the sum of supplies equal to the sum of demands: For every set of supply nodes S, and all the demand nodes $N(S)$ reachable from them (which are also neighbors and immediate successors), the total demand of $N(S)$ is at least as great as the supply of S:

$$b(S) \leqq |b(N(S))|.$$

We note first that the set $S \cup N(S)$ is a closed set in the bipartite transportation graph, and, conversely, every closed set in this graph is of the form $S \cup N(S)$ for some set of supply nodes S. We next assign a weight to each node equal to its supply (negative for demand). The maximum-closure problem optimal value in the resulting node weighted graph has to be of value 0 for the condition to be satisfied.

The maximum-closure problem can also be used to solve the *feasibility* of the minimum-cost network-flow problem in a general graph. To see that, notice that for a network with supplies, demands, and finite capacities there is a well-known transformation setting all the capacity lower bounds to 0 and the upper bounds to infinity (see, e.g., [1]). With this transformation, we get a graph with node weights (supplies/demands) and infinite capacity arcs. The feasibility necessary and sufficient condition is that for any (closed) subset of nodes, D, the net supply in D does not exceed the capacity of the cut separating the set D from the rest of the graph:

$$b(D) \leqq C(D, \bar{D}).$$

That cut is of value 0 for a closed-set D (and infinite otherwise). So the problem is to demonstrate that the value of the maximum closed set is 0, a value that is achieved for the empty set and for the set of all nodes.

For a glimpse at the range of applications unique to closure graphs, we provide the following (partial) list:

1. Optimal policies for setting repair kits and spare machines [50].
2. Scheduling of fabrication and assembly operations under due dates in a job shop [16].
3. Optimal weapon allocation against layered defenses [52].
4. Tournament team elimination [58, 38].
5. Storing large database records in two storage levels [15].
6. Maximum-density subgraph, which is a subgraph of a maximum ratio of edges to nodes [22].
7. The strength of a network, defined as the minimum ratio of a cut capacity to the number of connected components it creates (which is the total number of components minus 1) [10].
8. Bipartite matching, independent set on bipartite graphs, and vertex cover on bipartite graphs [48].
9. Optimization of nonlinear polynomial functions over box constraints, when the polynomials have the "bipartition property" [31] or when they are "unate" functions [29].
10. Solving monotone integer programs with two variables per inequality [36].
11. 2-approximations for problems with two variables per inequality [37, 33].

For additional discussion of minimum-cut applications, the reader is referred to a classic paper by Picard and Queyranne [55].

2.5. *Bipartite Implementation*

Any closure graph has a corresponding bipartite graph on which the solution to the maximum-closure problem is identical to the solution on the original closure graph. Indeed, the maximum closure problem can be equivalently stated with a set of arcs going only between V^+ and V^-. Specifically, it suffices to solve the maximum-closure problem in the bipartite graph $B_G = (V^+ \cup V^-, A^{+/-})$, where

$$A^{+/-} = \{(i,j) | i \in V^+, j \in V^-, \text{ and there is a directed path from } i \text{ to } j \text{ in } (V, A)\}.$$

The equivalence of the general graph version of the problem and the bipartite closure is proved next:

Lemma 1. *The optimal solution to a maximum-closure problem in G has the same value as that of the optimal maximum-closure solution on the corresponding bipartite graph B_G.*

Proof. Consider an optimal closed-set D in the graph G. Let D^+ be the set of nodes in D with positive weight, $D^+ = D \cap V^+$. $D^+ \cup N(D^+)$ is a closed set in the bipartite graph B_G. We claim that $b(D^+ \cup N(D^+)) = b(D)$. First, $D^+ \cup N(D^+) \subseteq D$ since $N(D^+)$ is clearly in the closure of D^+. If the containment is strict, then there exists a node $v \in D \setminus \{D^+ \cup N(D^+)\}$. Obviously, $v \notin V^+$, else D could not be optimal, so the weight of v is either negative

or 0. If $b(v) < 0$, then it cannot belong to a maximum closed set in G, since it is not a successor of any node of positive weight in D and thus can be removed along with its predecessors, leaving a set that is closed and of strictly larger weight in G. Therefore, $b(v) = 0$ and $b(D) = b(D^+ \cup N(D^+))$.

Now, let $D_1 \cup D_2$ be an optimal closed set in B_G with $D_1 \subseteq V^+$ and $D_2 \subseteq V^-$. Then, because the set is closed, $D_2 \subseteq N(D_1)$, and due to the optimality, $D_2 = N(D_1)$. If there are positive weight nodes in the closure of D_1 in G that are not included in D_1, then these could be added while strictly increasing the weight of $D_1 \cup D_2$ in B_G, contradicting the optimality of this set. Thus, $D_1 \cup D_2$ is also a closed set in G.

Note that the optimal closed sets in G and B_G may differ if there are zero-weight nodes. To recover those, we find the closure of the maximum closed set in B_G in the graph G. ■

The advantage of the bipartite representation of the closure problem is that problems on bipartite graphs may be solved more efficiently than are problems of the same size on general graphs, (e.g., [28]). On the negative side, the bipartite graph contains more arcs than does the corresponding general graph, as there is an arc between a node and *all* its negative-weight successors, even if not immediate successors. This trade-off appears to tip in favor of the nonbipartite formulation in the empirical study of the LG algorithm and several push-relabel algorithm implementations [34].

3. LITERATURE REVIEW

We assume that the readers are familiar with the vast literature on maximum-flow algorithms. The review here focuses only on the literature directly related to the LG algorithm and the open-pit mining problem. Even with this limited focus, the body of literature is extensive. We provided an expanded review of the literature in [34].

A number of algorithms have been developed over the years to solve the open-pit mining problem. These include the algorithms by Lerchs and Grossmann [49], Johnson and Sharp [41], Robinson and Prenn [57], Koborov [46], Koenigsberg [47], Dowd and Onur [13], Zhao and Kim [65], and Huttagosol and Cameron [39]. Among these, only the LG algorithm and the network simplex solve the problem optimally. The other algorithms are either variants of the LG algorithm or heuristic algorithms that do not guarantee an optimal solution. With Picard's proof that maximum-closure problems are reducible to the minimum-cut problem (Section 2.2), it is possible to apply any of the known efficient maximum-flow algorithms. Still, in the mining literature, there were relatively few studies of optimizing algorithms other than the LG algorithm until recently.

Until the early 1980s, heuristic algorithms were widely used in the mining industry because they execute faster and are conceptually simpler than are optimizing algorithms (which deliver an optimal solution). With advances in computer technology, optimizing algorithms became commonplace. Whittle Programming Pty. Ltd.'s commercial open-pit optimization package *Lerchs and Grossman 3-D*, which uses the LG algorithm, has become the most popular package in the mining industry, with a 4-D version to support sensitivity analysis. Other commercial packages include MULTIPIT, which uses Francois-Bongarcon and Guibal's algorithm [18], and PITOPTIM, which uses a maximum-flow algorithm [21].

As Whittle pointed out in [64], 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 for a typical mine. An actual example given in [64] showed that the difference was 5%. The heuristics that were used for the example were not specified.

3.1. *Heuristic Algorithms*

Most heuristic algorithms consider cones—formed by a "base" block and all its successor blocks above it. The moving cone method (described, e.g., in [53]) 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. The algorithm terminates when no more cones can be added. It is easy to devise an example where no such cone exists while there still is an optimal solution of arbitrarily high value, thus proving that such an algorithm is not optimal.

Robinson and Prenn's algorithm [57] checks, in turn, each cone that has an ore block at 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 [46] is a variant of this idea. Dowd and Onur [13] 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 is their implementation of the LG algorithm.

Other heuristic algorithms include the dynamic programming methods of Johnson and Sharp [41] and Koenigsberg [47]. Several additional heuristic algorithms were discussed by Kim [43] and Koenigsberg [47].

3.2. *Optimizing Algorithms*

One variant of the LG algorithm was developed by Zhao and Kim [65]. As in the LG algorithm, the blocks in the model are partitioned into subsets. The main difference between the two algorithms is 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 is LG's algorithm. No direct computational comparison between the two algorithms is provided in the paper.

Vallet [62] proposed an interesting variant of the LG algorithm. Rather than partitioning the nodes into strong and weak (which are sets of nodes of total positive weight and negative weight, respectively—see Section 4 for definitions), Vallet classifies the sets based on the ratio B/V, where B is the total weight, and V, the volume of the set. He referred to this ratio as the strength. If a block in a set with higher strength lies beneath a block in a set with lower strength, then the two sets are merged and possibly regrouped.

Other researchers made use of maximum-flow algorithms to solve the open-pit mining problem. Giannini et al. [21] developed the software package PITOPTIM which uses Dinic's maximum-flow algorithm for computing the optimal pit contour. Computation times for their implementation are given in [21], but there are no comparisons with other algorithms.

Huttagosol and Cameron [39] formulated the open-pit mining problem as a *transportation problem*. This is effectively the bipartite reduction described earlier in Section 2.5. Huttagosol and Cameron proposed using the network simplex method for solving the problem. Their computational experiments showed that the network simplex is slower than is the LG algorithm. This is the only reported computational comparison of LG to the network simplex that we know of.

3.3. *Parametric Analysis*

Parametric 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. To perform parametric analysis, the common approach is to run the LG algorithm for a monotone sequence of parameter values, while contracting the maximum closed set found up to that point and restarting the algorithm on the contracted graph. (The correctness of such a procedure follows from the monotonicity of the closed set, as discussed in more detail in Section 7.) We studied in [34] the performance of such an algorithm as compared to a parametric analysis implementation which maintains the certificate of optimality computed for one parameter value as a starting point for the computation for the next parameter value. The result of this experimentation, reported in [34], is that avoiding the restart leads to considerably better running time. This and additional literature on parametric analysis in open-pit mining were reported in [34].

The concept of *parameterization* is frequently used by the mining industry. This concept, although not immediately formulated as standard parametric analysis, is shown in Section 7.2.3. to be derived as a by-product of such an analysis.

Let B_1 denote the total benefit of all the blocks in pit P_1, V_1 denote the volume of pit P_1, and B_i and V_i denote the total benefit and volume, respectively, of pit $(P_i \setminus P_{i-1})$, $i = 2, \dots, q$. In other words, pit P_1 has the highest benefit-to-volume ratio, while P_q has the lowest benefit-to-volume ratio.

Definition. *The computation of a series of nested pits* $P_1, P_2, \dots, P_q$ *such that* $B_1/V_1 > B_2/V_2 > \cdots > B_q/V_q$, *is called* **parameterization**.

The rationale for parameterization is that mining the most valuable ore rock as early as possible would maximize the net present value (NPV). Therefore, the outcome of the parameterization is used to generate a mining schedule.

Additional literature on open-pit mining algorithms, on preprocessing techniques, and on experimental studies was reviewed in [34].

4. THE LG ALGORITHM

4.1. *The Extended Network and Additional Definitions*

Let $G = (V, A)$ be a node-weighted graph, and let G_{st} be the related closure graph (defined in Section 2.2).

Given a rooted tree, T, T_v is the subtree suspended from node v that contains all the descendants of v in T. $T_{[v,p(v)]} = T_v$ is the subtree suspended from the edge $[v, p(v)]$ (we use the wording *edge* to emphasize that the direction of the arc is immaterial to the discussion). An immediate descendant of a node v, a *child* of v, is denoted by $ch(v)$, and the unique immediate ancestor of a node v, the *parent* of v, is denoted by $p(v)$.

A node is said to be at *level* ℓ in a rooted tree if it is at a distance of ℓ edges (arcs) from the root.

We define a graph called the *extended network*, G^{ext}. The extended network is the graph G appended by a root node r and with arcs going from r to all nodes of V, $G^{\text{ext}} = (V \cup \{r\}, A^{\text{ext}})$. Note that a tree in the extended graph is a forest in $G = (V, A)$.

Given a spanning tree in the extended graph rooted at r, a child v of r in r defines a subtree, T_v, in G to which we refer as a *branch*. That child of r, v, is the root of that respective branch. See Figure 2 for such a tree, where r_1, r_2, r_3 are roots of branches.

Let $e = [p, q]$ be an edge (the arc (p, q) or (q, p)) in a tree such that $p = p(q)$ is the parent of q. The edge e is said to *define* T_q, the subtree rooted at q, and to *support*

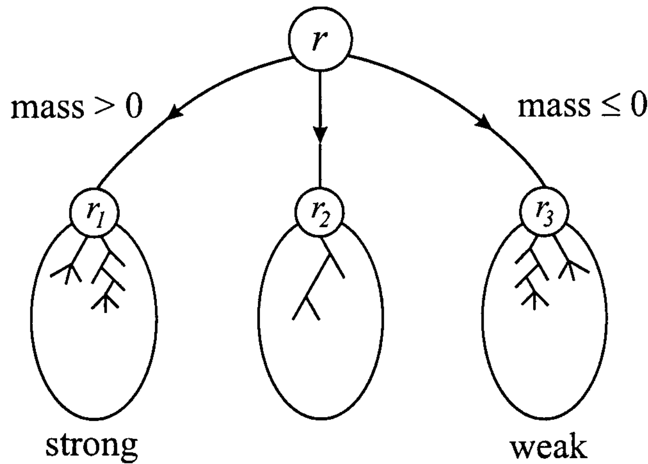

FIG. 2. A normalized tree with three branches. Each r_i is a root of a branch.

its *mass*, which is the sum of weights of nodes in T_q, $b(T_q)$, also denoted by M_q.

Let M_e denote the mass of an edge e, and with some ambiguity, we also denote the mass supported by a node k by M_k. Note that the mass supported by a node is not the same as its weight (or benefit). The mass supported by a node depends on the particular tree structure, as it is the *sum* of weights of all nodes in the subtree rooted at the node.

4.2. Overview

As will be shown, the LG algorithm is a dual algorithm for the maximum-closure problem that works with a superoptimal solution to obtain feasibility. Its relationship to maximum flow is thus rather roundabout—it is a dual algorithm for the maximum-closure problem, which is itself a complement of the minimum-cut problem (its objective value plus the cut capacity add up to a constant, M^+). The minimum-cut problem is, in turn, the dual of the maximum-flow problem.

One interpretation of the LG algorithm is that it identifies in the closure graph a spanning forest with two types of trees: One type is the *strong* trees which are trees with a set of nodes spanned, D, such that the sum of positive weights (which are also the capacity of the cut $C(\{s\}, D)$) is greater than the sum of the absolute values of the negative weights, $C(\{s\}, D) > C(D, \{t\})$ or have total node weight that is positive. The other trees in the forest are called *weak*. The strong trees form a candidate set for the source set of the cut. That set is not necessarily closed, but its total weight can only exceed the value of the maximum closure—a superoptimal solution. (See Lemma 2 for a proof of the superoptimality property.)

4.3. Normalized Trees

At each iteration, the LG algorithm creates a spanning tree rooted at r in the extended graph, called a *normalized tree*, with the property that a maximum closed set in the tree is easily identifiable.

An arc of a spanning tree rooted at r in the extended graph either points toward the root (upward) or points away from the root (downward). To distinguish the orientation of the arcs with respect to the root in a particular tree, [49] used the notation of a p-edge and an m-edge, where p and m stand for plus and minus. We choose the more intuitive terminology of downward and upward with respect to the root position at the top of the tree, as in Figure 2.

Definition. *A downward arc is* **strong** *if it supports a mass that is strictly positive. An upward arc is* **strong** *if it supports a mass that is zero or negative. Arcs that are not strong are said to be* **weak**.

Definition. *A branch is* **strong** *if the arc that links it to r is strong; otherwise, it is* **weak**. *The nodes of a strong (weak) branch are strong (weak) nodes.*

Since all arcs adjacent to the root point away from the root, such arcs are downward arcs. They are strong if the corresponding mass—or total weight—of the branch supported by that arc is positive and weak otherwise.

Definition. *A spanning tree rooted in r in the extended graph is* **normalized** *if the only strong arcs it contains are adjacent to the root r.*

A normalized tree always exists. For instance, a standard normalized tree for a graph G^{ext} with set of weights b is $T = T_0(G, b) = (V \cup \{r\}, A(T))$ for $A(T) = \{(r, j) | j \in V\}$, $M_{(r,j)} = b_j$. This normalized tree has no arcs that are not adjacent to the root and is thus obviously strong.

Let a normalized tree be $T = (V \cup \{r\}, A(T))$, and let $A^T = A \cap A(T)$ denote the arcs of the tree in G that exclude the root adjacent arcs.

With a slight abuse of terminology, we also refer to the forest (V, A^T) in G as a normalized tree.

To prove the validity of the algorithm, LG showed that the set of strong nodes of a normalized tree is its maximum closed set. This proof, however, is not necessary here for proving the correctness of the algorithm since we establish the existence of a feasible flow associated with each iteration of the LG algorithm (see Section 5). That flow value is equal to the value of the cut defined by the closed set at the termination of the algorithm, thus proving that it is a minimum cut. The lemma is nevertheless useful in enhancing the understanding of the properties of normalized trees.

Let S be the set of all strong nodes.

Lemma 2. *Given a normalized tree,* (V, A^T),
(i) The union of the strong branches S is a maximum closed set in (V, A^T),
(ii) The weight of the strong nodes can be only greater than the maximum-closure value in $G = (V, A)$.

Proof.
(i) Every single branch, and, in particular, the union of strong branches, is obviously a closed set in (V, A^T) as the tree contains no arcs of A between the branches.

We now show that any proper subset of a strong branch must be included in a maximum closed set and that any subset of a weak branch cannot increase the weight of a closed set in (V, A^T).

We first prove that eliminating any proper subset of a strong branch from S can only reduce the weight of the closure. A proper subset of a branch may either contain the root of the branch or contain all descendants of a node in the branch, or it may be obtained by removing from the branch a sequence of subsets, each a subtree rooted at some node of the branch, and at most one containing the root of the branch. It is therefore sufficient to show that removing a proper subset containing the root or a proper subset containing all descendants of a certain node will only reduce the weight value of the strong nodes.

Let a strong branch B be partitioned into B_1 and B_2 with the root of the branch contained in B_1, and B_2 is a subtree rooted in an arc a connecting B_1 to B_2. The mass of B is equal to the sum of the masses of B_1 and B_2, $b(B) = b(B_1) + b(B_2)$.

Consider the four cases when B_1 is removed or B_2 is removed and for each one of these cases when a is directed downward or directed upward. Figure 3 depicts this schematic partition.

Suppose that a is directed downward: Arc a then must support nonpositive mass (else it is strong). We cannot remove B_2 although its mass is nonpositive, since that will violate the closure of B_1. Removing B_1 can only reduce the total weight of the set since B has positive mass and B_2 has negative mass.

Suppose that a is directed upward: Then, a must support a positive mass. Removing B_1 will violate the closure of B_2 and removing B_2 will strictly decrease the weight of the branch by the mass of a.

Similarly, it is argued that no subset of a weak branch would contribute to the weight of the closure and can only decrease it. A weak subset of nodes can have a 0 contribution to the total weight of the closure. Thus, the set of strong nodes S is a *minimal* source set among all minimum-cuts.
(ii) The weight of a maximum closed set in a graph can only go up as arcs are being removed from the graph. That is so since the removal of arcs relaxes some of the precedence constraints. Therefore, a maximum closed set in (V, A^T) can be only of greater value than the maximum closure in (V, A), as $A^T \subseteq A$. ■

Corollary 1. *Any proper subset of the strong nodes is not a maximum closure in* (V, A^T).

As a result, at optimum, the set of strong nodes is a *minimal* maximum-closure set.

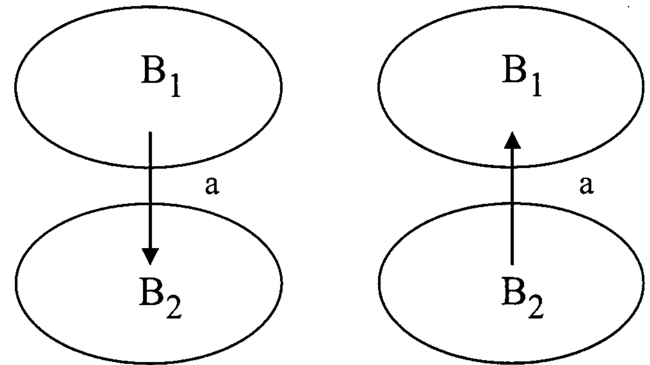

FIG. 3. Branch partition.

4.4. *The Description of the LG Algorithm*

Each iteration of the algorithm consists of identifying an infeasibility in the form of an arc from a strong node s to a weak node w. (Note that the algorithm works in a graph without source and sink, thus, we can use the notation of s for a strong node without risk of ambiguity.) When such an arc is found, the strong and weak branches containing s and w, S and W, are merged. The tree is normalized by "splitting" or "renormalizing" arcs that have become strong as a result of the merger.

An iteration involves the operations:
Merger—Adding an arc (s, w) from a strong branch S to a weak branch W and removing the arc from the root of the branch S to the root r. Consequently, the strong branch becomes suspended from the strong node s and the masses supported by arcs along the path from the root of the strong branch to the merger node s are exactly the masses of the complement set of nodes to the ones previously supported. The status of the two branches before and after a merger is depicted in Figure 4.
Mass updates—Only masses of arcs along the merger path are modified. Arcs along the section of the path within S now support the complement mass within S—that is, the mass of the set of nodes in S complementing the set supported prior to the merger; the merger arc now supports the total weight of the strong branch, and within the weak branch, each arc along the path from the weak node w to the root of W supports, in addition to the previous mass, also the mass of the nodes of the strong branch.
Renormalization—With the update of the masses, some arcs become strong. Such arcs are removed and their subtrees are reattached to r as separate strong branches. A fact that will be shown in Lemma 3 is that only downward arcs can become strong. We will use this property in the algorithm's statement.

We let the generic merger arc be (s, w), where s is a strong node. The input to the algorithm is a graph and an initial (normalized) tree.

Procedure LG [$G = (V, A)$, $b_j\ \forall j \in V$, T]
$V_S^{(T)} = \{v \in V | v \text{ is strong in } T\}$.
While $(V_S^{(T)}, V \setminus V_S^{(T)}) \neq \emptyset$, **do**
 `select` $(s, w) \in (V_S^{(T)}, V \setminus V_S^{(T)})$.
 Let r_s be the root of the branch containing s
 and r_w be the root of the branch containing w.
 {*Merger*} $T \leftarrow T \setminus (r, r_s) \cup (s, w)$.
 {*Mass update and renormalization*}
 Let $e = [a, b]$, $b = p(a)$, be the next arc
 encountered on the path $[r_s, \ldots, s, w, \ldots, r_w]$.
 (i) If e is on the path $[r_s, \ldots, s]$, **Do**
 If e is a downward arc and $M_{r_s} - M_e > 0$,
 call **Renormalize edge** (T, $b_j\ \forall j \in V$,$[a, b]$).
 Set $M_{(r,a)} = M_{r_s} - M_e$. Set $M_{r_s} \leftarrow M_e$.
 Else, $M_e \leftarrow M_{r_s} - M_e$.
 (ii) If $e = (s, w)$, $M_{(s,w)} \leftarrow M_{r_s}$.
 (iii) If e is on the path $[w, \ldots, r_w]$, **Do**
 If e is downward arc and $M_{r_s} + M_e > 0$,
 call **Renormalize edge** (T, $b_j\ \forall j \in V$,$[a, b]$).
 Set $M_{(r,a)} = M_{r_s} + M_e$. Set $M_{r_s} \leftarrow -M_e$.
 Else, $M_e \leftarrow M_{r_s} - M_e$.
end
Return "$V_S^{(T)}$ is a maximum closed set."

Procedure Renormalize edge (T, $b_j\ \forall j \in V$, $[a, b]$)
Replace $[a, b]$ by (r, a): $T \leftarrow T \setminus [a, b] \cup (r, a)$.
{a and the subtree rooted at a form a separate branch.}
Return T.
end

An example illustrating the procedure LG is given in Figure 5 of the next section.

Prior to proceeding with the complexity analysis of the algorithm, we repeat here Property 3 from [49] which states that when considering removal of strong arcs from the merged tree upward arcs never need to be considered. As it turns out, this property is of interest, but has no effect on the complexity analysis of the algorithm.

Lemma 3. *After a merger, an upward arc is never strong.*

Proof. Note that throughout the algorithm the value of M_{r_s} remains nonnegative.

Consider first an upward arc e on the section of the path $[r_s, \ldots, s]$. Then, prior to the merger, e was a downward arc with mass $M_e \leqq 0$. After the merger, the mass of e is $M_{r_s} - M_e = M_{r_s} + |M_e| > 0$, and, thus, e is a weak arc.

Consider now an upward arc e on the section of the path $[w, \ldots, r_w]$. Prior to the merger, it was a weak arc with mass $M_e > 0$. After the merger, the mass of e is $M_{r_s} + M_e > 0$, and, therefore, it is a weak arc. ∎

As a result of the merger, either some nonempty set of strong nodes becomes weak or a nonempty set of weak nodes becomes strong, but not both:

Lemma 4. *After a merger of a strong branch S with a weak branch W, exactly one of the following two possibilities must occur: Either*
(i) Some strong nodes become weak or
(ii) Some weak nodes become strong,
but not both.

Proof. Consider the position of the last strong arc encountered. If it is on the strong section of the path $[r_s, \ldots, s]$, then node s, T_s, and possibly some other strong nodes become weak. If the last strong arc is on the weak section of the path $[r_w, \ldots, w]$, then some weak nodes become strong, including w and T_w.

Note that to incur no change, the arc (s, w) must be strong. But that arc supports the capacity of S which is positive and is directed toward the root; hence, it is a weak upward arc. (Or using the previous lemma, it cannot be strong as an upward arc.) ∎

The complexity of the algorithm now follows: It is of interest to note that this proof was given in [49] as a proof of the "'finiteness" of the algorithm.

Theorem 1. *Given a node-weighted graph $G = (V, A)$, with b_i the weight of node $i \in V$. The number of iterations of the LG algorithm is bounded by $O(|V| \sum_{i \in V^+} b_i)$, and the complexity per iteration is $O(m)$.*

Proof. We show that when some strong nodes become weak the total weight of the strong nodes strictly decreases, and when some weak nodes become strong, then the weight of the strong nodes does not increase but the number of weak nodes strictly decreases. This implies that between two consecutive decreases of the weight of strong nodes there can be, at most, n iterations without a decrease. As a result, the number of iterations is bounded by the total weight of positive weight nodes times $n = |V|$.

Consider a partition of the strong branch S into two components: the subtree T_s rooted at s and the remainder, $T_{r_s} = S \setminus T_s$. Similarly, consider a partition of W into T_w and $T_{r_w} = W \setminus T_w$ as in Figure 4.

(i) Strong nodes become weak: In this case, all of T_s and some of T_{r_s} must become weak. Let edge $[a, b]$ be the last on the path $[r_s, \ldots, s]$ to become strong (downward arc) after the merger. (If no arc has become strong, then $T_b = \emptyset$.) Now, $b(S) = b(T_b) + b(S \setminus T_b)$. Since $[a, b]$ was a weak upward arc prior to the merger, it supported the positive mass $b(S \setminus T_b)$. Hence, the mass of strong nodes went strictly down from $b(S)$ to $b(T_b)$.

(ii) Weak nodes become strong: A downward arc e on $[w, \ldots, r_w]$ must become strong due to the merger. Let $\bar{M}_e$ be the (nonpositive) mass e supported prior to the merger and M_e afterward. Then, $M_e = \bar{M}_e + M_{S'} > 0$, where S' is a portion of the strong branch that is in the created strong branch with the weak nodes supported by e. Hence, $M_e \leqq M_{S'}$ and the total weight of strong nodes after the merger has not increased. It is possible, however, that the weight has not changed. In that case, note

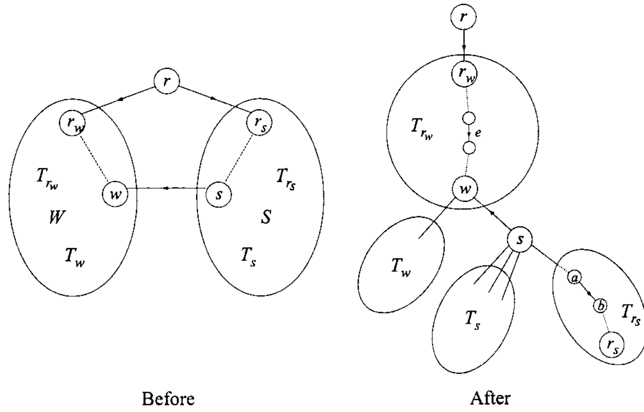

FIG. 4. Before and after merger.

that at least one node— and, in fact, the entire subtree T_w rooted at w—switched status from weak to strong. Hence, there is a strict decrease in the number of weak nodes as required.

The work per iteration consists of finding an arc from a strong node to a weak one and then updating the status of arcs on the path and the status of the nodes in the branch. Finding an arc can be done using, say, breadth-first-search, in $O(m)$. Inverting the strong branch S requires only the reversal of the parent–child relationship along the strong section of the merger path, which takes $O(1)$ operations per arc. Performing mass updates while scanning the merger path for arcs to be renormalized (or split) is accomplished in total time of $O(n)$ (one arithmetic operation of subtraction or addition per each arc on the merger path). Another operation is to maintain the status label of nodes as weak or strong. If implemented in a straightforward manner, this operation requires relabeling all nodes that change their status. The work here would be $O(n)$ per iteration. The total work per iteration is thus $O(m)$. ∎

Remark. *The choice of the arc (s, w) in the* `select` *step of the algorithm is arbitrary. Variations of the algorithm could be based on restricting the choice to some nodes, in particular branches or in some particular order. Two such variations were proposed by [65] and [62]. Our strongly polynomial variant, described in Section 6.4, is based on one specific selection mechanism.*

Remark. *Corollary 2 implies that the set of strong nodes $V_S^{(T)}$ is* `minimal` *in the sense that omitting any subset of strong nodes can only strictly reduce the total weight or violate closure feasibility.*

4.5. Normalizing a Given Tree

For the initial normalized tree in **procedure LG**, we used the "trivial" tree with each branch consisting of a single node. However, any guess of a subset of nodes for a maximum closure can be easily converted into a normalized tree. A collection of disjoint cones, or a good feasible solution (or pit contour), for instance, can be normalized. In fact, any tree in G^{ext} can be normalized using the procedure **Normalize**. The procedure takes as input a spanning tree T in G^{ext} and outputs T as normalized. At any iteration, T' is a subtree of T (pruned and sharing the same root with T) that has not yet been normalized. The procedure works in linear time $O(n)$.

procedure Normalize (T)
begin $M_v = 0\ \forall v \in T$. $T' = T$.
While there exists a leaf node which is not a child of r
  Do
    Select a leaf node in T' which is not a child of r.
    {Compute M_v:} $M_v \leftarrow M_v + b_v$.
    {Update $M_{p(v)}$:}
      For $M_v > 0$: If $(v, p(v))$ is an upward arc,
      $M_{p(v)} \leftarrow M_{p(v)} + M_v$.
      Else, **Renormalize edge** $(T, b_j\ \forall j \in V, [v, p(v)])$.
      For $M_v \leqq 0$: If $(p(v), v)$ is a downward arc,
      $M_{p(v)} \leftarrow M_{p(v)} + M_v$.
      Else, **Renormalize edge** $(T, b_j\ \forall j \in V, [v, p(v)])$.
    {Remove $[v, p(v)]$:} $T' \leftarrow T' \setminus [v, p(v)]$.
end.

To see that at termination the tree T is normalized, observe that at each iteration $T \setminus T'$ induces a collection of branches (trees) that is normalized. This is established by using induction on the size of $T \setminus T'$ and noting that any arc that is strong is renormalized (and thus removed).

5. FLOW ASSIGNMENT: GENERATING A FEASIBLE FLOW ASSOCIATED WITH A NORMALIZED TREE

Given a maximum-closure problem on a graph $G = (V, A)$, we describe here a procedure that takes a normalized tree $T = (V, A^T)$ as input and outputs a feasible flow in the related graph G_{st}. The only arcs that get nonzero flow assigned are the arcs of A^T and the arcs adjacent to the source and sink. The assigned flow has the property that it is maximum among flows restricted to the arcs of the normalized tree and source, sink adjacent arcs, as will be proved in Corollary 4.

Although all arcs of A^T have infinite capacity, we shall say in this section that an arc is *saturated* if the flow on the arc is equal to the absolute value of the mass that it supports in the normalized tree T. We call the absolute value of the mass supported by an arc the *mass-capacity* of the arc. This definition is different from the standard interpretation of saturation that occurs when a flow on an arc meets the capacity upper bound. The flow construction is such that all root adjacent arcs, which are not in the graph G, are not assigned any flow.

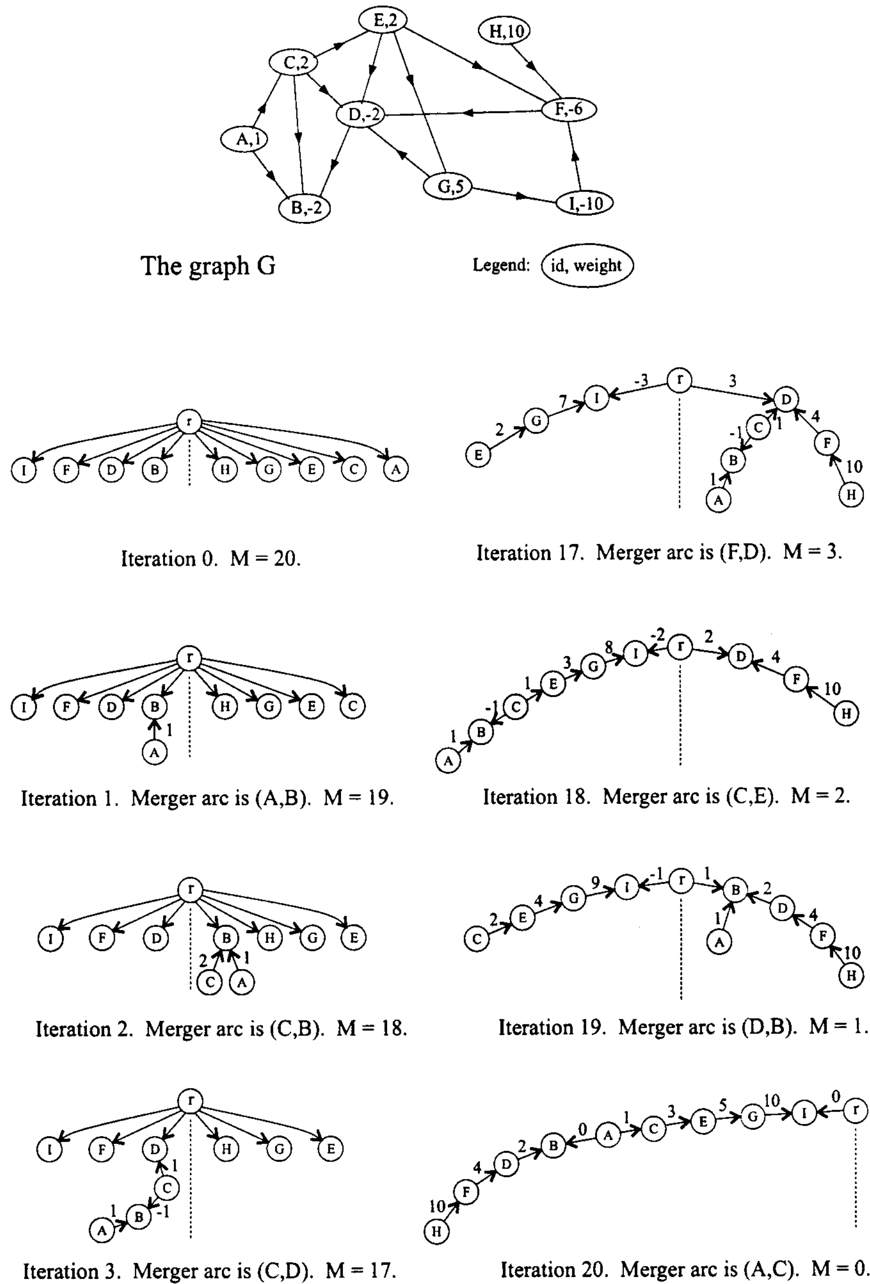

FIG. 5. An example: the dashed line separates the strong branches on the right from the weak ones on the left.

In strong branches, the algorithm saturates all downward arcs not adjacent to the root with their masses (or rather the absolute value of the masses, since downward arcs support nonpositive mass). All negative weight nodes have their entire weight sent as flow to the sink. In weak branches, the algorithm saturates all upward arcs with their masses and all arcs from the source to the positive-weight nodes. The flow has the property that every arc in A^T is assigned flow not exceeding the (absolute value of the) mass it supports.

The idea is to show that this initial setting of flows can be completed to a feasible flow that satisfies flow-balance constraints, that is, let $f(i, j)$ be the flow value assigned to arc (i, j), where $f(s, v) \leqq b_v$ for $v \in V^+$ and $f(v, t) \leqq -b_v$ for $v \in V^-$; then, for each node $v \in V^+$,

$$inflow(v) = \sum_{(i,v)\in A^T} f(i, v) + f(s, v) = \sum_{(v,i)\in A^T} f(v, i) = outflow(v),$$

and for each node $v \in V^-$,

$$inflow(v) = \sum_{(i,v)\in A^T} f(i, v) = \sum_{(v,i)\in A^T} f(v, i) + f(v, t) = outflow(v).$$

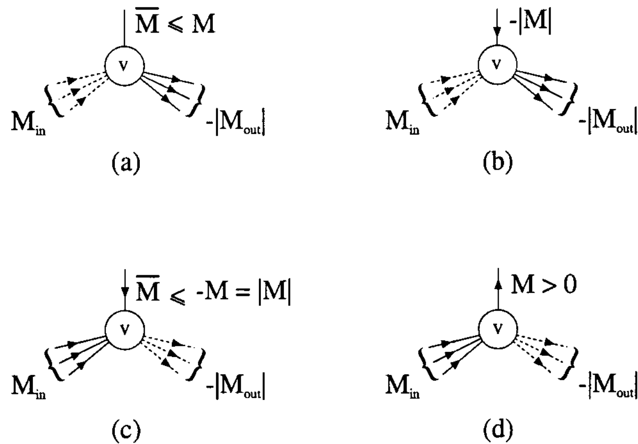

FIG. 6. Flow assignment: (a,b) in strong branches; (c,d) in weak branches.

The proof of the following theorem is constructive and demonstrates how to construct a feasible flow corresponding to a normalized tree in $O(n)$ steps:

Theorem 2. *Every normalized tree has a feasible flow associated with it in the related graph G_{st}. That flow uses only arcs of A^T in addition to source and sink adjacent arcs and saturates the source-adjacent arcs in weak branches and the sink-adjacent arcs in the strong branches.*

Proof. All root adjacent arcs get a flow of value zero preassigned. The assignment of flows for the other arcs is such that no flow exceeds the (absolute value of the) mass capacity of the arc. The constructive procedure is described first for arcs in strong branches and next for arcs in weak branches.

Strong branch: First, all downward arcs are mass-saturated. Next, flows are assigned recursively to upward arcs one level at a time, starting from arcs adjacent to level 1 nodes. Assume that all flows at levels $1, \dots, \ell - 1$ have been assigned.

Consider a node v at level ℓ that has an upward arc connecting it to its parent [Fig. 6 (a)]. That arc supports a positive mass M and a flow of $\bar{M}$ assigned to it, where $0 \leqq \bar{M} \leqq M$. A node at level 1— adjacent to r—falls in this category with $\bar{M} = 0$ assigned to the arc from its parent r (although it is not technically an upward arc, the same discussion applies). Let M_{out} be the sum of masses of downward arcs from v to its children. Since these are negative, the amount of flow that was assigned already to all these arcs is $-M_{\text{out}}$.

Let M_{in} be the sum of masses of upward arcs from the children of v to v. These arcs have not yet been assigned flows. Because of the way masses are calculated,

$$M_{\text{in}} - |M_{\text{out}}| + b_v = M > 0.$$

Hence, the dynamic value of the imbalance satisfies $\mathit{outflow}(v) - \mathit{inflow}(v) = \bar{M} + |M_{\text{out}}| \leqq M_{\text{in}} + b_v$. Therefore, $M_{\text{in}} + b_v$ has sufficient mass capacity to balance the gap between outflow and inflow: Specifically, if b_v is negative, then $f(v,t) = -b_v$. It is then possible to assign flows to the incoming arcs in an arbitrary fashion up to their mass capacity until outflow(v) equals inflow(v) (possibly saturating one arc's mass prior to assigning positive flow to the next). If b_v is positive, then assign in an arbitrary fashion flows to the incoming arcs (up to their mass capacity) and up to b_v units of flow to the arc (s, v). There is sufficient capacity in these arcs to balance the flow.

Consider now a node v that has a downward arc leading to it from its parent. The mass of that arc is $M < 0$ and the flow is $-M$ [Fig. 6 (b)]. Now, the dynamic imbalance between inflow and outflow is

$$\mathit{outflow}(v) - \mathit{inflow}(v) = |M_{\text{out}}| + M = M_{\text{in}} + b_v.$$

If $b_v > 0$, then $f(s,v) = b_v$. If $b_v < 0$, then $f(v,t) = -b_v$. In either case, all remaining incoming arcs are saturated with their masses in order to balance the flow.

Weak branch: Here, all upward arcs are set saturated with their (positive) mass. Consider a node v at level ℓ after all the arcs adjacent to its ancestors were assigned flows. Consider, first, such a node with a downward arc of mass M (nonpositive) connecting it to its parent and a flow of $\bar{M} \leqq |M|$ assigned to that arc [Fig. 6 (c)]. Then,

$$M_{\text{in}} - |M_{\text{out}}| + b_v = -|M|.$$

With this partial assignment, the node has the dynamic imbalance of

$$\begin{aligned} \mathit{inflow}(v) - \mathit{outflow}(v) &= M_{\text{in}} + \bar{M} \leqq M_{\text{in}} + |M| \\ &= |M_{\text{out}}| - b_v. \end{aligned}$$

Hence, $|M_{\text{out}}|$ has sufficient mass capacity to balance the gap between inflow and outflow: If $b_v > 0$, then $f(s,v) = b_v$ and the rest is balanced with the outgoing arcs to level $\ell + 1$. If $b_v < 0$, then $f(v,t)$ is set $\leqq -b_v$ and the balance of the flow is distributed arbitrarily (say, saturating one arc at a time for its absolute mass capacity and leaving the last assigned arc possibly unsaturated) among the outgoing arcs. The case of a level 1 node applies here with $\bar{M} = 0$.

Consider now a node v with an upward arc of positive mass M connecting it to its parent [Fig. 6 (d)]. Then,

$$M_{\text{in}} - |M_{\text{out}}| + b_v = M.$$

The imbalance at v is $M_{\text{in}} - M = |M_{\text{out}}| - b_v$. If $b_v > 0$, then $f(s,v) = b_v$ and we saturate all outgoing arcs to their mass capacity. If $b_v < 0$, then $f(v,t)$ is set $\leqq -b_v$ and the balance of the flow is distributed arbitrarily among the outgoing arcs.

As for the complexity, note that each arc in the tree is considered once without backtracking. There are only

$O(n)$ arcs in the tree; hence, the complexity of the flow assignment is $O(n)$. ▪

Corollary 2. *The feasible flow constructed is a maximum flow in the related graph G_{st}.*

Proof. A strong branch S has the property that $C(\{s\}, S) > C(S, \{t\})$ as its total mass or sum of weights is positive. The maximum amount of flow possible through S is, thus, $C(S, \{t\}) = \min\{C(\{s\}, S), C(S, \{t\})$. The flow constructed in Theorem 4 is equal to $C(S, \{t\})$ (it saturates all sink-adjacent arcs in the strong branch) and, thus, is maximum.

Similarly, for a weak branch W, $C(\{s\}, W) \leqq C(W, \{t\})$, and the flow constructed is of value $C(\{s\}, W)$ and, thus, is maximum. ▪

Let a tree network be an s, t graph so that when s and t are removed the remaining arcs form a forest, or an (undirected) acyclic graph.

Corollary 3. *The maximum flow on a tree network in a closure graph can be found in linear time.*

Example: Consider a strong branch of mass 3 displayed in Figure 7. It includes nodes $\{a, b, c, d, e, f\}$ of weights $\{2, -3, -1, 2, 4, -1\}$, respectively. The sink adjacent arcs from nodes b, c, f are saturated and so are the downward arcs in the branch (a, c), (c, f). Flows are then assigned to the remaining arcs adjacent to a and then to those adjacent to b and to c. Finally, flows are assigned to the arcs adjacent to the leaves d and e (f has no unassigned arc adjacent to it). The flows on the arcs that are permitted to be unsaturated are displayed next to their mass capacity in Figure 7.

Remark. *It can be shown that the set of nodes reachable from the source in the residual graph with respect to the constructed feasible flow is precisely the set of strong nodes. Therefore, at optimality, this set forms the minimal source set of a minimum cut.*

5.1. The Monotonicity of the Constructed Feasible Flow

The changes in the feasible flow constructed mimic the changes in the total weight of the strong nodes, but in the opposite direction. More precisely, we establish that the strong mass can only go down through consecutive iterations, while the value of the corresponding feasible flow constructed goes up by the same increment:

Lemma 5. *Consider two normalized trees T_1 and T_2 with sets of strong nodes S_1 and S_2 and weak nodes W_1 and W_2, respectively. Then, the corresponding constructed flows f_1 and f_2 satisfy*

$$|f_1| - |f_2| = b(S_2) - b(S_1).$$

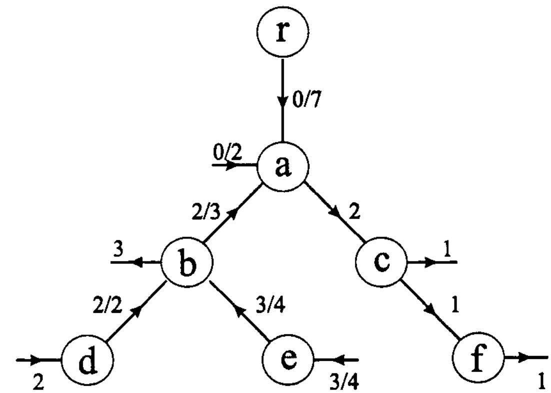

FIG. 7. The construction of feasible flow.

Proof. The value of the flow in T_1 is $C(S_1, \{t\}) + C(\{s\}, W_1)$. In the strong branches, this flow value is $C(S_1, \{t\}) = C(\{s\}, S_1) - b(S_1)$, since $b(S_1) = C(\{s\}, S_1) - C(S_1, \{t\})$. The total flow is thus $|f_1| = C(\{s\}, V) - b(S_1)$. Similarly, for the tree T_2, $|f_2| = C(\{s\}, V) - b(S_2)$. The statement of the lemma thus follows. ▪

5.2. Relationship of LG to Push-Relabel and Network Simplex

Since the time this paper was first written in 1996, we were able to extend the algorithm described here to general graphs. The algorithm that solves the maximum-flow problem on general graphs works, instead of with masses, with pseudoflows. A pseudoflow on a network satisfies capacity constraints, but may violate flow balance constraints by creating deficits and excesses at nodes. The relationship of the pseudoflow algorithm to push-relabel and network simplex is clearer than that of the LG algorithm. We use those insights to comment on the major differences between LG and other known maximum-flow algorithms.

A normalized tree is reminiscent of the basic arcs simplex tree and, as such, raises the possibility that the LG algorithm may be a simplex algorithm. Indeed, both algorithms work with a tree, and an iteration amounts to adding an arc to the tree and removing arcs from the created cycle. On that point it is clear that the LG algorithm cannot be a simplex algorithm: It adds a merger arc but may remove an arbitrary number of arcs (or edge splits). In [32], we constructed a simplex algorithm that works with a normalized tree, but the outcome of each iteration is different from that of LG. Moreover, we discovered in a recent empirical study (by Anderson and Hochbaum) that the performance of this simplex algorithm is inferior to that of the LG algorithm in terms of running time on several classes of graphs.

The push-relabel algorithm works with preflows—that is, flow values that satisfy capacity constraints but violate flow-balance constraints in creating excesses. Excess happens when inflow is greater than outflow. The push-relabel algorithm amounts to dispatching flow from

nodes with excess to those closer to the sink, as measured by distance labels. In contrast, the LG algorithm works with *sets* of nodes—the branches which can accumulate either excesses or deficits. This feature of dealing with sets of nodes, rather than with individual nodes, distinguishes LG from push-relabel. Here, again, the empirical study mentioned above shows that the pseudoflow algorithm, extending the LG algorithm, is substantially faster than is the push-relabel algorithm or Dinic's algorithm (implemented by Cherkassky and Goldberg [7]).

6. COMPLEXITY IMPROVEMENTS

6.1. *Reverse Graphs*

Since the maximum-closure problem is equivalent to the minimum-cut problem, it is possible to substitute finding a minimum cut in the related graph by finding a minimum cut in the reverse graph, where the roles of source and sink are reversed and all arcs are reversed.

Using reverse graphs is advantageous when $M^+ = \sum_{i \in V^+} b_i > \sum_{i \in V^-} |b_i| = M^-$. In mining applications, the sum of positive-weight nodes is typically larger than that of negative-weight nodes. This occurs since the volume of soil considered is presumed to be relatively rich in ore; else, that location would not have been considered in the first place. We found that reversing the graph is indeed of considerable benefit in speeding up performance [34].

The complexity of the LG algorithm is thus $O(mn \cdot \min\{M^+, M^-\})$.

6.2. *0-Mass Branches*

Inspecting closely the proof of the complexity bound of the LG algorithm, it is evident that the factor of $|V| = n$ in the number of iterations results from mergers with a total weight of strong nodes that remains unchanged. This happens in case (ii) of Theorem 4 where a downward arc that supported 0 mass prior to the merger supports the mass M_S afterward. This happens only in cases where a part of the weak branch contains a subtree of 0 total weight. We shall call such a subtree a 0-mass branch.

To ensure that the weight of strong nodes goes *strictly* down at each iteration, we refine the partition of the weak branches into negative mass branches and 0-mass branches. We thus partition the nodes in V for a given normalized tree $T = (V, A^T)$ into three types of sets: $V_S^{(T)}$, $V_W^{(T)}$, and $V_0^{(T)}$, which are strong; weak and in branches of negative mass; and weak and in branches of zero mass, respectively.

The algorithm is now modified so that a merger is either due to an arc in $(V_S^{(T)}, V_W^{(T)})$ or to a *path* that starts at a strong node $s \in V_S^{(T)}$ and ends at a weak node $w \in V_W^{(T)}$. Such a path is of the form

$$[s, v_1, \ldots, v_1', v_2, \ldots, v_2', \ldots, v_k, \ldots, v_k', w],$$

where $(s, v_1), (v_1', v_2), \ldots, (v_i', v_{i+1}), \ldots, (v_k', w)$ are all directed arcs and $[v_i, \ldots, v_i']$ is an undirected path contained entirely in some 0-mass branch $T_i^{(0)}$, for $i = 1, \ldots, k$. We call such a path a *path via 0-mass*.

In the statement of **improved LG**, we only detail the adjusted merger process. The mass updates and renormalization are modified in an obvious way.

Procedure improved LG ($G = (V, A)$, $b_j\ \forall j \in V$, T)
$V_S^{(T)} = \{v \in V | v \text{ is strong in } T\}$.
While there is a path via 0-mass
$[s, v_1, \ldots, v_1', v_2, \ldots, v_2', \ldots, v_k, \ldots, v_k', w]$
from $s \in V_S^{(T)}$ to $w \in V_W^{(T)}$ **do**
For r_v the root of the branch containing v:
{*Merger*} $T \leftarrow T \setminus \{(r, r_s), (r, r_{v_1}), \ldots, (r, r_{v_k})\}$
$\cup \{(s, v_1), (v_1', v_2), \ldots, (v_k', w)\}$.
{*Mass updates and renormalization*}
end
Output "$V_S^{(T)}$ is a maximum closed set."

Remark. *To find multiple optima, we check in the* **while** *loop also for arcs from* $s \in V_S^{(T)} \cup V_0^{(T)}$ *to* $w \in V_W^{(T)}$ *to ensure that the 0-mass branches are also closed at termination. Then, the strong nodes along with any subset of the 0-mass branches constitute an alternative optimal solution. In fact,* $V_S^{(T)} \cup V_0^{(T)}$ *is a* `maximal` *maximum-closure set.*

Complexity: The number of iterations in **improved LG** is $O(M^+)$. At each iteration, the work to find a path and to remove strong and 0-mass arcs is $O(m)$. Hence, the complexity of the improved algorithm is $O(mM^+)$.

With both adjustments, for reverse graphs and 0-mass mergers, the complexity of the algorithm is $O(m \min\{M^+, M^-\})$. This complexity, while improved, is still pseudopolynomial.

6.2.1. *Complexity Tightness*

To see that the complexity expression is attainable, we use the 9-node example in Figure 5. In this example, the number of iterations is precisely $M^+ = \sum_{i \in V^+} b_i$. By generating a family of graphs in which this 9-node graph repeats arbitrarily many times, we conclude that the complexity of **improved LG** is met, namely, that the mass of strong nodes decreases by one unit at each iteration.

For a sharper complexity estimate, we replace the value M^+ by the gap between the initial mass of strong nodes and their mass at termination. In the example of Figure 5, the mass of the optimal closed set is 0, so the gap is equal to $M^+ = 20$. The optimal solution in this graph is the empty set or any closed set of total value

0. Figure 5 illustrates iterations 0–4 and 17–20 where at each iteration the mass of strong nodes decreases by 1.

6.3. A Polynomial Scaling Algorithm

To improve the complexity of the algorithm further, we speed up the closing of the "gap" between the initial guessed value M^+ and the optimal value of the maximum-closure. One standard method of achieving such an outcome is by using a scaling technique which dates back to the fundamental work of Edmonds and Karp [14]:

Procedure scaled LG ($G = (V, A)$**,** $b_j \ \forall j \in V$**)**
Let $p = \lceil \log_2(\max_j\{b_j\}) \rceil - 1$. For all $j \in V$, $\bar{b}_j \leftarrow \lfloor \frac{b_j}{2^p} \rfloor$.
Let $T = (V \cup \{r\}, A(T))$ for $A(T) = \{(r, j) | j \in V\}$.
For all $j \in V$, $M_{(r,j)} = \bar{b}_j$.
$V_S^{(T)} = \{v \in V | v \text{ is strong in } T\}$.
Until $p = 0$, **do**
  Set $\bar{b}_j \leftarrow \lfloor \frac{b_j}{2^p} \rfloor$, for all $j \in V$.
  Update masses of nodes and arcs in T.
  Call **Renormalize tree** (T, $\bar{b}_j \ \forall j \in V$).
  Call **improved LG**($G = (V, A), \bar{b}_j \ \forall j \in V, T$).
  Return optimal normalized tree T.
  $p \leftarrow p - 1$.
end
Output "$V_S^{(T)}$ is a maximum closed set."

Complexity: The value of the maximum closure at each iteration multiplied by 2^p can only be a lower bound on the value of the maximum closure with the weights b_j. After the rescaling, the weights of all nodes increase by at most 1 additional unit each. The gap between the weight of the strong nodes and the lower bound is then at most n.

The work per call to **improved LG** is thus $O(mn)$. Let $||b|| = \min\{\max_{j \in V^+} b_j, \max_{j \in V^-} |b_j|\}$. We use the reverse graph if the second term is the minimum. With this feature, the complexity of scaled LG is $O(mn \log_2 ||b||)$. This complexity is polynomial but not strongly polynomial.

6.4. A Strongly Polynomial Variant: The Lowest Label Rule

Procedure LG selects at each iteration any merger arc (s, w) from a strong to a weak node (`select` step). We call such an arc an *active* arc. This section shows that restricting the choice of the active arc leads to a strongly polynomial algorithm. The choice of the arc depends on the labeling of the nodes.

The lowest label rule was motivated by the distance labels introduced by Goldberg [23]. Similar rules were used by Goldfarb and Hao [25, 26].

The labeling scheme is described recursively. Initially, all nodes are assigned the label 1, $\ell_v = 1$, $\forall v \in V$. At each iteration, an arc (s, w) is selected so that w is a lowest label weak node among all possible active arcs in (S, W).

Upon a merger, using the arc (s, w), the label of the strong node s becomes the label of w plus 1 and all nodes of the strong branch with labels smaller than that of s are updated to be equal to the label of s. Formally, we replace the statement "`select` $(s, w) \in (V_S^{(T)}, V \setminus V_S^{(T)})$" by

" `select` $(s, w) \in (V_S^{(T)}, V \setminus V_S^{(T)})$ so that ℓ_w is minimum; $\ell_s \leftarrow \ell_w + 1$
$\forall v \in S$, $\ell_v \leftarrow \max\{\ell_v, \ell_s\}$."

Let $level(v)$ be the distance of node v, in terms of the number of arcs in the path, to the root of the tree r. The labels satisfy the following invariant properties throughout the execution of the algorithm:
Invariant 1: For every arc $[u, v]$ so that $(u, v) \in A^T$ and every arc $(u, v) \in A$, $\ell_u \leqq \ell_v + 1$. (This means that $|\ell_u - \ell_v| \leqq 1$ for $(u, v) \in A^T$.)
Invariant 2: [Monotonicity]. On any path in a branch from the root down, labels of nodes are nondecreasing.
Invariant 3: For weak nodes, $\ell_v \leqq level(v)$.
Invariant 4: Labels of nodes are nondecreasing over the execution of the algorithm.

Let ℓ_u, ℓ_v be the labels prior to the relabeling at iteration $k + 1$ and ℓ'_u, ℓ'_v be the labels after the relabeling is complete at iteration $k + 1$.
Proof of Invariant 1. Assume by induction that the invariant holds through iteration k, and prove that it holds through iteration $k + 1$ as well. Obviously, the invariant is satisfied at the outset, when all labels are equal to 1. Consider an arc (u, v) after the relabeling at iteration $k+1$ and let arc (s, w) be the merger arc in this iteration. We sort out the different cases according to the status of the nodes at the beginning of iteration $k + 1$.
u strong, v weak: At iteration $k + 1$, only node u could have changed labels as weak nodes are not relabeled. Since $\ell_v \geq \ell_w$ and w is a lowest label weak merger node,

$$\ell'_u = \max\{\ell_u, \ell_w + 1\} \leqq \ell_v + 1 = \ell'_v + 1.$$

u strong, v strong: Suppose that the label of u has increased, then $\ell'_u = \ell_w + 1$. If the label of v has not likewise increased, then $\ell_v \geq \ell_w + 1$ and $\ell'_v \geq \ell'_u$ and the inequality is satisfied. Otherwise, the two nodes have the same label, $\ell'_v = \ell'_u$.

u weak, v strong: Only the label of v could have been updated and then only upward.
u weak, v weak: Weak nodes do not get relabeled, so the inequality that was satisfied at iteration k is still satisfied at the end of iteration $k + 1$.

Note that the only arc that is out-of-tree at iteration k and in-tree at iteration $k + 1$ is the arc (s, w). Because of the update of the label of s, $|\ell'_s - \ell'_w| \leqq 1$. ∎

Proof of Invariant 2. Assume by induction that monotonicity is satisfied through iteration k. The operations that might affect monotonicity at iteration $k+1$ are relabeling and splitting of branches. As a result of relabeling, the nodes on the section of the path $[r_s, \ldots, s]$ are all labeled with the label ℓ_w+1, since, previously, all the labels of these nodes were $\leqq \ell_s$ by the inductive assumption of monotonicity. After the merger, the roles of parents and children are reversed along the path, but since they all have the same label, the monotonicity still holds. It also holds for all subtrees that are suspended from the merger path nodes since parent/child relationships are not modified for the subtrees and, thus, $\ell_u \leqq \ell_v$ implies that $\ell'_u \leqq \ell'_v$. ∎

Proof of Invariant 3. The roots of weak branches are always labeled 1 since no weak branch is created throughout the algorithm. From the monotonicity and Invariant 1, all weak branches satisfy that the labels of nodes are less than or equal to their level. ∎

Proof of Invariant 4. From Invariant 1, $\ell_s \leqq \ell_w + 1$, and, thus, the label of s can only increase. The method of labeling all other nodes implies that their label can only increase. ∎

Lemma 6. *Between two consecutive mergers using merger arc (s, w), the labels of s and w must increase by at least 1 each.*

Proof. Upon the first merger using (s, w), let the initial label of w be $\ell_w = L$. After the merger's relabeling, $\ell_s \geq L + 1$. Before (s, w) can serve again as a merger arc, both nodes must become strong and subsequently have the arc between them renormalized so that w is weak and s remains strong. This can happen if either

- w is above s in a strong branch and the merger node is s or a descendant of s. Because of the monotonicity of labels, after such a merger, the label of w must satisfy $\ell_w \geq L + 1$. Or,
- (w, s) serves as a merger arc. But, then, w is relabeled to be at least $\ell_s + 1$ and $\ell_w \geq \ell_s + 1 \geq L + 2$.

In either case, w is relabeled so that $\ell_w \geq L+1$. Upon repeated use of (s, w) as merger arc, $\ell_s \geq \ell_w+1 \geq L+2$. Thus, the labels of s and w have increased by at least 1 each between the consecutive mergers. ∎

Corollary 4. *The number of iterations in the "lowest label" variant is at most $O(mn)$.*

Proof. Invariant 3 implies that the weak labels are bounded by n (or, rather, by the number of weak nodes). But according to the labeling method (the label of a weak node $+1$), no strong node can ever be labeled more than n. By Invariant 4, labels are nondecreasing. Consequently, the same arc can be used in mergers at most n times. ∎

Arguments identical to those used in Lemma 6 demonstrate that between two consecutive edge renormalizations of an arc the labels of each of its endpoints must increase by at least 1. Thus, the number of edge "slices" or renormalization operations is at most n times per arc. A different argument leading to the same conclusion begins with the observation that the number of strong branches can never exceed n. At each iteration, either the number of strong branches decreases by 1 (when all the nodes of the merged strong branch become weak) or it increases by a nonnegative increment. Since we can "accumulate" at most an mn deficit in the number of strong branches, the total increase throughout the algorithm in the number of strong branches is no more than $n + mn$. We conclude

Corollary 5. *The number of edge normalizations in the "lowest label" variant is at most $O(mn)$.*

With $O(m)$ work per iteration, the total complexity is $O(m^2n)$. We present a more careful analysis demonstrating $O(mn \log n)$ complexity.

Theorem 3. *The complexity of the lowest label algorithm is $O(mn \log n)$.*

Proof. As before, an arc (u, v) is *active* if u is strong and v is weak. Each iteration requires one to
1. Identify an active arc of smallest label ℓ_v;
2. Merge the strong and weak branches and update the masses of arcs along the merger paths;
3. Renormalize, by identifying recursively a positive downward arc on the merger path;
4. Remove the tree that is supported by that arc as a new strong branch.

We take advantage of the invariant properties of the labels to identify easily an active arc of lowest label. To find an active arc of weak label ℓ, there must be a strong node of label $\ell - 1$, ℓ or $\ell + 1$. Because of the monotonicity, it suffices to search strong branches with roots that have label no larger than $\ell + 1$.

We now search for an active arc or lowest label ℓ by scanning a strong branch in a depth first search manner for arcs adjacent to strong nodes of labels no more than $\ell + 1$.

For each strong node, either a merger arc is found or its entire adjacency list has been scanned and the node will no longer be visited until its label is incremented. Since there are at most n labels, the total scanning per

label value for all nodes has complexity of $O(m)$ for a total of $O(mn)$ throughout the algorithm.

For the update of the masses and the maintenance of the merged and inverted trees, we rely on Sleator and Tarjan's dynamic trees data structure [59]. In this data structure, the operations of merger, rerooting of trees, finding the minimum or maximum weight arc on the path from a node to the root of the tree, and adding a constant to the weight of arcs along such paths all have amortized complexity of $O(\log n)$ per operation. Here, we first perform the mass update and later the renormalization—that is, finding first positive mass arc on the merger path. Once this is found, the quantity added to the remaining arcs along the path is adjusted by a constant and the process calls again for finding the first nonnegative mass arc on the merger path. We now show that Sleator and Tarjan's procedure of finding maximum-weight arc can be modified to find a first nonnegative mass arc.

The concept of dynamic trees is to extend operations that are efficiently implemented on paths to be implemented on trees with the same complexity. For the dynamic trees data structure, a path of length k is represented as a binary tree of depth $O(\log k)$ whose nodes in symmetric order are the vertices of the path. Let the leftmost node be the root of the (weak) branch and the rightmost node be the endpoint of the path. In this tree, each node v is labeled by the maximum value of a certain quantity associated with the section of the path from that node to the endpoint, $\max(v)$. In this context, the quantity is the maximum value of downward arc masses from the endpoint of the path to the node v. Or, in the binary tree, it is the maximum value of the nodes that are descendants of the right child of v. The procedure FIND-FIRST identifies the first node that is adjacent to a downward arc with mass ≥ 0 on the path from v to the root r_w:

procedure FIND-FIRST(v)
Until $v = r_w$, **do**:
If $\max(v) < 0$, then $v \leftarrow parent(v)$
Else $u = left - child(v)$.
    While $\max(u) < 0$, $u = left - child(u)$.
    Otherwise, ($\max(u) \geq 0$), if u is a leaf
    of the binary tree then output u, and stop,
    else $u = right - child(u)$.
    end
end
Output "no strong edge on path."

If the procedure outputs the value v, then $(ch(v), v)$ is the first downward arc on the merger path with mass ≥ 0.

The complexity of this procedure for a tree of height h is at most $3h$, since it involves going up the tree at most once per node along a path to the root, going down to the right at most h nodes, and going down to the left at most h nodes. A path is of length n at most; thus, h is at most $O(\log n)$. Thus, the overall complexity is $O(mn \log n)$. ▪

7. PARAMETRIC AND SENSITIVITY ANALYSIS

Currently[2], sensitivity analysis in the mining industry is performed by *repeated* applications of an algorithm that computes a single optimal pit for a given set of parameter values. For example, to see the effect of commodity price on the optimal pit, a series of runs is performed where the commodity price is varied.

We call the analysis for a given set of values, *sensitivity analysis for given parameter values*. The parametric algorithms that we propose are more efficient than are the (repeated application) methods currently in use, which resolve the problem for each parameter value.

An alternative to sensitivity analysis for given parameter values is a parametric analysis generating all possible values—breakpoints—of a relevant parameter where the contour of the pit (or the cut) is affected. This analysis provides more information than does the sensitivity analysis, since for each given parameter value, it is only necessary to find the interval of the breakpoints where the parameter value falls. This type of analysis is referred to as *complete parametric analysis*. This type of analysis has never been performed in the industry as it is viewed as computationally "intractable."[3]

When performing sensitivity analysis, one of the elements in the block value formula is identified as the parameter of interest, and its effect on the optimal pit is analyzed. This parameter could be the ore commodity value or the processing cost per block. Let λ denote this parameter. The block weight 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 dependent on λ. The open-pit graph $G = (V, A)$, with the weights $b_i(\lambda)$ associated with the nodes, is denoted by G_λ.

It will be assumed, without loss of generality, in the discussion hereafter that the values of d_i are nonnegative.

7.1. Sensitivity Analysis for Given Parameter Values

Here, we are given a set of parameter values, arranged in a monotone increasing order, $\lambda_1 < \lambda_2 < \cdots < \lambda_q$.

Let S_λ be a minimal maximum closure in the graph G_λ. It is well known that as λ increases so does the set S_λ. Gallo et al. [20] proved this fact using the properties of the push-relabel algorithm. An alternative proof uses the properties of the LG algorithm.

Lemma 7. *For a sequence of parameter values, $\lambda_1 < \lambda_2 < \cdots < \lambda_q$, the corresponding minimal optimal closures satisfy*

$$S_{\lambda_1} \subseteq S_{\lambda_2} \subseteq \cdots \subseteq S_{\lambda_q}.$$

Proof. As λ increases from λ_i to λ_{i+1}, the masses of all branches increase. As a result, the mass of the closed set of strong branches S_{λ_i} in the normalized tree is going up, so the set remains strong. Some weak branches may become strong and require extra processing, by merging with weak branches, to guarantee closure. Yet, none of the strong nodes in S_{λ_i} change their status from strong to weak in the process. Thus, the closed set corresponding to the new value of the parameter, $S_{\lambda_{i+1}}$, must contain S_{λ_i}. ∎

In the interest of brevity, we defer the detailed implementation of the LG algorithm and its polynomial scaling variant to the Appendix. We only note that at the start of the computation for a new parameter value the optimal tree from the previous parameter value is maintained without retracting any prior merger. In particular, for the strongly polynomial variant of sensitivity LG, the invariant properties of the labels are preserved. Some weak branches may become strong and may require renormalization, but the labels may be preserved without change between consecutive calls and no merger needs to be retracted. The running time is therefore $O(mn \log n)$.

7.2. Complete Parametric Analysis

Gallo et al. [20] devised a complete parametric analysis algorithm based on the push-relabel algorithm. Their algorithm is described for arc capacities of source and sink adjacent arcs that are linear in the parameter λ. For our graphs, the capacities of the arcs are monotone piecewise linear:

$$c(s, v) = \max\{0, c_v + \lambda d_v\} \quad \text{and} \quad c(v, t) = -\min\{0, c_v + \lambda d_v\}.$$

For these particular parametric functions, a large enough constant is added to all capacities $C(\{s\}, V)$ and $C(V, \{t\})$ so that no capacity function crosses zero for the range of interest. It is noted in [20] that the minimum cuts are preserved when adding a constant to the source and sink adjacent capacities, and, therefore, their algorithm for linear capacities applies.

Note that the algorithm for complete parametric analysis can be applied for arbitrary monotone nondecreasing source adjacent arcs' capacities and monotone nonincreasing sink adjacent arcs's capacities. A detailed description is provided in [32].

The parametric algorithm in an interval where all capacities are linear works by comparing the total benefit (or mass) of a minimal maximum closed set at the left endpoint of the interval to that of a maximal maximum closed set at the right endpoint of the interval. If these are two identical functions of λ, then there is no breakpoint in between. Otherwise, there is at least one breakpoint, and we bisect the interval at the intersection point of the two lines, searching further for such a breakpoint.

7.2.1. A Complete Parametric Analysis Using Push-Relabel We sketch here briefly how the algorithm of Gallo et al. [20] can be adapted to the LG algorithm. For a given interval where we search for breakpoints, we run the algorithm twice: from the lower endpoint of the interval where the maximal source set of the cut obtained at that value shrunk into the source and from the highest endpoint of the interval where the maximal sink set of the cut is shrunk into the sink. The runs proceed for the graph and reverse graph until the first one is done. The newly found cut subdivides the graph into a source set and a sink set, one of which is smaller in terms of the number of nodes $n_1 \leqq \frac{1}{2}n$. In that smaller interval, two new runs are initiated from both endpoints. In the larger interval, however, we *continue* the previous runs using two properties:

- *Reflectivity*: The complexity of the algorithm remains the same whether running it on the graph or reverse graph.
- *Monotonicity*: Running the algorithm on a monotone sequence of parameter values has the same complexity as that of a single run.

Under these assumptions, one run is "reflected" to the opposite endpoint and thus viewed as monotone continuation and the other continues as a monotone continuation.

An essential ingredient in the algorithm is the availability of maximal and minimal maximum closed sets. The LG algorithm provides a minimal maximum closed set, and in Remark 6.2, we discuss how to derive a maximal one as well.

We denote a maximal maximum closed set by $S^{\max}$ and a minimal maximum closed set by $S^{\min}$. Let $b_\lambda(D) = \sum_{j \in D}[c_j + \lambda d_j]$ for $D \subseteq V$. We assume, henceforth, that the LG algorithm and its variants deliver as output both $S^{\min}$ and $S^{\max}$. As a corollary of Lemma 7, we get the following result that is used to contract nodes in the graph and reduce its size: A *contraction* of a set of nodes means replacing the set by a single node that has as incoming and outgoing arcs, all the incoming and outgoing arcs incident with the set.

Lemma 8 Contraction Lemma. *For $\lambda \in (\lambda_1, \lambda_2)$, a maximum closed set S_λ in the graph G_λ in which the set S_{λ_1} is contracted with the source and $\bar{S}_{\lambda_2} - S_{\lambda_1}$ is contracted with the sink, is also a maximum closed set in G_λ.*

A contraction procedure was also used in [20], but with a minor error. There, the set contracted with the sink is $\bar{S}_{\lambda_2}$, which renders the contraction invalid as explained in [35], Remark 2.

7.2.2. A Complete Parametric Analysis for LG For a given interval (λ_1, λ_2) where arc capacities are linear, we can find all breakpoints by using the procedure **parametric linear-LG**. The procedure is initialized with values of λ_1, λ_2 as in [20]. Here, "variant LG" stands for "improved LG," "scaled LG," or "lowest label LG."

Procedure parametric linear-LG $(\lambda_1, \lambda_2, G = (V, A),$
$c_j, d_j \ \forall j \in V, S^{\max}_{\lambda_1}, S^{\min}_{\lambda_2})$
$S_1 = S^{\max}_{\lambda_1}, S_2 = S^{\min}_{\lambda_2}$.
Contract: $s \leftarrow s \cup S_1$, $t \leftarrow t \cup \bar{S}_2 - S_1$. If $V = \{s, t\}$, halt and output "no breakpoints."
If the two functions $b_\lambda(S_1)$ and $b_\lambda(S_2)$ are not identical, find λ^* such that $b_{\lambda^*}(S_1) = b_{\lambda^*}(S_2)$. Else, halt and output "no breakpoints."
Call **variant LG**$(G_{\lambda^*}, b_j(\lambda^*) = c_j + \lambda^* d_j \ \forall j \in V,$
$T_0(G_{\lambda^*}, b(\lambda^*)))$ for the output $S^{\min}_{\lambda^*}, S^{\max}_{\lambda^*}$
If λ^* is a breakpoint, output λ^*. Else continue,
    Call **parametric linear-LG**
$(\lambda_1, \lambda^*, G = (V, A), c_j, d_j \ \forall j \in V, S^{\max}_{\lambda_1}, S^{\min}_{\lambda^*})$
    Call **parametric linear-LG**
$(\lambda^*, \lambda_2, G = (V, A), c_j, d_j \ \forall j \in V, S^{\max}_{\lambda^*}, S^{\min}_{\lambda_2})$
end

It remains to describe how to verify that λ^* is a breakpoint. Observe that for a breakpoint the right derivative of the function b_λ at λ^* is strictly smaller than the left derivative of b_λ at λ^*. To verify that, pick a small $\epsilon > 0$. λ^* is a breakpoint if

$$b_{\lambda^*+\epsilon}(S^{\max}_{\lambda^*}) - b_{\lambda^*}(S^{\max}_{\lambda^*}) > b_{\lambda^*}(S^{\min}_{\lambda^*}) - b_{\lambda^*-\epsilon}(S^{\min}_{\lambda^*}).$$

Alternatively, compare the two functions of λ: $b_\lambda(S^{\max}_{\lambda^*})$ and $b_\lambda(S^{\min}_{\lambda^*})$.

As for the complexity of the procedure for LG with improved LG implementation or the lowest label, we use arguments similar to those used in [20]: Let $m_1 + m_2 \leqq m$, $n_1 + n_2 \leqq n$ and $n_1 \leqq \frac{1}{2}n$. The running time $T(m, n, M)$ for a graph on m arcs, n nodes, and total (positive) weight M satisfies for a constant Q the recursion

$$T(m, n, M) = T(m_1, n_1, M_1) + T(m_2, n_2, M_2) + 2Qm_1(M_1 + M_2)$$

Hence, $T(m, n, M) = QmM = O(mM)$.

For the lowest label implementation, $T(m, n)$ is the running time on a graph with m arcs and n nodes:

$$T(m, n) = T(m_1, n_1) + T(m_2, n_2) + 2Qm_1 n_1 \log n.$$

The solution is $T(m, n) = O(mn \log n)$.

7.2.3. Use of Complete Parametric Analysis in Mine Scheduling As discussed in the literature review, the notion of parameterization with a series of nested pits that have increasing benefit (or metal content) per block is used in mine scheduling. We show here that such series can be generated using complete parametric analysis, when the term that is independent of the parameter is constant for all blocks. That is, $b_j = c + \lambda d_j$. This structure occurs when processing costs, c, are identical for all blocks.

This information enables one to schedule the sequence of pits to be mined so as to maximize NPV (net present value).

Defining $S_0 = \emptyset$, the parametric analysis provides a series of q nested pits,

$$S_1 \subset S_2 \subset \cdots \subset S_q.$$

Recall the notation $b(S) = \sum_{j \in S} b_j$.

Lemma 9. *The nested pits generated by complete parametric analysis satisfy*

$$\frac{b(S \setminus S_{i-1})}{|S_i \setminus S_{i-1}|} \geq \frac{b(S_{i+1} \setminus S_i)}{|S_{i+1} \setminus S_i|}$$

for $i = 1, \ldots, q - 1$.

Proof. Since S_i is an optimum closed set for λ_i, the benefit of the set $S_{i+1} \setminus S_i$ is nonpositive, $\sum_{j \in S_{i+1} \setminus S_i} [c + \lambda_i d_j] \leqq 0$. Yet, S_i is a minimal maximum closed set and it is nonempty, so $\sum_{j \in S_i \setminus S_{i-1}} [c + \lambda_i d_j] > 0$. Hence,

$$c + \lambda_i \frac{\sum_{j \in S_{i+1} \setminus S_i} d_j}{|S_{i+1} \setminus S_i|} \leqq 0 < c + \lambda_i \frac{\sum_{j \in S_i \setminus S_{i-1}} d_j}{|S_i \setminus S_{i-1}|}.$$

Thus,

$$\frac{b(S_i \setminus S_{i-1})}{|S_i \setminus S_{i-1}|} = \frac{c|S_i \setminus S_{i-1}| + \lambda \sum_{j \in S_i \setminus S_{i-1}} d_j}{|S_i \setminus S_{i-1}|} > \frac{c|S_{i+1} \setminus S_i| + \lambda \sum_{j \in S_{i+1} \setminus S_i} d_j}{|S_{i+1} \setminus S_i|} = \frac{b(S_{i+1} \setminus S_i)}{|S_{i+1} \setminus S_i|}.$$

■

8. CONCLUSIONS

The LG algorithm investigated in this paper is of a great deal of interest for its different perspective on the minimum-cut problem in closure graphs. We investigate here the LG algorithm and provide insights into the maximum-flow problem from the perspective of this new algorithm. We study the complexity of the algorithm and proposed variants of the algorithm that have polynomial and strongly polynomial time complexities. By-products of this complexity study include useful guidelines for efficient implementations.

Our study fills in the missing link between the solution delivered by the LG algorithm and the maximum-flow problem. The work here may thus be viewed as a

new algorithm for the maximum-flow problem on closure graphs. That new algorithm is shown to have attractive features that permit efficient parametric analysis implementation.

Since the approach of the algorithm is dramatically different from those of other algorithms for maximum flow, we view the introduction of the LG algorithm as an important contribution to the set of tools available for the maximum-flow problem. It is noted that in follow-up work we were able to devise a new algorithm for maximum flow on *general* graphs that appears to perform more efficiently in practice than other known algorithms. Moreover, that algorithm has parametric analysis and flexibility features that are valuable in the context of sensitivity analysis. These permit us to solve the problem with changed capacities, even on arcs that are not adjacent to source and sink, using a so-called "warm-start," which effectively makes use of the optimal solution to the previous instance solved. This capability is not shared by any other maximum-flow algorithm and is thus of substantial practical use.

Acknowledgments
The author thanks Anna Chen for her contribution to the earlier phases of the research reported here and for her permission to include these results. The author is indebted to the referees for detailed and thoughtful comments, pointing out additional references and making numerous corrections and simplifying suggestions.

APPENDIX

Adapting LG to Sensitivity Analysis

The following procedure generates a series of optimal pits or closures corresponding to the values of the given parameter.

Procedure sensitivity LG ($\lambda_1 < \lambda_2 < \cdots < \lambda_q$,
$G = (V, A)$, $c_j, d_j\ \forall j \in V$)
$T = (V \cup \{r\}, A')$ for $A' = \{(r, j) | j \in V\}$.
$i = q$.
Until $i = 0$ **do**
  $\lambda = \lambda_i$.
  Call **improved LG** (G_λ, $c_j + \lambda d_j\ \forall j \in V$, T).
  Output the closed set S_{λ_i}.
  $i \leftarrow i - 1$.
end

The complexity of sensitivity LG is easy to establish. When we start, the total weight of the positive weight nodes is as large as possible, $M^+(\lambda_q) = \sum_{j \in V^+(\lambda_q)} [c_j + \lambda_q d_j]$. This mass is going down with the first call to **improved LG**. When the tree T is returned, the total mass of strong nodes is only smaller. Substituting for these nodes, the smaller weights for the smaller value of λ can only reduce further the strong nodes' mass (or total weight). Now, some of the strong branches become weak and the new set of strong branches is no longer closed and LG applies several merger steps. Hence, throughout, the mass of strong nodes is monotonically decreasing. The complexity is thus $O(mM^+(\lambda_q) + qn)$, where the qn factor is required for the renormalization at each call. This is again possible to improve by using the reverse graph and the ascending ordering of the parameter, for an improved running time of $O(m \min\{M^+(\lambda_q), M^-(\lambda_1)\} + qn)$.

A Polynomial Variant of Sensitivity LG

Repeated calls to **scaled LG** may replace the calls to **improved LG**. In that case, the complexity is $O(mnq \log ||b||)$, where $||b|| = \min\{\max_{j \in V^+} c_j + \lambda_k d_j, \max_{j \in V^-} |c_j + \lambda_1 d_j|\}$.

It is possible, however, to further improve this running time and replace q by $\log q$ by using a form of bisection, along with contraction. Prior to calling this procedure, we need to find the maximum closed set for the lowest and highest values of λ in the set of parameters.

Procedure sensitivity scaled-LG ($\lambda_1 < \cdots < \lambda_q$,
$G = (V, A)$, $c_j, d_j\ \forall j \in V$, S_{λ_1}, S_{λ_q})
low = 1, high = q.
med = $\lceil \frac{\text{low}+\text{high}}{2} \rceil$.
$\lambda = \lambda_{\text{med}}$.
Contract: $s \leftarrow s \cup S_{\lambda_{\text{low}}}$, $t \leftarrow t \cup \bar{S}_{\lambda_{\text{high}}} - S_{\lambda_{\text{low}}}$.
While high $-$ low ≥ 3 **do**
  Call **scaled LG**
  (G_λ, $c_j + \lambda d_j\ \forall j \in V$).
  Output the closed set S_λ.
  Call **sensitivity scaled-LG**
  ($\lambda_{\text{low}} < \cdots < \lambda$, $G = (V, A)$, c_j, d_j
  $\forall j \in V$, $S_{\lambda_{\text{low}}}$, S_λ) .
  Call **sensitivity scaled-LG**
  ($\lambda < \cdots < \lambda_{\text{high}}$, $G = (V, A)$, c_j, d_j
  $\forall j \in V$, S_λ, $S_{\lambda_{\text{high}}}$) .
end

With each bisection of the interval $(\lambda_{\text{low}}, \lambda_{\text{high}})$ that has in the graph m arcs and n nodes, the contracted graphs corresponding to the subintervals $(\lambda_{\text{med}}, \lambda_{\text{high}})$ and $(\lambda_{\text{low}}, \lambda_{\text{med}})$ have m_1, n_1 and m_2, n_2 arcs and nodes, respectively, where $m_1 + m_2 \leqq m$ and $n_1 + n_2 \leqq n$. The complexity of **sensitivity scaled-LG** for k parameter values, $T_k(m, n, ||b||)$, thus satisfies

$$T_q(m, n, ||b||) = T_{\frac{q}{2}}(m_1, n_1, ||b||) + T_{\frac{q}{2}}(m_2, n_2, ||b||) + O(mn \log ||b||).$$

A solution to this recursive equation is $T_q(m, n, ||b||) = O(mn \log q \log ||b|| + nq)$.

Notice that when q exceeds n it is always more efficient to run the complete parametric analysis of Section 7.2, from which one can deduce any sensitivity analysis information for any prescribed set of parameter values.

Notes

1. The minimum s, t-cut problem is distinguished from the minimum 2-cut problem in that it seeks a partition of the nodes of the graph into two nonempty subsets that must contain the specified nodes s in one and t in the other. The distinction between these two problems is further discussed in Section 2.3, Remark 2.3.
2. This statement is based on the open literature. Proprietary software may possibly use more advanced concepts.
3. This assessment was offered by industry practitioners in private communication with the author.

REFERENCES

[1] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin, Network flows: Theory, algorithms, and applications, Prentice Hall, Englewood Cliffs, NJ, 1993.

[2] C. G. Alford and J. Whittle, 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. 1986, pp. 201–207.

[3] M.L. Balinski, On a selection problem, Mgmt Sci 17 (1970), 230–231.

[4] L. Caccetta and L.M. Giannini, On bounding techniques for the optimum pit limit problem, Proc AusIMM 290(4) (1985), 87–89.

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

[6] L. Caccetta, L.M. Giannini, and P. Kelsey, Optimum open pit design: A case study, Asia-Pacific J Oper Res 8 (1991), 166–178.

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

[8] T. Coleou, Technical parameterization of reserves for open pit design and mine planning, Proc 21st Int APCOM Symp, 1989, pp. 485–494.

[9] W.H. Cunningham, A network simplex method, Math Prog 1 (1976), 105–116.

[10] W. Cunningham, Optimal attack and reinforcement in a network, J ACM 32 (1985), 549–561.

[11] E. Dalhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis, The complexity of multiterminal cuts, SIAM J Comput 23 (1994), 864–894.

[12] B. Denby and D. Schofield, Genetic algorithms: A new approach to pit optimisation, Proc 24rd Int APCOM Symp, 1993, pp. 126–133.

[13] P.A. Dowd and A.H. Onur, Optimising open pit design and sequencing, Proc 23rd Int APCOM Symp, 1992, pp. 411–422.

[14] J. Edmonds and R. Karp, Theoretical improvements in algorithmic efficiency for network flow problems, J ACM 19 (1972), 248–264.

[15] M. J. Eisner and D. G. Severance, Mathematical techniques for efficient record segmentation in large shared databases, J ACM 23 (1976), 619–635.

[16] B. Faaland and T. Schmitt, Scheduling tasks with due dates in fabrication/assembly process, Oper Res 35 (1987), 378–388.

[17] L. R. Ford, Jr. and D. R. Fulkerson, A simple algorithm for finding maximal network flows and an application to the Hitchcock problem, Can J Math 9 (1957), 210–218.

[18] D. Francois-Bongarcon and D. Guibal, Algorithms for parameterizing reserves under different geometrical constraints, Proc 17th Int APCOM Symp, 1982, pp. 297–309.

[19] D. Gale, A theorem of flows in networks, Pac J Math 7 (1957), 1073–1082.

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

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

[22] A. V. Goldberg, Finding a maximum density subgraph, UC Berkeley report UCB/CSD/84/171, 1984.

[23] A. V. Goldberg, A new max-flow algorithm, Tech report MIT/LCS/TM-291, MIT, Cambridge MA, 1985.

[24] A.V. Goldberg and R.E. Tarjan, A new approach to the maximum flow problem, J Assoc Comput Mach 35 (1988), 921–940.

[25] D. Goldfarb and J. Hao, A primal simplex method that solves the maximum-flow problem in at most nm pivots and $O(n^2m)$ time, Math Prog 47 (1990), 353–365.

[26] D. Goldfarb and J. Hao, On strongly polynomial variants of the network simplex algorithm for the maximum-flow problem, OR Lett 10 (1991), 383–367.

[27] O. Goldschmidt and D.S. Hochbaum, A polynomial algorithm for the K-cut problem, Math Oper Res 19 (1994), 24–37.

[28] D. Gusfield, C. Martel, and D. Fernandez-Baca, Fast algorithms for bipartite network flow, SIAM J Comput 16 (1987), 237–251.

[29] P. Hansen and B. Simeone, Unimodular functions, Discr Appl Math 14 (1986), 269–281.

[30] R. Hassin, Maximum-flow in (s,t) planar networks, Inf Process Lett 13 (1981), 107.

[31] D.S. Hochbaum, On a polynomial class of nonlinear optimization problems, UC Berkeley manuscript, March 1990.

[32] D.S. Hochbaum, The pseudoflow algorithm: a new algorithm and a new simplex algorithm for the maximum-flow problem, UC Berkeley manuscript, April 1997.

[33] D.S. Hochbaum, A framework for half integrality and 2-approximations with applications to feasible cut and minimum satisfiability, UC Berkeley manuscript, April 1996. Extended abstract version in Proc APPROX98, Lecture Notes in Computer Science 1444, Jansen and Rolim (Editors), Springer-Verlag, 1998, Berlin Heidelberg, pp. 99–110.

[34] D. S. Hochbaum and A. Chen, Performance analysis and best implementations of old and new algorithms for the open-pit mining problem, Oper Res 48 (2000), 894–914.

[35] D. S. Hochbaum and S.-P. Hong, About strongly polynomial time algorithms for quadratic optimization over submodular constraints, Math Prog 69 (1995), 269–309.
[36] D.S. Hochbaum and J. Naor, Simple and fast algorithms for linear and integer programs with two variables per inequality, SIAM J Comput 23 (1994), 1179–1192.
[37] D.S. Hochbaum, N. Megiddo, J. Naor, and A. Tamir, Tight bounds and 2-approximation algorithms for integer programs with two variables per inequality, Math Prog 62 (1993), 69–83.
[38] A. Hoffman and J. Rivlin, "When a team is 'mathematically' eliminated?" Princeton Symposium of Math Programming, 1967, H.W. Kuhn (Editor), Princeton University Press, Princeton, NJ, 1970, pp. 391–401.
[39] P. Huttagosol and R. Cameron, A computer design of ultimate pit limit by using transportation algorithm, Proc 23rd Int APCOM Symp 1992, pp. 443–460.
[40] T.B. Johnson, Optimum open pit mine production scheduling, Ph.D. Thesis, Dept. of IEOR, University of California, Berkeley, May 1968.
[41] T.B. Johnson and W.R. Sharp, A three-dimensional dynamic programming method for optimal ultimate open pit design, U.S. Bureau of Mines, Report of Investigation 7553, 1971.
[42] D.R. Karger and C. Stein, A new approach to the minimum-cut problem, J ACM 43 (1996), 601–640.
[43] Y.C. Kim, Ultimate pit limit design methodologies using computer Models—The state of the art, Mining Eng 30 (1978), 1454–1459.
[44] Y.C. Kim and W.L. Cai, Long range mine scheduling with 0-1 programming, Proc 22nd Int APCOM Symp 1990, pp. 131–144.
[45] V. King, S. Rao, and R. Tarjan, A faster deterministic maximum-flow algorithm, J Alg 17 (1994), 447–474.
[46] S. Koborov, Method for determining optimal open pit limits, Rapport Technique ED 74-R-4, Dept. of Mineral Engineering, Ecole Polytechnique de Montreal, Canada, Feb. 1974.
[47] E. Koenigsberg, The optimum contours of an open pit mine: An application of dynamic programming, Proc 17th Int APCOM Symp, 1982, pp. 274–287.
[48] E.L. Lawler, Combinatorial optimization: Networks and matroids, Holt Rinehart and Winston, New York, 1976.
[49] H. Lerchs and I.F. Grossmann, Optimum design of open-pit mines, Trans C.I.M. 68 (1965), 17–24.
[50] J.W. Mamer and S.A. Smith, Job completion based inventory systems: Optimal policies for repair kits and spare machines, Working paper no. 318, Western Management Science Institute, University of California, 1983.
[51] H. Nagamochi and T. Ibaraki, Computing edge-connectivity in multigraphs and capacitated graphs, SIAM J Discr Math 5 (1992), 54–66.
[52] D. Orlin, Optimal weapons allocation against layered defenses, Naval Res Log Q 34 (1987), 605–617.
[53] M. T. Pana, The simulation approach to open pit design, Proc 5th APCOM Symp, Tucson, AZ, 1965.
[54] J. C. Picard, Maximal closure of a graph and applications to combinatorial problems, Mgmt Sci 22 (1976), 1268–1272.
[55] J.C. Picard and M. Queyranne, Selected applications of minimum cuts in networks, INFOR 20 (1982), 394–422.
[56] J.M.W. Rhys, A selection problem of shared fixed costs and network flows, Mgmt Sci 17:3 (1970), 200–207.
[57] R.H. Robinson and N.B. Prenn, An open pit design model, Proc 15th Int APCOM Symp, 1977, pp. 1–9.
[58] B. Schwartz, Possible winners in partially complete tournaments, SIAM Rev 8 (1966), 302–308.
[59] D. D. Sleator and R. E. Tarjan, A data structure for dynamic trees, J Comput Syst Sci 24 (1983), 362–391.
[60] M. Stoer and F. Wagner, A simple min-cut algorithm, J ACM 44 (1997), 585–591.
[61] G.S. Thomas, Optimisation and scheduling of open pits via genetic algorithms and simulated annealing, Proc 1st Int Symp on Mine Simulation via the Internet, Paper TG085A, http://www.metal.ntua.gr/msslab/MineSim96.
[62] R. Vallet, Optimisation mathematique de l'exploitation d'une mine a ciel ouvert ou le problem de l'enveloppe, Ann Mine Belg Feb. (1976), 113–135.
[63] Q. Wang and H. Sevim, An alternative to parameterization in finding a series of maximum-metal pits for production planning, Proc 24th Int APCOM Symp, 1993, pp. 168–175.
[64] J. Whittle, The facts and fallacies of open pit optimization, manuscript, 1989.
[65] Y. Zhao and Y.C. Kim, A new optimum pit limit design algorithm, Proc 23rd Int APCOM Symp, 1992, pp. 423–434.