Chapter 2

ENUMERATING NEAR-MIN S-T CUTS

Ahmet Balcioglu
Operations Research Dept.
Naval Postgraduate School
Monterey, CA 93943

R. Kevin Wood
Operations Research Dept.
Naval Postgraduate School
Monterey, CA 93943
kwood@nps.navy.mil

Abstract We develop a factoring (partitioning) algorithm for enumerating near-minimum-weight s-t cuts in directed and undirected graphs, with application to network interdiction. "Near-minimum" means within a factor of $1+\epsilon$ of the minimum for some $\epsilon \geq 0$. The algorithm requires only polynomial work per cut enumerated provided that ϵ is sufficiently (not trivially) small, or G has special structure, e.g., G is a complete graph. Computational results demonstrate good empirical efficiency even for large values of ϵ and for general graph topologies.

Keywords: graphs, networks, cuts, enumeration, polynomial-time algorithm

Introduction

Researchers have studied various classes of cuts in graphs and devised efficient algorithms for enumerating these cuts. This paper addresses a particular class of cuts that has not received the same attention as others, specifically, near-minimum-weight minimal s-t cuts. We develop, implement and test an algorithm to enumerate these cuts in directed or undirected graphs.

We focus on *directed graphs* $G = (V, E)$, with positive integer edge weights and two special vertices, a source s and a sink t. An *s-t cut* C is a minimal set of edges whose removal breaks all directed s-t paths; if removal of C breaks all paths but C is non-minimal, it is a *non-minimal s-t cut*. Non-minimal s-t cuts do not interest us in this paper, except in the way that they interfere with our identification of minimal s-t cuts. All cuts discussed are minimal s-t cuts unless otherwise specified; note that a minimum-weight cut must be minimal because all edge weights are positive.

The problem of finding an s-t cut of minimum weight among all possible s-t cuts in G is the *minimum s-t cut problem* (MCP). This paper studies two extensions of MCP, the problem of *enumerating all minimum-weight s-t cuts* in G (AMCP) and the problem of *enumerating all near-minimum s-t cuts* (ANMCP) whose weight is within a factor of $1 + \epsilon$ of the minimum for some $\epsilon \geq 0$. The main contribution of this paper is an efficient procedure for the latter extension, when ϵ is small, or for certain graph topologies. Even when not provably efficient, the algorithm shows good empirical efficiency on our test problems. A cut-enumeration algorithm is "efficient" if the amount of work per cut enumerated is polynomial in the size of G.

The analogs of AMCP and ANMCP in undirected graphs G can also be solved using our techniques. An s-t cut C in an undirected graph is defined just as in a directed graph, the only difference being that the paths broken by C consist of undirected edges. However, if we make the standard transformation that replaces each undirected edge in G by two directed, anti-parallel edges, each with the weight of the original undirected edge, then each s-t cut in the resulting directed graph corresponds directly to an s-t cut in the original graph. Thus, an efficient technique to enumerate s-t cuts in directed graphs will efficiently enumerate s-t cuts in undirected graphs.

Another type of cut can be defined in an undirected graph G. A *disconnecting set* (DS) is a minimal set of edges whose deletion disconnects G. (Often called a "cut," we use "DS" to avoid confusion.) The problems of finding and enumerating certain DSs are related to our problems and will be discussed briefly, so: (a) The problem of finding a minimum-weight DS is denoted MDSP, (b) the problem of enumerating all minimum-weight DSs is denoted AMDSP, and (c) the problem of enumerating all near-minimum-weight DSs is denoted ANMDSP.

In the remainder of this paper, we denote a minimum-weight s-t cut as C_0 and a near-minimum-weight (minimal) s-t cut as C_ϵ. $\mathbf{C}_0(G)$ and $\mathbf{C}_\epsilon(G)$ denote the set of minimum and near-minimum cuts in G,

respectively. We will often substitute "max" and "min" for "maximum" and "minimum," respectively.

A military application, namely network interdiction, first brought ANMCP to our attention; see Wood (1993) and references therein, Boyle (1998) and Gibbons (2000). A "network user" attempts to communicate between vertices s and t in a directed network while an "interdictor," using limited resources (aerial sorties, cruise missiles, etc.), tries to interdict (break, destroy) all s-t paths to prevent communication between s and t. By treating the amount of resource required to interdict an edge as its weight or capacity, the interdictor can solve a max-flow problem and identify a min-weight s-t cut, i.e., min-resource s-t cut, to prevent that communication. It is clear from this application why we are only interested in minimal s-t cuts.

But, there may be secondary criteria, e.g., collateral damage, risk to attacking forces, etc., that the interdictor wishes to consider when determining the best interdiction plan. In this case, near-optimal solutions with respect to the primary criterion can be obtained by solving ANMCP; then those solutions can be evaluated against the secondary criteria for suitability. One of those near-optimal "good solutions" might produce more desirable results than an "optimal solution" obtained by solving MCP or AMCP (Boyle 1998, Gibbons 2000). Integer-programming techniques could substitute for this enumeration approach, but the empirical efficiency of our methods bodes well for enumeration. In fact, the secondary criteria could be incorporated into our recursive cut-enumeration algorithm to force peremptory backtracking, i.e., to help trim the "enumeration tree." This might result in an even more efficient interdiction algorithm.

Another application of ANMCP arises in assessing the reliability and connectivity of networks; see Provan and Ball (1983) and Colbourn (1987).

AMCP and AMDSP have been intensively studied, but ANMCP has not received the same attention. One brute-force approach for ANMCP is to enumerate all s-t cuts, i.e., solve AMCP, and then discard the cuts that do not have near-minimum weight. All s-t cuts can be enumerated efficiently (Tsukiyama et al. 1980, Abel and Bicker 1982, Karzanov and Timofeev 1986, Shier and Whited 1986, Ahmad 1990, Sung and Yoo 1992, Prasad et al. 1992, Nahman 1995, Patvardhan et al. 1995, and Fard and Lee 1999). Unfortunately, this fact cannot lead to an efficient general approach for AMCP or ANMCP because the number of minimal s-t cuts in a graph may be exponential in the size of that graph while the number of minimum and near-minimum cuts may be polynomial. For instance, if G is a complete directed graph with edge weights of 1,

the total number of minimal s-t cuts is $2^{(|V|-2)}$, the number of minimum cuts is 2 and the number of cuts of the next largest size is $2(|V|-2)$.

All minimum s-t cuts can be enumerated efficiently, that is, AMCP can be solved efficiently. Picard and Queyranne (1980) find a max flow in a weighted directed graph G, create the corresponding residual graph, and then demonstrate the one-to-one correspondence of minimum s-t cuts in G to closures in the residual graph. (A closure is a set of vertices with no edges directed out of the set.) They go on to present an algorithm, not necessarily an efficient one, to enumerate these closures and thus all min cuts. Provan and Ball (1983) use the concept of "s-directed minimum cuts" to enumerate minimum s-t cuts in both directed and undirected graphs. However, neither their algorithm nor Picard and Queyranne's may be efficient for directed graphs (Provan and Shier 1996). Gusfield and Naor (1993), Provan and Shier (1996) and Curet et al. (2002) all give efficient algorithms for AMCP based on results from Picard and Queyranne. Provan and Shier's work is related to Kanevsky (1993) who finds all minimum-cardinality "separating vertex sets" as opposed to separating edge sets.

Ramanathan and Colbourn (1987) enumerate "almost-minimum cardinality s-t cuts." They bound the number of cuts enumerated, and the complexity of their algorithm, by $O(m^k n^{k+2})$, where $n = |V|$, $m = |E|$ and where $k \geq 1$ is a constant by which the cardinality of an almost-min cut exceeds the cardinality of a min cut. This algorithm applies only to undirected graphs and has polynomial complexity only if k is fixed.

Karger and Stein (1996) introduce a randomized algorithm for solving ANMDSP by repeated applications of edge contraction: Identify an edge that is probably not part of a near-min-weight DS and merge its endpoints into a single new vertex such that the new graph still contains a near-min DS with high probability. With high probability, their algorithm enumerates all DSs whose weight is within a factor α of the minimum in $O(n^{2\alpha} \log^2 n)$ expected time. They also derive an upper bound $O(n^{2\alpha})$ on the number of these DSs. Karger (2000) later improves this upper bound to $O(n^{\lfloor 2\alpha \rfloor})$. Nagamochi et al. (1997) give a deterministic algorithm for solving ANMDSP based on Karger and Stein's techniques. They show that all near-min DSs can be enumerated in $O(m^2 n + n^{2\alpha} m)$ time. Unfortunately, it is unlikely that this approach can be extended to enumeration problems involving s-t cuts (Karger and Stein 1996).

Vazirani and Yannakakis (1992) propose an algorithm for solving ANMCP and ANMDSP. Their extended abstract claims that the algorithm has polynomial complexity, but that claim is based on this unproven assertion: "Fact: Given a partially specified cut, we can find with one

max-flow computation a minimum weight s-t cut consistent with it." That is, they believe that this problem has polynomial complexity:

P0: Find a min cut C in $G = (V, E)$ containing required edges $\hat{E} \subset E$. In general, their algorithm will need to find cuts that exclude certain edges and include others, but it must efficiently solve the special case of P0, too. In another paper, (Carlyle and Wood 2002), we show P0 to be an NP-complete problem, so the claim of Vazarani and Yannakakis is false. This does not prove that their algorithm is inefficient—they may be able to identify efficiently partially specified cuts that arise within their algorithm according to some special sequence—but, their proofs are incorrect.

Boyle's algorithm (Boyle 1998) for solving constrained network-interdiction problems on undirected planar graphs can be modified to enumerate near-minimum cuts, but generalization to the non-planar case seems unlikely. Gibbons (2000) describes an algorithm for solving ANMCP in directed or undirected graphs, but that algorithm may enumerate a cut more than once. Empirically, the running time and number of cuts enumerated in his algorithm grow rapidly as the size of graph and ϵ increase, so that algorithm is impractical except for small problems.

There is a connection between the problems of enumerating near-min s-t cuts in graphs and enumerating extreme points of polytopes (e.g., Avis and Fukuda 1996, Bussieck, and Lübbecke 1998), because a modified dual of the max-flow linear program can be guaranteed to possess 0-1 extreme-point solutions that identify cuts (e.g., Wood 1993). But the literature on extreme-point enumeration is silent on efficient enumeration of near-optimal extreme points which is analogous to enumerating near-min minimal and non-minimal cuts. Nor does this literature address the enumeration of near-optimal extreme points possessing special properties, which might be analogous to enumerating near-min minimal cuts. Further research on extreme-point enumeration may lead to results applicable to enumerating near-min minimal s-t cuts, but is beyond the scope of this paper.

The discussion above shows the need for additional work on ANMCP, so this paper proposes a new algorithm to solve the problem, and provides theoretical and empirical results on its efficiency. The algorithm first identifies a min-weight s-t cut C_0, and then recursively partitions ("factors") the space of possible cuts, possibly including some non-minimal ones, by forcing inclusion and/or exclusion of edges $e \in C_0$ in subsequent cuts.

1. Preliminaries

Let $G = (V, E)$ be an *edge-weighted directed graph* with a finite set of *vertices* V and a set of ordered pairs of vertices, $E \subseteq V \times V$, called *edges*. An *undirected graph* is defined similarly, except that its edges are unordered pairs from $V \times V$. We typically use e or (u, v) to denote an edge $e = (u, v)$, and we let $n = |V|$ and $m = |E|$. We distinguish two vertices s and t in V as the *source* and *sink*, respectively. Edge weights are specified by a *weight function* $w : E \to Z^+ \backslash \{0\}$. We denote the weight of edge $e = (u, v)$ as w_e or $w(u, v)$ and the vector of edge weights as $\mathbf{w} = (w_{e_1}, w_{e_2}, \ldots, w_{e_m})$.

A directed *s-t path* in G is a sequence of vertices and edges of the form $s, (s, v_1), v_1, (v_1, v_2), v_2, \ldots, v_{k-1}, (v_{k-1}, t), t$. A minimal *s-t cut* in G is a minimal set of edges C whose removal disconnects s from t in G, i.e., breaks all directed *s-t* paths. If C is a proper superset of some *s-t* cut, it is a *non-minimal s-t cut*. When no confusion will result, we use "*s-t* cut" and "cut" interchangeably with "minimal *s-t* cut." The value $w(C) = \sum_{e \in C} w_e$ is the *weight* of cut C.

A *minimum cut* C_0 is an *s-t* cut whose weight, $w_0 = w(C_0)$, is minimum among all *s-t* cuts. All minimum cuts are minimal because edge weights are positive. A *near-minimum minimal cut* C_ϵ is a minimal *s-t* cut whose weight is at most $\bar{w}_\epsilon = (1 + \epsilon)w_0$ for some $\epsilon \geq 0$. $\mathbf{C}_0(G)$ and $\mathbf{C}_\epsilon(G)$ denote the set of minimum and near-minimum (minimal) cuts, respectively.

An *s-t flow* $\mathbf{f}$ in a directed graph G is a function $f : E \to Z^+$ where $0 \leq f(e) \leq w_e$ for all $e \in E$ and $\sum_{(u,v) \in E} f(u, v) = \sum_{(v,u) \in E} f(v, u)$ for all $u \in V \backslash \{s, t\}$. The value of the flow from s to t is $F = \sum_{(s,u) \in E} f(s, u) - \sum_{(u,s) \in E} f(u, s)$. In the *maximum-flow problem*, we wish to find a flow $\mathbf{f}^*$ that yields a maximum value for F, denoted F^*.

As a result of the max-flow min-cut theorem and its proof (e.g., Ahuja et al. 1993, pp. 184-185), we know that $w(C_0) = F^*$. It is also well known that, given any maximum flow $\mathbf{f}^*$, we can identify a minimum cut C_0 in $O(m)$ time.

A rooted tree T is a connected, acyclic, undirected graph in which one node (vertex), called the "root" and denoted by r, is distinguished from the others. A rooted tree, called an *enumeration tree*, will describe the enumeration process used for solving AMCP and ANMCP on a graph G. To avoid confusion with "vertices" in G, we use the term "node" to mean a vertex in an enumeration tree. A node i in tree T with root r is said to be at level (depth) l if the length of the unique path from r to i is of the form $P_i = r, (r, i_1), i_1, \ldots, i_{l-1}, (i_{l-1}, i), i$. Every node along path P_i, except node i, is an *ancestor* of i, and if i is ancestor of j, then

j is a *descendant* of i. For any path P_i, i_{k-1} is the *parent* of i_k, and i_k is the *child* of i_{k-1}.

2. Theoretical Results

This section develops our algorithm for ANMCP through two intermediate stages.

2.1 Basic Algorithm

Algorithm 1 below outlines an approach to solving ANMCP. Some steps may be difficult to implement, but it illustrates the general approach our final algorithm will use.

Algorithm A1
DESCRIPTION: A generic partitioning algorithm for enumerating near-min, minimal s-t cuts.
INPUT: A directed graph $G = (V, E)$, distinct source and sink vertices $s, t \in V$, edge-weight vector $\mathbf{w}$ of positive integers, and tolerance $\epsilon \geq 0$.
OUTPUT: All minimal s-t cuts C_ϵ such that $w(C_\epsilon) \leq (1+\epsilon)w(C_0)$ where C_0 is a min-weight cut of G.

```
begin
  Find a min-weight s-t cut C_0 in G;
  w̄_ε ← ⌊(1 + ε)w(C_0)⌋;
  E^+ ← ∅;                          /* set of edges to be included */
  E^- ← ∅;                          /* set of edges to be excluded */
  EnumerateA1(G,s,t,w,E^+,E^-, w̄_ε);
end

Procedure EnumerateA1(G,s,t,w,E^+,E^-,w̄_ε)
begin
  Step A: Let C' be min-weight minimal cut in G such that
      E^+ ⊆ C' and E^- ∩ C' = ∅;
  if ( no such cut exists ) return;
  if ( w(C') > w̄_ε ) return;
  Step B: print( C' );
  for ( each edge e ∈ C'\E^+ ) begin;
    E^- ← E^- ∪ {e};
    EnumerateA1(G, s, t, w, E^+, E^-, w̄_ε);
    E^- ← E^-\{e};
    E^+ ← E^+ ∪ {e};
  endfor;
```

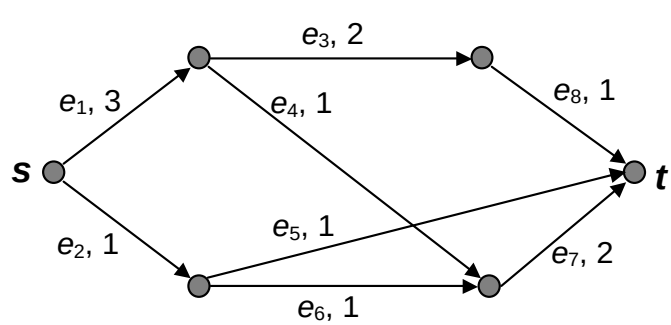
return;
end.

Algorithm A1 begins by finding an initial min-weight cut C_0 and its weight $w(C_0)$. The algorithm then calls the procedure *EnumerateA1* which attempts to find a new min cut by *processing* the edges of the initial cut such that the edges are forced into (*included in*) or out of (*excluded from*) any new near-min cuts. Suppose that $C_0 = \{e_1, e_2, \ldots, e_k\}$ is the initial minimum cut. Based on this cut, the set of near-min cuts, $\mathbf{C}_\epsilon(G)$, is partitioned as

$$\begin{aligned}\mathbf{C}_\epsilon(G) \quad = \quad & [\mathbf{C}_\epsilon(G) \cap \bar{e}_1)] \cup [\mathbf{C}_\epsilon(G) \cap e_1 \cap \bar{e}_2)] \\ & \cup [\mathbf{C}_\epsilon(G) \cap e_1 \cap e_2 \cap \bar{e}_3] \\ & \cup \cdots \cup [\mathbf{C}_\epsilon(G) \cap e_1 \cap e_2 \cap \cdots \cap e_{k-1} \cap \bar{e}_k] \\ & \cup [\mathbf{C}_\epsilon(G) \cap e_1 \cap \cdots \cap e_k], \end{aligned} \tag{2.1}$$

where $\mathbf{C}_\epsilon(G) \cap e_1 \cap e_2 \cap \cdots \cap e_{k'-1} \cap \bar{e}_{k'}$ is shorthand notation for a set that contains all near-min cuts incorporating e_1 through $e_{k'-1}$ but not $e_{k'}$. The cuts in this partition, except for the unique cut of the last term which has already been found as C_0, are identified by recursively calling *EnumerateA1* with the argument sets E^+ and E^-, where E^+ denotes included edges and E^- denotes excluded edges. The procedure calls itself recursively for every edge of the locally minimum cut that has not already been forced into that cut at higher level in the enumeration. ("Local" and "locally" refer to flows and cuts defined on graphs within the enumeration tree.) The procedure backtracks when it determines that no acceptable cuts remain below a given node.

Figure 2.1. Sample graph. Numbers represent edge weights.



To illustrate how this enumeration works, suppose we wish to solve ANMCP on the graph of Figure 2.1 for $\epsilon = 0.4$. The associated enumeration tree (an instance of a rooted tree) for this problem is given in Figure 2.2. The enumeration algorithm first finds a minimum cut $\{e_2, e_4, e_8\}$ at the root node (level 0), and then recursively partitions the solution space via $\{\bar{e}_2\}$, $\{e_2, \bar{e}_4\}$ and $\{e_2, e_4, \bar{e}_8\}$. Once an edge of a cut at some node k has been processed, it will never be processed again at any descendant node of k, because its status as "included" or "excluded" with respect to the current cut has been fixed at node k. The branches with $\{\bar{e}_2\}$,$\{e_2, \bar{e}_4\}$ and $\{e_2, e_4, \bar{e}_8\}$ correspond to searches for a new min cut by processing the edges as described. If a search is successful, it defines a *productive* node where a new near-min cut is identified and that cut's unprocessed edges are recursively processed. Otherwise, the search leads to a *terminal* node, where the algorithm backtracks. The procedure is correct because it implements inclusion-exclusion (Equation 2.1) in a straightforward, recursive manner. The actual implementation of the algorithm could be difficult, and efficiency poor however, because edge inclusion may be difficult to ensure (although edge exclusion is easy). This topic is explored further in Section 2.2.

A "relaxed" version of Algorithm A1, denoted "Algorithm A2," can be defined by modifying Steps A and B to:

> Step A: Let C' be min-weight minimal or non-minimal s-t cut in G such that $E^+ \subseteq C'$ and $E^- \cap C' = \emptyset$;
>
> Step B: **if** (C' is non-minimal) print(C');

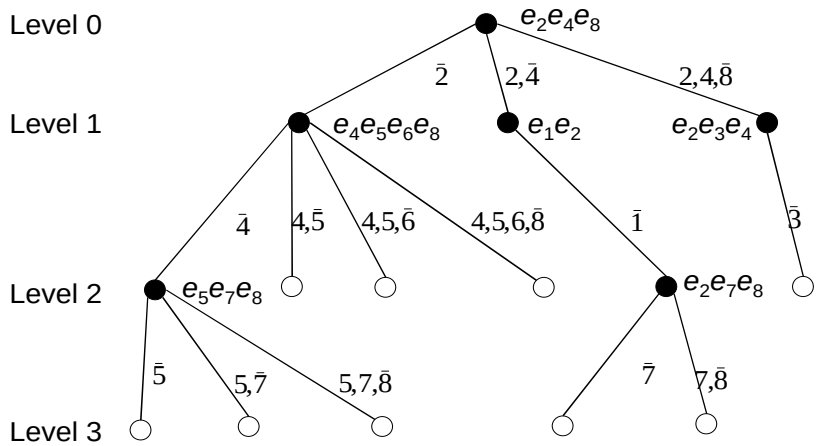
Algorithm A2 may waste time working with non-minimal cuts because it partitions the space consisting of all minimal cuts and possibly some non-minimal ones, It does solve ANMCP, however, because it prints only the minimal cuts. Our final implementable algorithm, Algorithm B, closely mimics Algorithm A2.

Note that Algorithm A2 will not necessarily identify all non-minimal cuts C' satisfying $w(C') \leq \bar{w}_\epsilon$, which is good because their identification wastes computational effort. Consider, for instance, the last term of equation (2.1), $\mathbf{C}_\epsilon(G) \cap e_1 \cap \cdots \cap e_k$ when $C_0 = \{e_1, \ldots, e_k\}$. This subset of the partition includes all non-minimal cuts containing $C_0 = \{e_1, \ldots, e_k\}$. However, Algorithm A2 does not partition $\mathbf{C}_\epsilon(G) \cap e_1 \cap \cdots \cap e_k$ further, because any cut it might contain other than C_0 is a superset of C_0 and must therefore be non-minimal.

2.2 An Implementable Algorithm

Algorithm B, below, implements a variant of Algorithm A2. We *quasi-exclude* an edge e from every cut in a subtree of the enumeration tree

Figure 2.2. Enumeration tree for the graph of Figure 2 with $\epsilon = 0.4$. The enumeration scheme is represented from top to bottom and left to right. Each filled-in node corresponds to a cut whose weight is no larger than $\bar{w}_\epsilon = \lfloor (1+0.4)w(C_0) \rfloor = \lfloor (1+0.4)3 \rfloor = 4$. The edges of the cut are listed next to the node. Numbers with bars over them represent the number of the edge excluded from a cut; numbers without bars represents edges to be included. Unfilled nodes are terminal nodes where the algorithm backtracks.



by simply setting $w_e = \infty$, represented by a suitably large integer. Every near-min cut in G must have a finite weight, so setting $w_e = \infty$ effectively eliminates cuts containing e. This means that quasi-exclusion implements true exclusion. The graph G with edge e quasi-excluded is denoted $G - e$.

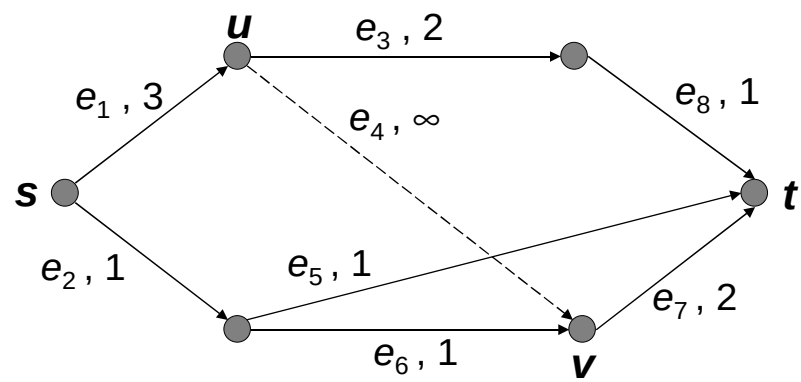
We *quasi-include* $e = (u, v)$ by effectively adding two edges to G, (s, u) and (v, t), both with infinite weights. The graph G with edge e quasi-included is denoted $G + e$. Now, any cut of a graph must contain at least one edge from every path, so any cut of $G+e$ must contain (s, u), (u, v) or (v, t), and any finite-weight cut of $G+e$ must contain $e = (u, v)$. (We can omit (s, u) if $u = s$, and omit (v, t) if $v = t$.) In reality, we implement quasi-inclusion of $e = (u, v)$ by temporarily treating u as an additional source and v as an additional sink. Unfortunately, quasi-inclusion can create modified graphs with minimal cuts that correspond to non-minimal cuts in the original graph G. (See the example below.) Thus, Algorithm B must screen for non-minimal cuts, just as Algorithm A2 does.

Figure 2.3 illustrates the quasi-inclusion and -exclusion of edges. $G + E^+ - E^-$ denotes G with E^+ quasi-included and E^- quasi-excluded. The caption describes an example of how quasi-inclusion of an edge in the depicted graph can create a modified graph in which a minimal cut is non-minimal in the original graph G.

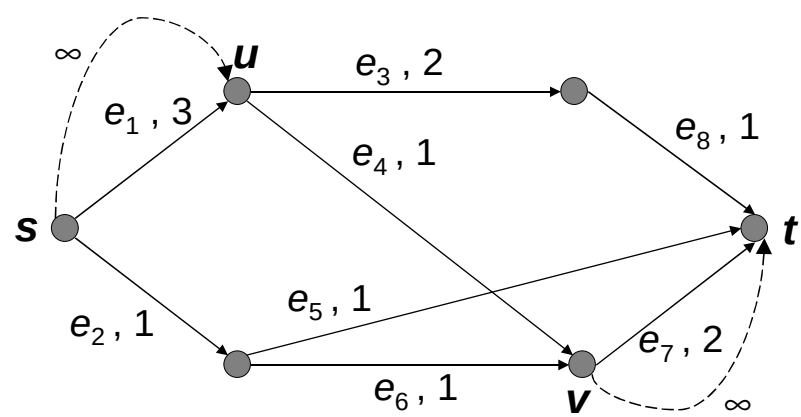
Note that Algorithm B calls a subroutine *MaxFlow* which is assumed to return a min cut C' in the current graph G' along with the weight of that cut $w(C')$, which equals the value of the max flow. Also note that, in order to keep the notation similar to the earlier algorithms, Algorithm B repeatedly modifies a copy of the original graph G, rather than recursively modifying and "unmodifying" a single copy of G. The actual Java implementation of Algorithm B uses the latter, more efficient approach.

Algorithm B
DESCRIPTION: An implementable version of Algorithm A2 to solve ANMCP;
INPUT:A directed graph $G = (V, E)$, s, t, $\mathbf{w}$, and ϵ.
OUTPUT: All minimal s-t cuts C_ϵ in G with
$w(C_\epsilon) \leq (1 + \epsilon)w(C_0)$.
begin
$[w_0, C_0] \leftarrow MaxFlow(G, s, t, \mathbf{w})$;
$\bar{w}_\epsilon \leftarrow \lfloor (1 + \epsilon)w_0 \rfloor$;
$E^+ \leftarrow \emptyset$; /* set of edges to be included */
$E^- \leftarrow \emptyset$; /* set of edges to be excluded */
$EnumerateB(G, s, t, \mathbf{w}, E^+, E^-, \bar{w}_\epsilon)$;

Figure 2.3. Quasi-inclusion and -exclusion of an edge from cuts in a graph G. (a) Edge e_4 is quasi-excluded from every cut by setting $w_4 = \infty$. (b) Edge e_4 is quasi-included in every cut by, in effect, adding infinite-weight edges (s, u) and (v, t) to G. Quasi-inclusion can create difficulties (not indicated explicitly in the figures): $\{e_1, e_2, e_8\}$ forms a minimal cut of $G + e_8$ (G with edge e_8 quasi-included), but $\{e_1, e_2, e_8\}$ is not a minimal cut of G.



(a)



(b)

```
end.

Procedure EnumerateB(G, s, t, w, E+, E−, w̄_ε)
begin
  w' ← w;   G' ← G;
  for ( each edge e = (u, v) ∈ E−) w'(u, v) ← ∞;
  for ( each edge e = (u, v) ∈ E+ ) begin
    add artificial edge (s, u) to G' and let w'(s, u) ← ∞;
    add artificial edge (v, t) to G' and let w'(v, t) ← ∞;
  endfor;
    /* G' and w' are now interpreted to include artificial edges */
  [w', C'] ← MaxFlow(G', s, t, w');
  if ( w' > w̄_ε ) return;
  if ( C' is minimal in G ) print (C');
  for ( each edge e ∈ C'\E+ )  begin
    E− ← E− ∪ {e};
    EnumerateB(G, s, t, w, E+, E−, w̄_ε);
    E− ← E−\{e} ;
    E+ ← E+ ∪ {e};
  endfor;
  return;
end.
```

2.3 Correctness of Algorithm B

We assume the correctness of Algorithm A2. Algorithm B begins by finding a min cut in G and determining $\bar{w}_\epsilon$ using a max-flow algorithm, all in an obviously correct manner. That is, the main routine of Algorithm B correctly implements the main routine of Algorithm A2. Then, where Algorithm A2 finds a minimal or non-minimal, min-weight cut that includes edges in E^+ and excludes edges in E^-, Algorithm B solves a max-flow problem and finds a min cut C' in $G + E^+ - E^-$. C' may or may not be a minimal cut in the original graph G, but its deletion from G does disconnect s from t in G, because (a) $C' \subseteq E$ because artificial edges have infinite weight and thus cannot be contained in C', and (b) every s-t path in $G + E^+ - E^-$ is disconnected by deleting C', and (c) every path in G is also a path in $G + E^+ - E^-$ by construction; thus, all paths in G are disconnected by deleting C' from that graph. Algorithm B is clearly finite and will be correct as long as it correctly partitions the space of all minimal cuts in G (along with some non-minimal cuts perhaps).

The partitioning will be correct if no non-minimal cuts that Algorithm A2 might identify are lost in the calls to *EnumerateB* and no non-minimal cuts are repeated. The following two lemmas suffice to prove this.

Lemma 1 *Let C be a finite-weight set of edges in G and let E^+ and E^- be quasi-inclusion and quasi-exclusion sets, respectively, produced while running Algorithm B. Suppose that $C \cap E^+ = E^+$ and $C \cap E^- = \emptyset$. Then C is a minimal cut of G only if C is also a finite-weight minimal cut of $G + E^+ - E^-$.*
Proof: Since $C \cap E^- = \emptyset$, C has finite weight in $G + E^+ - E^-$. $G + E^+$ has the same topological structure as $G + E^+ - E^-$, so we need only be concerned with the former. Now, C is clearly a cut in $G + E^+$ because the fact that $C \cap E^+ = E^+$ means that no edges crossing from the s side of the cut in G to the t side have been added; only edges from the t side of the cut to t or from s to some vertex on the s side of C could have been added through quasi-inclusion of E^+. So, C is a cut in $G + E^+$, and it must be minimal because every path in G is also a path in $G + E^+$. ∎

Lemma 2 *Let C be a set of edges in G and let E^+ and E^- be, respectively, quasi-inclusion and quasi-exclusion sets produced while running Algorithm B. Suppose that $C \cap E^+ \neq E^+$ or $C \cap E^- \neq \emptyset$. Then, C is a not a finite-weight minimal cut of $G + E^+ - E^-$.*
Proof: We know that quasi-exclusion properly implements edge exclusion. Thus, C cannot be a finite-weight minimal cut of $G + E^+ - E^-$ if some edge of C has been excluded, i.e., if $C \cap E^- \neq \emptyset$. From the discussion on quasi-inclusion, we know that all finite-weight minimal cuts of $G + E^+$ must contain E^+, i.e., C cannot be a finite-weight minimal cut of $G + E^+ - E^-$ if $C \cap E^+ \neq E^+$. ∎

The following theorem results.

Theorem 1 *Algorithm B solves ANMCP.* ∎

2.4 Complexity of Algorithm B

We show below that Algorithm B has polynomial complexity when G and/or $\mathbf{w}$ satisfy certain conditions. This discussion ignores minimality testing after the next lemma because it cannot add to Algorithm B's worst-case complexity for any problem, assuming that at least $O(m)$ work must arise at every node of the enumeration tree:

Lemma 3 *Testing whether or not a set of edges C in G is a minimal s-t cut can be accomplished in $O(m)$ time.*

Proof: Assume that C is a minimal or non-minimal cut and mark all vertices as "unreachable." Now, perform a breadth-first search starting at s, trying to reach as many vertices as possible without traversing any cut edges $(u, v) \in C$. Mark the vertices reached as "reachable from s." Conduct a similar search, traversing edges backward from t, marking the vertices reached as "can reach t." (If a backward search from t can reach v, then v can reach t along a directed path.) By definition, C is minimal if and only if, for all edges $(u, v) \in C$. u is reachable from s and v can reach t. The amount of work involved in the two searches and testing the edges in C is clearly $O(m)$. ∎

2.4.1 Complexity Analysis of Min-Cut Enumeration.

We first analyze the complexity of enumerating minimum cuts ($\epsilon = 0$), since this is an important special case of near-min cut enumeration. Consider the enumeration tree of Figure 2.2. Every node in that tree is either *productive* and defines a new cut (the filled-in nodes), or it is an *unproductive* terminal node from which backtracking occurs immediately. In general, the quasi-inclusion technique can result in unproductive non-terminal nodes because it can identify a non-minimal cut and be unable to backtrack immediately. Fortunately, any non-minimal cut encountered while solving AMCP must correspond to a terminal node and an efficient procedure results.

We know that the worst-case complexity of solving, "from scratch," an initial max-flow problem on $G = (V,E)$ is $O(f(n, m))$, where $f(n, m)$ is a polynomial function of $n = |V|$ and $m = |E|$. (For instance, the first polynomial-time algorithm for max flows, a flow-augmenting path algorithm due to Edmonds and Karp (1972), has worst-case complexity $O(nm^2)$; the more modern pre-flow push algorithm due to Goldberg and Tarjan (1988) has $O(nm \log(n^2/m))$ worst-case complexity.) At each non-root node of the enumeration tree, the local max flow can be obtained by performing flow augmentations starting with the feasible flow from the parent node. (A feasible flow $\mathbf{f}$ in G must be feasible for $G - E^- + E^+$ because the latter graph is obtained from the former by increasing the capacity on certain edges, specifically $e \in E^-$, and adding some other edges, specifically $e \in E^+$. Neither of these operations reduces the capacity on any path in the original graph G.) Each flow augmentation requires $O(m)$ work using breadth-first search in a standard fashion, but the total amount of work performed at each node can be limited to $O(m)$, because (a) if the first search does not find a flow-augmenting path, a new min cut has been identified (this fact follows from the standard constructive proof of the max-flow min-cut theorem, e.g., Ahuja, et al. 1993, pp. 184-185), and (b) if a flow-augmenting path

is found, the locally maximum flow is at least $w_0 + 1$ and the algorithm can backtrack immediately. (The algorithm must be modified slightly to enable this "peremptory backtracking.") Thus the number of productive nodes is $|\mathbf{C}_0(G)|$.

Now, each non-terminal node can generate at most n child nodes assuming G has no parallel edges, and thus each productive node can generate at most n unproductive (terminal) nodes. Therefore, the total number of nodes generated is bounded by $n|\mathbf{C}_0(G)|$. The amount of work to generate each node except the first is $O(m)$, and the amount of work to generate the first node is $O(f(n,m))$ so we have the following result.

Theorem 2 *Algorithm B with $\epsilon = 0$ finds all minimum-weight s-t cuts (solves AMCP) in $O(f(n,m) + mn|\mathbf{C}_0(G)|))$ time.* ∎

This shows that Algorithm B is theoretically efficient for AMCP since only a polynomial amount of work is expended for each cut enumerated. The Algorithm is admittedly less efficient for solving AMCP than are some other algorithms from the literature: For instance, the algorithm of Provan and Shier (1996) solves AMCP in $O(f(n,m)+(m+n)(|\mathbf{C}_0(G)|))$ time. Nevertheless, our algorithm has several advantages in that (a) it is easy to implement, (b) its empirical efficiency is quite good (see Section 3) and, (c) it extends to near-min cut enumeration, i.e., to solving ANMCP, by simply setting $\epsilon > 0$.

2.4.2 Complexity Analysis of Near-Min Cut Enumeration.

The argument of the previous section leads quickly to this corollary:

Corollary 1 *If $\min_{e\in E} w_e > w(C_0)\epsilon$, then Algorithm B solves ANMCP in $O(nf(n,m)|\mathbf{C}_0(G)|)$ time.*

Proof: "$f(n,m)$" is a multiplier on $|\mathbf{C}_0(G)|$ here because we are unable to bound the number of flow augmentations required in *EnumerateB* to establish a new max flow: We simply resort to the bound implied by solving each max-flow problem from scratch. As before, the multiplicative factor n will bound the number of terminal nodes emanating from a productive node.

Just as in Theorem 2, the statement of the Corollary will be true if Algorithm B can always backtrack when it finds a non-minimal cut, that is, if every non-minimal cut corresponds to a terminal node. This is true because any non-minimal cut must have weight at least $w(C_0) + \min_{e\in E} w_e > w(C_0) + w(C_0)\epsilon = w(C_0)(1+\epsilon) = \bar{w}_\epsilon$, ∎

So, Algorithm B is efficient when ϵ is sufficiently small. It is also efficient when G has special topology.

Theorem 3 *Algorithm B solves ANMCP in $O(nf(n,m)|\mathbf{C}_0(G)|)$ time whenever G contains an edge of the form (s,u) for each $u \in V\backslash\{s,t\}$ and an edge of the form (v,t) for each $v \in V\backslash\{s,t\}$.*
Proof: The statement will be true if quasi-inclusion and -exclusion never change the vertex-to-vertex connectivity of a graph, because then any minimal cut of $G + E^+ - E^-$ must be a minimal cut of G. But quasi-inclusion never changes connectivity irrespective of graph topology. Quasi-exclusion for $e = (u,v)$ always adds edges of the form (s,u) and (v,t), but as specified, G already contains such edges. ∎

Corollary 2 *Algorithm B solves ANMCP in $O(nf(n,m)|\mathbf{C}_0(G)|)$ time when G is a complete directed graph or complete acyclic graph with $s < t$ in the acyclic (topological) ordering of the vertices.* ∎

Of course, the problem is trivial if $s > t$.

By the arguments of the preceding section, the number of nodes in enumeration tree should be bounded by $n(|\mathbf{C}_\epsilon(G)| + |\mathbf{C}''|)$, where $\mathbf{C}''$ denotes the set of near-minimum, non-minimal cuts identified as nodes in Algorithm B's enumeration tree where immediate backtracking is not allowed, i.e., the set of unproductive, non-terminal nodes in that tree.

The test for non-minimality takes $O(m)$ time at each node by Lemma 3. The search for a local max flow might require multiple flow augmentations and might be as hard as solving a max-flow problem from scratch. Therefore, the work expended at every node is $O(f(n,m)+m) = O(f(n,m))$.

If we could backtrack whenever a non-minimal cut was identified, then we could state that $\mathbf{C}'' = \emptyset$ and the resulting complexity for the whole algorithm would be $O(nf(n,m)|\mathbf{C}_\epsilon(G)|)$. But, it is easy to show by example that backtracking when a non-minimal cut is encountered can result in the loss of some valid minimal cuts. Thus, Algorithm B must continue partitioning, even on non-minimal cuts, until it can backtrack based on cut weight. This results in a complexity of $O(nf(n,m)(|\mathbf{C}_\epsilon(G)| + |\mathbf{C}''|))$, which may not be polynomial if $|\mathbf{C}''|$ is exponentially larger than $\mathbf{C}_\epsilon(G)$. Therefore, the worst-case complexity of Algorithm B for arbitrary ϵ and/or arbitrary graph topology is not well determined. We leave this complexity issue as a topic for future research.

3. Computational Results

This section reports on computational experiments with Algorithm B to demonstrate its empirical efficiency for solving both AMCP and ANMCP. We test Algorithm B on both weighted and unweighted grid graphs and on several problem instances from the literature.

Algorithm B is written and compiled using the Java 1.2.2 programming language (Sun Microsystems 1998). All tests are performed on a personal computer with a 733 MHz Pentium III processor and 128 MB of RAM, running under the Windows 98 SE operating system.

3.1 Efficient Implementation of Algorithm B

We have described Algorithm B in a simple form for clarity, but there are several modifications that improve its performance in practice. As discussed in Section 2.4.1, the *MaxFlow* routine of Algorithm B is implemented to solve an "incremental" max-flow problem. Specifically, $\mathbf{f}^*$ in G is a feasible flow in $G + E^+ - E^-$, so a flow-augmenting path algorithm operating on a graph at some non-root node in the enumeration tree simply begins with the maximum flow from the parent node, rather than starting with $\mathbf{f} = \mathbf{0}$. (See Ahuja et al. 1993, pp. 180-184.)

Another issue in an efficient implementation is edge inclusion. In theory, we quasi-include an edge (u, v) by adding infinite-weight edges (s, u) and (v, t) to the graph, but in practice we simulate this by simply treating u as an additional source and v as an additional sink.

Algorithm B also incorporates "peremptory backtracking" from within the *MaxFlow* routine. In particular, that routine does not need to solve a max-flow problem to completion if it augments enough flow to learn that the local max flow exceeds $\bar{w}_\epsilon$, which implies that any locally minimum cut C' must have $w(C') > \bar{w}_\epsilon$. When this situation occurs, *MaxFlow* halts and returns the current feasible flow value F, which causes *EnumerateB* to return immediately.

The rest of the implementation is straightforward. We use forward and reverse star representation of G as our data structure (Ahuja et al. 1993, pp. 35-38) and a variant of the shortest flow-augmenting-path algorithm of Edmonds and Karp (1972) for solving max-flow problems. (More sophisticated algorithms would speed computations somewhat, but this algorithm is more than adequate to verify the usefulness of our methodology.)

3.2 Test Problems

Our literature search has not uncovered any particular problem family designed for testing algorithms for AMCP and ANMCP, except for Grid Graph Families (GGFs) (Curet et al. 2002, Gibbons 2000), which we will explore. Additionally, we have modified some DIMACS problems (The Center for Discrete Mathematics and Theoretical Computer Science, DIMACS 1991) and several problem classes from Levine (1997) to test Algorithm B.

We have coded a GGF generator (GGFGEN) in Java to generate grid graphs. The height H of the grid measured in nodes, and its length L in nodes, determine the size of the generated graph. One other parameter is q, which indicates whether the graph is weighted ($q = 1$) or unweighted ($q = 0$). "Unweighted" simply means that all edge weights are 1 or ∞: For both weighted and unweighted graphs, the edges beginning at s and ending at t have infinite weights. Every vertex $u \in V \backslash \{s, t\}$ is connected to each adjacent vertex v (vertically and horizontally, assuming such adjacent vertices exist) with two directed edges, (u, v) and (v, u). Edge weights for weighted graphs are pseudo-random, uniformly distributed integer weights in the range [1,10]. GGFGEN produces a directed graph with $HL + 2$ vertices and $4HL - 2L$ edges. Figure 7 shows a graph generated by GGFGEN with inputs $H = 3$, $L = 4$, $q = 0$. Table 2.1 specifies the problems instances that are tested.

Figure 2.4. An unweighted, directed grid graph Generated by GGFGEN with inputs $H = 3$, $L = 4$, and $q = 0$. Edges incident to s and t have infinite weights. Other weights are all 1. Bi-directional edges between u and v represent two directed edges, (u,v) and (v,u).

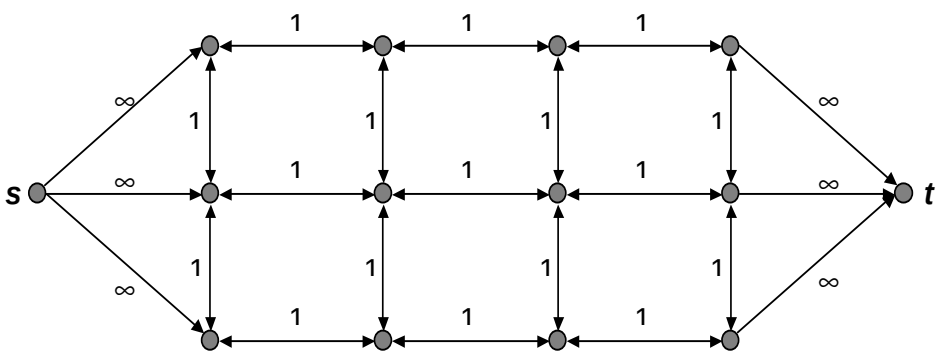


Table 2.1. Problem groups for GGF. This table shows the GGF input graphs and parameter settings with which we test Algorithm B. "$q = 0$" indicates unweighted graphs, and "$q = 1$" indicates weighted ones.

Problem Name	H	L	ϵ	q
GGF-square	5,10,15,20,25, 30,40,. . ., 90	5,10,15,20,25, 30,40,. . .,90	0.0, 0.05 .10, .15	0, 1
GGF-long	25	100,125, . . .,250	0.0	0

Table 2.2. Problem types for DBLCYCLEGEN and AD. Graphs of type DBLCYC-I are generated with DBLCYCLEGEN and have $n = 500$ vertices and 1000 undirected edges on two interleaved cycles; the undirected edges are converted to $m = 2000$ directed edges. Graphs of type AD are fully dense, directed acyclic graphs with $n = 50$ and $m = 1225$ unweighted edges.

Problem Name	n	ϵ	q
DBLCYC-I (Levine 1997)	500	0.00, 0.10, 1.25, 1.50, 1.75, 2.00	0, 1
AD (DIMACS 1991)	50	0.10, 0.20, ..., 0.70	0

We have also chosen two other graph generators from the literature, implemented in the C programming language and available via Internet for research use. The first is the Double-Cycle Generator (DBLCYCLEGEN) (Levine 1997), which generates undirected graphs that we convert to directed graphs. The single input parameter for DBLCYCLEGEN is $n = |V|$. DBLCYCLEGEN generates two interleaved cycles on n vertices: The outer cycle includes all n vertices with edge weights of 1000 and 997, and the inner cycle connects every third vertex of the outer cycle with the edges of weights 1 or 4. Vertices s and t are chosen to be as distant from each other as possible. A minimum cut lies in the middle of the graph with a weight of 2000 and there are many near-min cuts with weight 2006.

The second generator is the Acyclic Dense (AD) graph generator from DIMACS (1991). AD takes n as its input parameter and generates a fully dense, directed acyclic graph with n vertices and $m = n(n-1)/2$ edges. We replace the pseudo-randomly generated edge weights in AD with unit weights to observe the behavior of our algorithm on the underlying topological structure. In all cases, $s = 1$ and $t = n$ in the acyclic ordering of the vertices. Table 2.2 gives the generated problem types for DBLCYCLEGEN and AD.

3.3 Experiments on Unweighted Graphs

Table 2.3 presents run times of Algorithm B on GGF instances for solving AMCP. It takes less than 1 second for Algorithm B to identify all minimum cuts in grid graphs with up to 402 vertices and 1,560 edges. The number of calls to *MaxFlow*—this corresponds to the number of nodes in the enumeration tree—increases roughly linearly with n.

Table 2.4 summarizes the results for ANMCP on GGF-square instances with $\epsilon = 0.05$, 0.10, and 0.15. Solution times are expected to increase as ϵ increases because the number of cuts in any graph might

Table 2.3. Run times (in CPU seconds) for Algorithm B solving AMCP on unweighted instances of GGF-square and GGF-long graphs. GGF$H\times L$ denotes a GGF graph with an $H\times L$ grid of vertices. "Non-min'l cuts" denotes the number of non-minimal cuts encountered at non-terminal nodes of the enumeration tree. As predicted by Theorem 2, this number is 0 for AMCP. "Calls to MF" indicates the number of calls to *MaxFlow* and thus the total number of nodes in the enumeration tree.

Problem Name	n	m	$\lvert C_0\rvert$	$\lvert\mathbf{C}_0(G)\rvert$	Non-min'l Cuts	Calls to MF	Run Time (sec.)
GGF10×10	102	380	10	9	0	92	0.1
GGF20×20	402	1560	20	19	0	382	0.1
GGF30×30	902	3540	30	29	0	872	0.1
GGF40×40	1602	6320	40	39	0	1562	0.2
GGF50×50	2502	9900	50	49	0	2452	0.3
GGF60×60	3602	14280	60	59	0	3542	0.5
GGF70×70	4902	19460	70	69	0	4832	2.1
GGF80×80	6402	25440	80	79	0	6322	5.8
GGF25×100	2502	9800	25	99	0	2477	0.2
GGF25×125	3127	12250	25	124	0	3102	1.2
GGF25×150	3752	14700	25	149	0	3727	3.0
GGF25×175	4377	17150	25	174	0	4352	5.7
GGF25×200	5002	19600	25	199	0	4977	9.0
GGF25×225	5627	22050	25	224	0	5602	13.0
GGF25×250	6252	24500	25	249	0	6227	17.6

Table 2.4. Computational results for Algorithm B solving ANMCP on instances of unweighted GGF-square graphs with $\epsilon = 0.05, 0.10, 0.15$. $|C^+|$ denotes the cardinality of the largest acceptable cut.

Problem Name	n	m	$\|C_0\|$	$\|C^+\|$	$\|\mathbf{C}_\epsilon(G)\|$	Non-min'l Cuts	Calls to MF	Run Time (sec.)
$\epsilon = 0.05$								
GGF5×5	27	90	5	5	4	0	22	0.1
GGF10×10	102	380	10	10	9	0	92	0.1
GGF15×15	227	870	15	15	14	0	212	0.1
GGF20×20	402	1560	20	21	703	0	7906	1.5
GGF25×25	627	2450	25	26	1128	0	15506	3.9
GGF30×30	902	3540	30	31	1653	0	26856	12.0
$\epsilon = 0.10$								
GGF5×5	27	90	5	5	4	0	22	0.1
GGF10×10	102	380	10	11	153	0	956	0.1
GGF15×15	227	870	15	16	378	0	3306	0.5
GGF20×20	402	1560	20	22	13319	0	113090	20.1
GGF25×25	627	2450	25	27	27014	0	274550	74.4
GGF30×30	902	3540	30	33	924723	378	8911698	4535.1
$\epsilon = 0.15$								
GGF5×5	27	90	5	5	4	0	22	0.1
GGF10×10	102	380	10	11	153	0	956	0.2
GGF15×15	227	870	15	17	5264	0	35905	4.9
GGF20×20	402	1560	20	23	168283	153	1202033	215.3
GGF25×25	627	2450	25	28	431728	253	3621978	973.3
GGF30×30	902	3540	30	34	13465371	21843	113463496	46395.8

be exponential in the size of the graph. The algorithm is quite efficient for modest-size grid graphs with modest values of ϵ. Compared with Gibbons' results for ANMCP (Gibbons 2000), our results show a vast reduction in calls to *MaxFlow* and in run times.

Table 2.5 presents results for an unweighted AD graph. Corollary 2 requires that Algorithm B not generate any unproductive non-terminal nodes for the AD topology, and results displayed in the table provide empirical verification of this.

3.4 Experiments on Weighted Graphs

Here we use the GGF-square problems with edge weights being pseudo-randomly generated integers in the range [1,10]. Results for minimum

Table 2.5. Computational results on an unweighted, acyclic dense graph(AD) with various threshold levels ϵ. This is a fully dense graph with 50 vertices and 1225 edges. As predicted by Corollary 2, no non-minimal cuts are generated at non-terminal nodes, irrespective of ϵ.

ϵ	$\lvert C_0\rvert$	$\lvert\mathbf{C}_\epsilon(G)\rvert$	Non-min'l Cuts	Calls to MF	Run Time (sec.)
0.00	49	49	0	1275	0.3
0.10	49	544	0	13650	3.1
0.20	49	4063	0	101625	25.7
0.30	49	19798	0	495000	134.9
0.40	49	75893	0	1897375	542.9
0.50	49	249270	0	6231800	1861.6
0.60	49	730603	0	18265125	5544.8
0.70	49	1962849	0	49071275	15126.6

Table 2.6. Computational results for min-cut enumeration (AMCP) on weighted, GGF-square problems. As in the unweighted case, no non-minimal cuts are encountered. All min cuts are identified in less than one second for these instances.

Problem Name	n	m	w_0	$\lvert\mathbf{C}_0(G)\rvert$	Non-min'l Cuts	Calls to MF	Run Time (sec.)
GGF5×5w	27	90	17	1	0	7	0.1
GGF10×10w	102	380	42	2	0	92	0.1
GGF15×15w	227	870	45	1	0	19	0.1
GGF20×20w	402	1560	69	1	0	32	0.1
GGF25×25w	627	2450	87	1	0	33	0.1
GGF30×30w	902	3540	108	6	0	217	0.3

and near-minimum cut enumeration are summarized in Tables 2.6 and Table 2.7, respectively.

Finally, we test Algorithm B on the DBLCYC-I problems with ϵ ranging from 0.0 to 2.0. These are the only problems where Algorithm B encounters substantial numbers of non-minimal cuts from which immediate backtracking would be incorrect. At $\epsilon = 1.25$, the ratio of the number of near-min non-minimal cuts (encountered at non-terminal nodes) to the number of near-min minimal cuts jumps dramatically; see Table 2.8.

In summary, computational results above show that Algorithm B performs quite well on a variety of graph types. However, non-minimal cuts defining unproductive, non-terminal nodes in the enumeration tree can

Table 2.7. Computational results for near-minimum cut enumeration (ANMCP) on weighted GGF-square problems. w_0 is the minimum cut weight and $\bar{w}_\epsilon$ is the weight of the largest acceptable cut.

Problem Name	$\|V\|$	$\|E\|$	w_0	$\bar{w}_\epsilon$	$\|\mathbf{C}_\epsilon(G)\|$	Non-min'l Cuts	Calls to MF	Run Time (sec.)
$\epsilon = 0.05$								
GGF5×5	27	90	5	5	4	0	22	0.1
GGF10×10	102	380	10	10	9	0	92	0.1
GGF15×15	227	870	15	15	14	0	212	0.1
GGF20×20	402	1560	20	21	703	0	7906	1.5
GGF25×25	627	2450	25	26	1128	0	15506	3.9
GGF30×30	902	3540	30	31	1653	0	26856	12.0
$\epsilon = 0.10$								
GGF5×5	27	90	5	5	22	0	22	0.1
GGF10×10	102	380	10	11	956	0	956	0.1
GGF15×15	227	870	15	16	3306	0	3306	0.5
GGF20×20	402	1560	20	22	113090	0	113090	20.1
GGF25×25	627	2450	25	27	274550	0	274550	74.4
GGF30×30	902	3540	30	33	8911698	378	8911698	∼4.5k
$\epsilon = 0.15$								
GGF5×5	27	90	5	5	22	0	22	0.1
GGF10×10	102	380	10	11	956	0	956	0.2
GGF15×15	227	870	15	17	35905	0	35905	4.9
GGF20×20	402	1560	20	23	1202033	153	1202033	215.3
GGF25×25	627	2450	25	28	3621978	253	3621978	973.3
GGF30×30	902	3540	30	34	13465371	21843	113463496	∼46k

Table 2.8. Computational results for near-min cut enumeration (ANMCP) on weighted DBLCYC-I problems with $n = 500$ (and $m = 2000$). The number of non-minimal cuts encountered at non-terminal nodes increases substantially when ϵ becomes sufficiently large.

ϵ	w_0	$\bar{w}_\epsilon$	$\|\mathbf{C}_0(G)\|$	Non-min'l Cuts	Calls to MF	Run Time (sec.)
0.00	1000	1000	2	0	10	0.1
0.10	1000	1100	499	0	1976	0.5
1.00	1000	2000	511	8	2032	0.6
1.25	1000	2250	2479	237411	957178	207.8
1.50	1000	2500	2479	237411	957178	208.7
1.75	1000	2750	2479	237411	957178	207.4
2.00	1000	3000	2509	238041	959683	213.4

slow computations when the threshold parameter ϵ becomes large, at least for certain graph topologies. For dense acyclic graphs, the behavior of Algorithm B verifies Corollary 2: No non-minimal cuts are encountered at non-terminal nodes. However, for the double-cycle graphs DBLCYC-I, the number of non-minimal cuts generated can outnumber the minimal cuts by a large margin, at least when ϵ becomes large.

4. Conclusions and Recommendations

In this paper, we have developed an algorithm for ANMCP, defined as the problem of enumerating *all near-minimum-weight, minimal s-t cuts* C_ϵ in a directed graph $G = (V, E)$ with positive integer edge weights $w_e \ \forall \ e \in E$. The users specifies a value $\epsilon \geq 0$, and the algorithm finds all minimal *s-t* cuts C_ϵ such that $w(C_\epsilon) \leq (1 + \epsilon)w(C_0)$, where $w(C)$ denotes the weight of cut C, and C_0 is a min-weight cut. The algorithm first finds a min-weight cut C_0 in the input graph via a maximum-flow algorithm, and then recursively partitions the space of near-min cuts. Given a cut C, this partitioning is carried out by forcing inclusion and exclusion of edges from subsequent cuts. An edge (u, v) is *quasi-excluded* by simply setting its weight to infinity and *quasi-included* by implicitly introducing two infinite-weight edges in G, one extending from s to u and the other from v to t. The algorithm solves a max-flow min-cut problem for each modified graph that is obtained in the enumeration tree.

We have implemented our algorithm using the following enhancements to improve computational speed: (a) The algorithm solves a complete max-flow problem at the root node of enumeration tree but solves only "incremental" max-flow problems at the all other nodes (the max flow at a parent node is feasible for all child nodes and thus provides an advanced start for maximizing flows at those child nodes), and (b) quasi-inclusion of an edge (u, v) is simulated by treating u as an additional source and v as an additional sink, and (c) the algorithm backtracks directly from the max-flow subroutine, without identifying a locally minimum cut, if a feasible flow is found that exceeds the backtrack threshold. (That flow is a lower bound on the weight of the min cut.)

Unfortunately, the quasi-inclusion technique can lead to the enumeration of non-minimal cuts at non-terminal nodes of the enumeration tree. Non-minimal cuts are easily identified (and ignored), but they can increase the computational workload and stop us from deriving a polynomial-time bound for the worst-case complexity of the general algorithm: The algorithm cannot always backtrack when it finds a non-minimal cut. We do obtain, however, a polynomial bound of

$O(f(n,m)+nm|\mathbf{C}_\epsilon(G)|)$ when $\min_{e\in E} w_e > w(C_0)\epsilon$; here $\mathbf{C}_\epsilon(G)$ denotes the set of near-min cuts and $O(f(n,m))$ is the worst-case complexity of the max-flow algorithm being used. Thus, the algorithm has polynomial complexity, per cut enumerated, for the important special case of ANMCP when $\epsilon = 0$, i.e., AMCP: Enumerate all min-weight s-t cuts in G. We also determine the polynomial bound of $O(nf(n,m)|\mathbf{C}_\epsilon(G)|)$ for certain graph topologies such as complete graphs and complete acyclic graphs.

Computational results for $\epsilon > 0$ show that Algorithm B has good empirical efficiency as long as ϵ is not too large. Unfortunately, large ϵ can lead to the identification of many non-minimal cuts where the algorithm cannot immediately backtrack. Thus, many "unproductive" non-terminal nodes can be encountered in the enumeration tree, and it is only these nodes that stop us from proving polynomial complexity.

To improve the algorithm, one might try to improve the quasi-inclusion technique or develop a completely different technique for edge inclusion. For instance, we have not tried simply setting to 0 the capacity of an arc to be included. If "true edge inclusion" (as opposed to quasi-inclusion) can be efficiently implemented, this should yield a provably polynomial-time algorithm for near-min cut enumeration. However, it can be proven that the problem of finding a min cut that includes a specific set of edges is actually NP-complete.

If the current quasi-inclusion technique is retained, another approach might be used to avoid enumerating non-minimal cuts. In particular, edges that cannot occur in any minimal cut given those that are already included might be identified and marked as "forbidden for inclusion." These edges would be excluded, as usual, by setting their weights to infinity. An edge (u,v) can be forbidden from inclusion if (a) every s-u or v-t path contains at least one included edge, or (b) some included edge (u',v') must contain (u,v) in every s-u' path or in every v'-t path. This list is not all-inclusive, however.

Another practical improvement might result from this: The algorithm can backtrack whenever the set of quasi-included edges forms a cut in the original graph.

Computation times on some large graphs would be improved by solving the initial maximum-flow problem using a more efficient algorithm, e.g., Goldberg and Rao (1998). It may also be possible to show that the worst-case complexity of the algorithm is actually better than reported by amortizing the work involved in augmenting flows over the course of running the algorithm.

We have shown that Algorithm B will not enumerate any non-minimal cuts if every vertex $v \in V\backslash\{s,t\}$ has incident edges (s,v) and (v,t). It

would be interesting to determine if the algorithm will enumerate only minimal cuts for other graph topologies, too. For instance, using the dual of a planar graph and shortest-path techniques, it is possible to enumerate near-min cuts in an undirected s-t planar graph in polynomial time per cut. Thus, it is natural to wonder if Algorithm B can too.

Acknowledgments

The authors thank Matthew Carlyle, David Morton, Alexandra Newman and Craig Rasmussen for their helpful suggestions regarding this paper. The authors also thank the Office of Naval Research and the Air Force Office of Scientific Research for their support of this research.

References

Abel, V., and Bicker, R., (1982), "Determination of All Minimal Cut-Sets between a Vertex Pair in an Undirected Graph," *IEEE Transactions on Reliability*, Vol. R-31, pp. 167-171.

Ahmad, S.H., (1990), "Enumeration of Minimal Cutsets of an Undirected Graph," *Microelectronics Reliability*, Vol. 30, pp. 23-26.

Avis, D., and Fukuda, K., (1996) "Reverse Search for Enumeration," *Discrete Applied Mathematics*, Vol. 65, pp. 21-46.

Boyle, M.R., (1998), "Partial-Enumeration For Planar Network Interdiction Problems," Master's Thesis, Operations Research Department, Naval Postgraduate School, Monterey, California, March.

Bussieck, M.R., and Lübbecke, M.E., (1998), "The Vertex Set of a 0/1-Polytope is Strongly P-enumerable," *Computational Geometry*, Vol. 11, pp. 103-109.

Carlyle, W.M., and Wood, R.K., (2002) "Efficient Enumeration of Minimal and Non-Minimal s-t Cuts," working paper, Operations Research Department, Naval Postgraduate School, Monterey, CA.

Colbourn, C.J., 1987, *The Combinatorics of Network Reliability*, Oxford University Press.

Curet, N.D., DeVinney, J., Gaston, M.E. 2002, "An Efficient Network Flow Code for Finding All Minimum Cost s-t Cutsets," *Computers and Operations Research*, Vol. 29, pp. 205-219.

DIMACS, (1991), *The First DIMACS International Algorithm Implementation Challenge,* Rutgers University, New Brunswick, New Jersey. (Available via anonymous ftp from dimacs.rutgers.edu.)

Edmonds, J. and Karp, R.M., (1972), "Theoretical Improvements in Algorithm Efficiency for Network Flow Problems," *Journal of the ACM*, Vol. 19, pp. 248-264.

Fard, N.S., and Lee, T.H., (1999), "Cutset Enumeration of Network Systems with Link and Node Failures," *Reliability Engineering and System Safety*, Vol. 65, pp. 141-146.

Gibbons, M., (2000), "Enumerating Near-Minimum Cuts in a Network," Master's Thesis, Operations Research Department, Naval Postgraduate School, Monterey, California, June.

Goldberg, A.V., and Rao, S., (1998), "Beyond the Flow Decomposition Barrier," *Journal of the ACM*, Vol. 45, pp. 783-797.

Goldberg, A.V., and Tarjan, R.E., (1988), "A New Approach to the Maximum Flow Problem," *Journal of the ACM*, Vol. 35, pp. 921-940.

Gusfield, D., and Naor, D., (1993), "Extracting Maximal Information About Sets of Minimum Cuts," *Algorithmica*, Vol. 10, pp. 64-89.

Kanevsky, A., (1993), "Finding All Minimum-Size Separating Vertex Sets in a Graph," *Networks*, Vol. 23, pp. 533-541.

Karger, D.R., (2000), "Minimum Cuts in Near-Linear Time," *Journal of the ACM*, Vol. 47, pp. 46-76.

Karger, D.R., and Stein, C., (1996), "A New Approach to the Minimum Cut Problem," *Journal of the ACM*, Vol. 43, pp. 601-640.

Lawler, E.L., Lenstra, J.K., Rinooy Kan, A.H.G., and Shmoys, D.B., (1985), *The Traveling Salesman Problem*, John Wiley & Sons, Chichester, England.

Levine, M.S., (1997), "Experimental Study of Minimum Cut Algorithms," Master's Thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, May. (http://theory.lcs.mit.edu/∼mslevine)

Nagamochi, H., Ono, T., and Ibaraki, T., (1994), "Implementing an Efficient Minimum Capacity Cut Algorithm," *Mathematical Programming*, Vol. 67, pp. 297-324.

Nagamochi, H., Nishimura, K., and Ibaraki, T., (1997), "Computing All Small Cuts in an Undirected Network," *SIAM Journal on Discrete Mathematics*, Vol. 10, pp. 469-481.

Nahman, J.M., (1997), "Enumeration of Minimal Cuts of Modified Networks," *Microelectronics Reliability*, Vol. 37, pp. 483-485.

Patvardhan, C., Prasad, V.C. and Pyara, V.P., (1995), "Vertex Cutsets of Undirected Graphs," *IEEE Transactions on Reliability*, Vol. 44, pp. 347-353.

Picard, J.C., and Queyranne, M., (1980), "On The Structure of All Minimum Cuts in a Network and Applications," *Mathematical Programming Study*, Vol. 13, pp. 8-16.

Prasad, V.C., Sankar, V., and Rao, P., (1992), "Generation of Vertex and Edge Cutsets," *Microelectronics Reliability*, Vol. 32, pp. 1291-1310.

Provan, J.S., and Ball, M.O., (1983), "Calculating Bounds on Reachability and Connectedness in Stochastic Networks," *Networks*, Vol. 13, pp. 253-278.

Provan, J.S., and Shier, D.R., (1996), "A Paradigm for Listing (s,t)-Cuts in Graphs," *Algorithmica*, Vol. 15, pp. 351-372.

Ramanathan, A., and Colbourn, C.J., (1987), "Counting Almost Minimum Cutsets with Reliability Applications," *Mathematical Programming*, Vol. 39, pp. 253-261.

Shier, D.R., and Whited, D.E., (1986), "Iterative Algorithms for Generating Minimal Cutsets in Directed Graphs," *Networks*, Vol. 16, pp. 133-147.

Sung, C.S., and Yoo, B.K., (1992), "Simple Enumeration of Minimal Cutsets Separating 2 Vertices in a Class of Undirected Planar Graphs," *IEEE Transactions on Reliability*, Vol. 41, pp. 63-71.

Sun Microsystems Inc., (1998), Java Platform Version 1.2.2.

Tsukiyama, S., Shirakawa, I., Ozaki, H., and Ariyoshi, H., 1980, "An Algorithm to Enumerate All Cutsets of a Graph in Linear Time per Cutset," *Journal of the ACM*, Vol. 27, pp. 619-632.

Vazirani, V.V., and Yannakakis, M., (1992), "Suboptimal Cuts: Their Enumeration, Weight and Number," *Automata, Languages and Programming. 19th International Colloquium Proceedings*, Vol. 623 of Lecture Notes in Computer Science, Springer-Verlag, pp. 366-377.

Wood, R.K., (1993), "Deterministic Network Interdiction," *Mathematical and Computer Modeling*, Vol. 17, pp. 1-18.