Computational Feasibility of Increasing the Visibility of Vertices in Covert Networks

Yaniv Ovadia

Susan E. Martonosi, Advisor

Nicholas J. Pippenger, Reader

May, 2010

HARVEY MUDD COLLEGE

Department of Mathematics

Copyright © 2010 Yaniv Ovadia.

The author grants Harvey Mudd College the nonexclusive right to make this work available for noncommercial, educational purposes, provided that this copyright statement appears on the reproduced materials and notice is given that the copying is by permission of the author. To disseminate otherwise or to republish requires written permission from the author.

Abstract

Disrupting terrorist and other covert networks requires identifying and capturing key leaders. Previous research by Martonosi et al. (2009) defines a load metric on vertices of a covert network representing the amount of communication in which a vertex is expected to participate. They suggest that the visibility of a target vertex can be increased by removing other, more accessible members of the network. This report evaluates the feasibility of efficiently calculating the optimal subset of vertices to remove.

We begin by proving that the general problem of identifying the optimally load maximizing vertex set removal is NP-complete. We then consider the feasibility of more quickly computing the load maximizing single vertex removal by designing an efficient algorithm for recomputing Gomory-Hu trees. This leads to a result regarding the uniqueness of Gomory-Hu trees with implications towards the feasibility of one approach for Gomory-Hu tree reconstruction. Finally, we propose a warm start algorithm which performs this reconstruction, and analyze its runtime experimentally.

Contents

Abstract **iii**

1 Introduction **1**
1.1 Modeling of Covert Networks 1
1.2 Background Literature 2
1.3 Overview 2

2 The Load Maximizing Subset Problem is NP-Complete **5**
2.1 Problem Definition 5
2.2 Proof of NP-Completeness 6
2.3 Summary 11

3 Gomory-Hu Trees **13**
3.1 Gomory-Hu Tree Diversity 14

4 A Warm-Start Algorithm for Gomory-Hu Tree Reconstruction **17**
4.1 Algorithm Overview 17
4.2 Subroutine Descriptions 18
4.3 Experimental Performance 24

5 Future Work and Conclusion **27**
5.1 Alternative Gomory-Hu Tree Reconstruction Algorithm . . . 27
5.2 Additional Future Work 31
5.3 Conclusion 32

A The Gusfield Algorithm for Gomory-Hu Tree Construction **33**

Bibliography **35**

List of Figures

1.1 An example load calculation. 2

2.1 A multi-graph represented as a simple graph in a load maximizing subset problem instance. 6
2.2 An example of our reduction from SLMSP to ALMSP. 7
2.3 An instance of SLMSP generated by our 3SAT reduction. . . 10

3.1 An example graph and its corresponding Gomory-Hu tree. . 13
3.2 Example of a graph with multiple Gomory-Hu trees. 15

4.1 An example input to the join-paths subroutine. 21
4.2 An example execution of the join-paths subroutine. 23
4.3 Experimental performance of the warm-start algorithm. . . . 26

5.1 A depiction of the setup for the second case in Prop. 5.1. . . 28

Chapter 1

Introduction

Experts believe that clandestine organizations such as terrorist networks are heavily dependent upon a well hidden leadership thus providing specific targets for disrupting the network. We also assume that the visibility of an individual in the network correlates with the quantity of information passing through that individual, and that this quantity of information is related to the individual's position in the network's structure. This suggests that modifications to the network's structure could be engineered to force more information to pass through the target individual, thereby increasing that member's visibility. Specifically, Martonosi et al. (2009) proposed that networks could be restructured by removing more easily located members of the network.

We study the feasibility of quickly calculating which individuals to remove in order to optimally increase the target's visibility.

1.1 Modeling of Covert Networks

We represent the clandestine network as a simple undirected graph with vertices representing organization members, and we quantify the notion of visibility due to communication passing through a vertex according to the load metric proposed by Martonosi et al. (2009). Given a target vertex k, let $f_{ij}(G)$ be the value of a maximum flow (or equivalently, minimum cut) between vertices i and j in graph G, and let $G' = G \setminus \{v\}$ be the subgraph induced by $V \setminus \{v\}$. We define the *load* of k in G to be

$$L(k, G) = \sum_{i<j \in V \setminus \{k\}} f_{i,j}(G) - f_{i,j}(G \setminus \{k\})$$

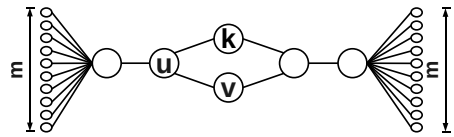


Figure 1.1: In this example graph, the load on k is only 3 because the flow between most pairs of vertices does not depend on k. If v is removed, the load increases to $(m+2)^2$; however, if u is removed, the load on k reduces to zero.

Since G is a simple graph, $f_{i,j}$ is equivalent to the number of edge disjoint (i,j)-paths, so we may interpret the load on k to be the sum over all possible pairs i, j of the number of (i,j)-paths which must pass through k.

Our goal of maximizing the visibility of k is therefore represented as maximizing k's load. Since the effects of vertex deletion on load can be complicated, determining an optimal subset of vertices to remove is non-trivial. Figure 1.1 depicts an example calculation of load, and demonstrates the effect of two possible vertex deletions.

1.2 Background Literature

Gomory and Hu (1961) described a tree structure (referred to today as a Gomory-Hu tree) which encodes a minimum cut between each pair of vertices in an undirected graph, and provided an algorithm for computing this structure using only $n-1$ maximum-flow computations. Gusfield (1990) later offered a conceptually simpler algorithm. Goldberg and Tsioutsiouliklis (2001) experimentally compared variations of the Gomory-Hu and Gusfield algorithms.

Bhalgat et al. (2007) described a randomized algorithm for constructing Gomory-Hu trees of unweighted graphs in an expected time of $\widetilde{O}(mn)$. This is the first algorithm which constructs Gomory-Hu trees without depending on a minimum cut subroutine.

1.3 Overview

We begin by proving that the general problem of identifying the optimal subset removal is NP-complete. This result indicates that a computation-

ally feasible solution is unlikely to exist.

A simpler variation of the problem is to identify the single vertex whose deletion most significantly increases the load on k. A solution to this problem may then be applied as a subroutine in a heuristic or approximation algorithm to solve the more general variation. The optimal single vertex removal can be identified in polynomial time by a simple brute force algorithm which calculates $L(k, G \setminus \{v\})$ for each $v \in V \setminus \{k\}$ and returns the best choice of v. However, since any heuristic or approximation algorithm which utilizes a solution to the single vertex problem will make many calls to this subroutine, we wish to develop a more efficient solution to the single vertex problem.

Specifically, since the brute force algorithm requires many all-pairs minimum cut computations (or equivalently, Gomory-Hu tree constructions), Martonosi et al. (2009) observed that the results of an all-pairs minimum cut calculation in G may be applied towards a more efficient calculation of all-pairs minimum cuts in $G \setminus \{v\}$.

Many Gomory-Hu trees may exist for a single graph. In Chapter 3, we demonstrate that this diversity is due not only to vertex permutation, but also the choice of minimum cuts when multiple such cuts exist. This suggests that constructing the specific tree which changes least following a vertex deletion is likely to be difficult.

Finally, in Chapter 4, we define a warm-start algorithm for recalculating Gomory-Hu trees, and analyze its performance experimentally.

Chapter 2

The Load Maximizing Subset Problem is NP-Complete

Perhaps the most applicable form of load maximizing vertex removal entails identifying the optimal subset of vertices to remove subject to the constraint that the vertices we remove are among the accessible vertices. We prove that this general variation of the problem is NP-complete, and therefore a polynomial time solution is unlikely to exist.

2.1 Problem Definition

Recall that the *load* on a vertex, k, as defined by Martonosi et al. (2009) is

$$L(k, G) = \sum_{s,t \in V \setminus k; s \neq t} f_{s,t}(G) - f_{s,t}(G \setminus k). \tag{2.1}$$

In this proof, we will also refer to the load with respect to (s, t) defined as

$$L_{s,t}(k, G) = f_{s,t}(G) - f_{s,t}(G \setminus k). \tag{2.2}$$

An instance of the All-Pairs Load Maximizing Subset Problem (ALMSP) is described by a 4-tuple (G, k, S, N) where G is a multi-graph representing the covert network (note that the multi-graph is a more general structure than a simple graph), $k \in V$ is the target vertex, $S \subseteq V$ is the set of vertices for which deletion is feasible (ie: the set of accessible individuals in the network), and $N \in \mathbb{N}$ is the minimum desired load on k. A solution is represented by a set of vertices $R \subseteq S$ such that

$$L(k, G \setminus R) \geq N.$$

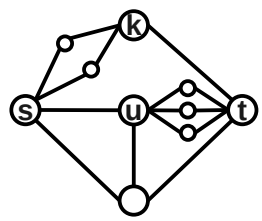


Figure 2.1: This simple graph (where the smaller vertices are stipulated not to be in R) represents the multi-graph where the small vertices are not present, and the pairs (s, k), and (u, t) are connected by 2 and 3 parallel edges respectively.

An instance of the Single-Pair Load Maximizing Subset Problem (SLMSP) is described by a 5-tuple $(G, k, S, N, \{s, t\})$, and a solution is a set of vertices $R \subseteq S$ such that

$$L_{s,t}(k, G \setminus R) \geq N.$$

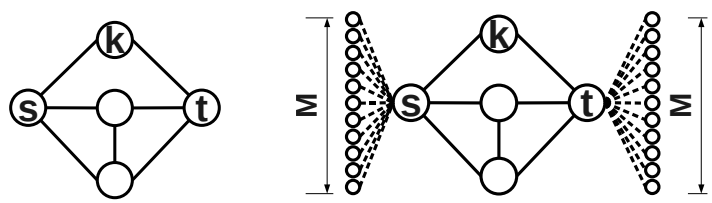
Although multi-graphs do not accurately model covert networks, we justify their use by noting that a repeated edge $\{u, v\}$ with r copies of itself in E could be represented by edges $\{u, w_i\}$ and $\{w_i, v\}$, where $w_i \notin S$ and $1 \leq i \leq r$. Figure 2.1 depicts an example of a simple graph encoding a multi-graph. This implies that for any form of LMSP, allowing or disallowing parallel edges does not affect its complexity by more than a polynomial factor. Henceforth, we may choose to either allow or disallow such edges in our reductions.

2.2 Proof of NP-Completeness

We begin by proving that there exists a polynomial time reduction of SLMSP to ALMSP. We will then prove that the SLMSP is NP-complete, implying that ALMSP is NP-complete as well.

2.2.1 Reduction of SLMSP to ALMSP

Given an instance of SLMSP, $P = (G, k, S, N, \{s, t\})$, we construct an instance of ALMSP, $P' = (G', k, S, N')$ such that a solution exists for P if and only if a solution exists for P'. For this reduction, we do not permit parallel edges.



(a) (b)

Figure 2.2: Figure 2.2a depicts an example of a graph G from an instance of ALMSP. Figure 2.2b depicts the graph G' constructed by our SLMSP to ALMSP reduction. The dashed lines represent $|V|$ parallel edges.

Choose M such that

$$M^2 > 2(|V|^3 - 2M|V|^2). \tag{2.3}$$

and define $N' = NM^2$. We construct $G' = (V', E')$ where

$$V' = V \cup \{\sigma_i : 1 \leq i \leq M\} \cup \{\tau_i : 1 \leq i \leq M\}$$

$$E' = E \cup \bigcup_{j=1}^{|V|} \{\{\sigma_i, s\} : 1 \leq i \leq M\} \cup \bigcup_{j=1}^{|V|} \{\{\tau_i, t\} : 1 \leq i \leq M\}.$$

The new graph is identical to G except for added vertices which are adjacent with $|V|$ parallel edges to either s or t. Figure 2.2 depicts an example of this reduction.

($\Rightarrow$) Let $R \subseteq S$ be a solution to P. We show that the same set is also a solution to P'. Observe that for all i, j, $f_{\sigma_i,\tau_j}(G') = f_{s,t}(G)$. It follows that we can split the load on k in G' into the contribution between vertices in G and the contribution between vertices of form σ_i and τ_j.

$$\begin{aligned} L(k, G' \setminus R) = & L(k, G \setminus R) \\ & + M \sum_{v \in V \setminus \{s,t,k\}} L_{v,s}(k, G \setminus R) \\ & + M \sum_{v \in V \setminus \{s,t,k\}} L_{v,t}(k, G \setminus R) \\ & + M^2 L_{s,t}(k, G \setminus R) \end{aligned} \tag{2.4}$$

Since R is a solution to P, we know that $L_{s,t}(k, G \setminus R) \geq N$. We may then use Equation 2.4 to state

$$\begin{aligned} L(k, G' \setminus R) &\geq L(k, G \setminus R) + M^2 L_{s,t}(k, G \setminus R) \\ &\geq L(k, G \setminus R) + M^2 N \\ &= N' \end{aligned}$$

($\Leftarrow$) Given a solution $R \subseteq S$ to P', we may write $L(k, G' \setminus R) \geq N'$. We use Equation 2.4 and the fact that $f_{u,v} < |V|$ for any u, v to state

$$\begin{aligned} L_{s,t}(k, G \setminus R) &\geq \frac{1}{M^2}[L(k, G' \setminus R) - L(k, G \setminus R) - 2M|V|^2] \\ &\geq \frac{1}{M^2}[N' - |V|^3 - 2M|V|^2] \\ &= N - \frac{|V|^3 - 2M|V|^2}{M^2} \\ &\geq N - \frac{1}{2}. \end{aligned}$$

Since all our values are integers, it follows that $L_{s,t}(k, G \setminus R) \geq N$.

2.2.2 SLMSP is NP-Complete

Calculating the load on k in a subgraph of G can be done in polynomial time, implying that SLMSP is nondeterministically polynomial. In order to prove our problem NP-complete, we need only show that there exists a polynomial time reduction of 3SAT to SLMSP.

The 3-Satisfiability Problem (3SAT) entails determining the existence of **true**/**false** assignments for the variables of a Boolean formula in 3-conjunctive normal form (3CNF) which cause the formula to evaluate to **true**. A Boolean formula in 3CNF consists of the conjunction of clauses of size 3, where a clause is a disjunction of literals. For example, a formula in 3CNF with 3 variables may be

$$(x_i \textbf{ or } \overline{x_j} \textbf{ or } x_k) \textbf{ and } (\overline{x_i} \textbf{ or } x_j \textbf{ or } x_k)$$

(where $\overline{x_j}$ indicates the negation of variable x_j), and one solution to this instance is the valuation ν where $\nu(x_i) = \nu(x_j) = \textbf{true}$ and $\nu(x_k) = \textbf{false}$.

Given a 3SAT instance with variables $x_1, \ldots, x_n$, and clauses $C_1, \ldots, C_m$ (note that for this proof, n and m do not refer to the number of vertices and

edges of a graph), we construct an instance $(G, k, S, N, \{s, t\})$ of SLMSP as follows:

Let s be a vertex in G with two edges to each vertex $\{c_i : 1 \leq i \leq n\}$. For each i, c_i is also incident to two edges leading to each of a_i and b_i. Both a_i and b_i are adjacent to d_i via a single edge, and both are adjacent to t also with a single edge. d_i is adjacent to the target vertex k. The target vertex is incident to $m + n$ edges between it and t. The features described so far represent the 3SAT variables. We will choose $x_i = \textbf{true}$ if $b_i \in R$, and $x_i = \textbf{false}$ if $a_i \in R$.

We then proceed to encode the m clauses. Let C_j $(1 \leq j \leq m)$ be a clause containing the negated or unnegated forms of three variables. Let s be adjacent to u_j with a single edge. For each variable x_{i_ℓ} represented in C_j, if $x_{i_\ell} \in C_j$, we create the edges $\{u_j, a_{i_\ell}\}$ and $\{a_{i_\ell}, v_j\}$. If $\overline{x_{i_\ell}} \in C_j$, we instead construct $\{u_j, b_{i_\ell}\}$ and $\{b_{i_\ell}, v_j\}$. Figure 2.3 depicts the SLMSP instance constructed to encode a simple 3SAT instance.

Finally, we select $N = m + n$, and

$$S = \{a_i : 1 \leq i \leq n\} \cup \{b_i : 1 \leq i \leq n\}.$$

We claim that for any instance of SLMSP which represents a 3SAT instance, a solution exists to the SLMSP instance if and only if the 3SAT instance is satisfiable as well.

($\Leftarrow$) Given a satisfying valuation of the 3SAT variables $\nu : \{x_i : 1 \leq i \leq n\} \to \{\textbf{true}, \textbf{false}\}$, we construct a SLMSP, $(G, k, S, N, \{s, t\})$ instance as described earlier. It is easy to show that if we choose

$$R = \{a_i : \nu(x_i) = \textbf{true}\} \cup \{b_i : \nu(x_i) = \textbf{false}\}$$

then

$$\begin{aligned} L_{s,t}(k, G \setminus R) &= f_{s,t}(G \setminus R) - f_{s,t}(G \setminus (R \cup \{k\}) \\ &= (m + 2n) - n = m + n \end{aligned}$$

Since the $L_{s,t}(k, G \setminus R) \geq N$, the problem instance is solved by R.

($\Rightarrow$) Given that an SLMSP instance, P, which encodes a 3SAT instance, has some solution R, we wish to show that a solution exists to the 3SAT instance. In particular, we will show that the valuation

$$\nu(x_i) = \begin{cases} \textbf{true} & : \{a_i, b_i\} \cap R = \{b_i\} \\ \textbf{false} & : \{a_i, b_i\} \cap R = \{a_i\} \end{cases} \tag{2.5}$$

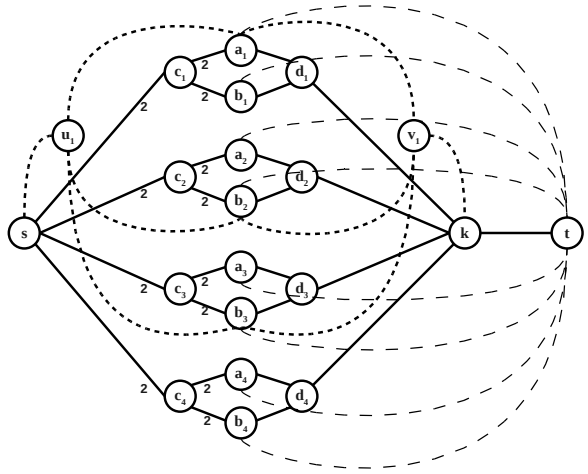


Figure 2.3: The SLMSP instance constructed from the 3SAT instance with four variables and a single clause $\{x_1, \overline{x_2}, \overline{x_3}\}$.

will satisfy the 3SAT instance.

Notice that

$$L_{s,t}(k, G \setminus R) =$$

$$\sum_{i=1}^{n} \min\{2, f_{c_i,t}(G \setminus (R \cup \{s\}))\} - \min\{2, f_{c_i,t}(G \setminus (R \cup \{s,k\}))\} \tag{2.6}$$

$$+ \sum_{j=1}^{m} \min\{1, f_{u_j,t}(G \setminus (R \cup \{s\}))\} - \min\{1, f_{u_j,t}(G \setminus (R \cup \{s,k\}))\}. \tag{2.7}$$

We perform a case analysis to show that the terms in (2.6) each have value at most one. Consider the ith term in the series. If both $\{a_i, b_i\} \subseteq R$, then c_i is isolated, and the term is zero, if $\{a_i, b_i\} \cap R = \emptyset$, then the difference is again zero, and if exactly one of $\{a_i, b_i\}$ is in R, then the result is one. Furthermore, it is clear that all terms in (2.7) are bounded above by one. Thus, since R is a valid solution, and the equation contains exactly $m + n$ terms, we conclude that each term is equal to one.

As demonstrated by our case analysis, this implies that for every pair $\{a_i, b_i\}$, exactly one of the two is selected for removal. Thus our description of ν in (2.5) is well defined. Finally, since every term in (2.7) has value one, the valuation leaves no clause without a satisfying variable. It follows that ν is a satisfying valuation for the 3SAT instance, and that 3SAT $\leq_P$ SLMSP.

2.3 Summary

By proving this problem NP-complete, we have shown that an efficient solution is unlikely to exist. Thus in order to continue pursuing load maximizing vertex deletions, we must resign to posing a more restricted problem, or seeking algorithms which do not guarantee optimality. One such simplification entails identifying only the single vertex whose deletion maximizes the load on k. Furthermore, this simpler variant of the problem may also serve as a subroutine to a heuristic or approximation algorithm for the general problem we have shown to be NP-complete.

By only considering the optimal single vertex removal, we observe that a polynomial time brute force algorithm exists for the problem (simply compute $L(k, G \setminus \{v\})$ for each $v \in V \setminus \{k\}$, and return the best choice of v). The ensuing chapters explore the feasibility of improving the efficiency of this simple strategy by preserving information from the construction of G's Gomory-Hu tree in order to provide a warm start to the calculation of of a Gomory-Hu tree for $G \setminus \{v\}$ ($v \in V$).

Chapter 3

Gomory-Hu Trees

A Gomory-Hu tree for an undirected graph G is a tree T which encodes the minimum cut value between every pair of vertices as well as a partitioning representing a cut of this value. For any pair of vertices $u, v \in G$, the value of a minimum (u, v)-cut is equal to the weight of the lightest edge on the unique (u, v)-path in T. Furthermore, we can obtain the partition for a minimum cut of this value from the connected components that result when this lightest edge is removed from T. Figure 3.1 depicts an example graph and its corresponding Gomory-Hu tree.

A Gomory-Hu tree is particularly useful because it can be computed with only $n - 1$ calls to a minimum cut subroutine as opposed to the $\binom{n}{2}$ calls required by a naïve algorithm. Gomory and Hu (1961) defined the first algorithm for constructing this structure, and Gusfield (1990) later produced a conceptually simpler algorithm. The Gusfield algorithm is described in Appendix A.

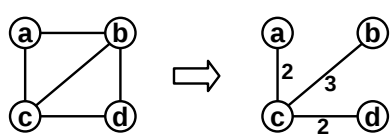


Figure 3.1: An example graph and its corresponding Gomory-Hu tree.

3.1 Gomory-Hu Tree Diversity

It is easy to notice that the Gusfield algorithm will produce different trees under different vertex permutations (consider K_n). The following result demonstrates that diversity of valid Gomory-Hu trees for a graph G is not a result of only the variety of vertex numberings, but also the potential existence of multiple minimum cuts between vertices.

Proposition 3.1. *Running the Gusfield cut-tree algorithm with every possible ordering of vertices does not necessarily yield every valid Gomory-Hu tree for the graph.*

Proof. Consider the algorithm's behavior on the complete graph $G = K_n$ using the following strategies for choosing a particular minimum-cut between vertices s and t.

- Choose the minimum cut (X, Y) where $s \in X$ and $|X|$ is as small as possible. We call this an *unbalanced* minimum cut.
- Choose a minimum cut (X, Y) where $s \in X$ and $|X| - |Y|$ is as small as possible. We will refer to this as a *balanced* minimum cut.

First we consider the Gusfield algorithm's behavior on K_n using only unbalanced minimum cuts. Since the set X derived from the minimum cut is always a singleton set containing only s, the vector representing the tree's structure is never edited. It follows that the resulting tree is a star graph centered about vertex 1 with all edges of weight $n - 1$.

Now consider the Gusfield algorithm's behavior on K_n using a single balanced cut at the first iteration, and unbalanced cuts for all subsequent minimum cut calculations. After the first iteration, every vertex in X except for s will have its edge in the vector `tree` redirected towards s. For all subsequent iterations, X is a singleton set, so the tree's structure is preserved. This ultimately yields a graph consisting of two star subgraphs and an edge joining the stars' centers.

These graphs are clearly not isomorphic, so it follows that using alternative vertex permutations does not account for the diversity of a graph's valid Gomory-Hu trees.

Figure 3.2 depicts two possible graphs produced by this construction when $n = 4$. □

One potential strategy for efficiently rebuilding a Gomory-Hu tree in response to a vertex deletion is to explicitly build the original Gomory-Hu tree such that for some subset of vertices, deletion of those vertices,

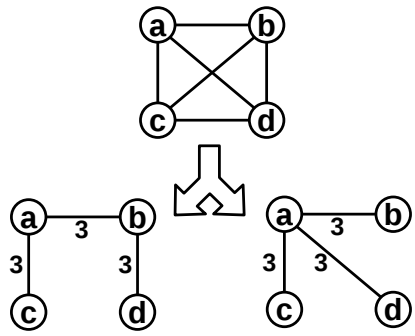


Figure 3.2: An example of a graph with two distinct Gomory-Hu trees. The difference is due to choices in minimum cuts as opposed to vertex permutation.

causes the Gomory-Hu tree to change as little as possible. For example, if we consider a Gomory-Hu tree for K_n structured as a star graph centered about vertex r, deletion of any vertex except r has little effect on the tree's topology.

Our result suggests that constructing this specific tree which changes least after a vertex deletion is likely to be difficult because not only will a proper permutation have to be identified, but appropriate choices for minimum cut will also have to be made.

Chapter 4

A Warm-Start Algorithm for Gomory-Hu Tree Reconstruction

In order to facilitate the calculation of the optimal single vertex removal, we now describe an algorithm for reconstructing the Gomory-Hu tree of a simple graph G in response to vertex deletion. We begin with a brief overview of the proposed algorithm followed by a detailed definition of the necessary subroutines.

4.1 Algorithm Overview

The process begins with the initial construction of a Gomory-Hu tree T for graph G. Since the reconstruction process will depend on more than the minimum-cuts encoded in T, we define a slightly modified form of Gusfield's algorithm which, in addition to constructing a Gomory-Hu tree, also provides a (u, v)-flow assignment for every edge $\{u, v\}$ in G. We refer to this algorithm as Gusfield-init.

In order to obtain a Gomory-Hu tree T' for $G \setminus \{v\}$, we employ another variation of Gusfield's algorithm which we will refer to as Gusfield-update. This algorithm behaves nearly identically to Gusfield's original algorithm, except every call to a maximum flow subroutine on vertices u and v is preceded by a process which uses the flow assignments obtained from Gusfield-init to build a (not necessarily maximum) (u, v)-flow assignment in G'. This flow assignment is then used as a starting point for the maximum flow subroutine.

4.2 Subroutine Descriptions

We describe three major subroutines. The first subroutine, Gusfield-init, constructs the initial Gomory-Hu tree from scratch. We then describe Gusfield-update which uses flow assignments associated with an existing Gomory-Hu tree to warm start the construction for a new Gomory-Hu tree (and collection of flow assignments) following vertex deletion. Finally we describe the join-paths subroutine which uses a set of edge disjoint (a, b)-paths and a set of edge disjoint (b, c)-paths to construct a set of (a, c)-paths. This subroutine is used by both the Gusfield-init and Gusfield-update subroutines.

4.2.1 Gusfield-init

Given a graph G, this subroutine returns a Gomory-Hu tree T for G as well as a (u, v)-flow assignment for every edge $\{u, v\}$ in G, stored as a list of edge disjoint (u, v)-paths. The following pseudocode will yield arrays `tree` and `weights` which encode T's topology and edge weights, and `flow-paths` which stores the computed flow assignments corresponding with the edges in T.

This subroutine returns, in addition to a tree structure with weights, a vector `flow-paths` of length n where entry i contains a list of paths from i to $\texttt{tree}[i]$, and $\texttt{tree}[i]$ is the vector describing the tree's topology.

Gusfield-init Algorithm

Let `tree` be an array of length n which encodes the topology of the resulting Gomory-Hu tree. Initially, $\texttt{tree}[i] = 1$ for all i. Ultimately, the edges of the Gomory-Hu tree will be the pairs $(i, \texttt{tree}[i])$.

Let `weights` be an array of length n which stores the weight of edge $(i, \texttt{tree}[i])$ in entry i.

Let `flow-paths` be an array of length n which, in position i, stores a list of $(i, \texttt{tree}[i])$-paths.

for $s = 2$ to n **do**

  $t = \texttt{tree}[s]$

  Compute a minimum cut $(X, \overline{X})$ between s and t where X is the set of vertices on s's side of the cut. Let $f_{s,t}$ be the size of this minimum cut, and let `st-paths` be a list of edge disjoint (s, t)-paths obtained by the max flow calculation which acquired the minimum cut.

  $\texttt{weights}[s] = f_{s,t}$

  $\texttt{flow-paths}[s] = \texttt{st-paths}$

  for $i = 1$ to n **do**

    if $i \neq s$ and $i \in X$ and $p[i] = t$ **then**

```
        reassign tree[i] = s
        Use join-paths on reverse(flow-paths[s]) and flow-paths[i] to
        construct a list of (i, s) paths, and store them in flow-paths[i].
      end if
    end for
    if tree[t] ∈ X then
      tree[s] = tree[t]
      tree[t] = s
      weights[s] = weights[t]
      weights[t] = f_{s,t}
      Use join-paths on flow-paths[s] and flow-paths[t], and store the
      result in flow-paths[s].
      Reverse every path in st-paths and store the results in flow-paths[t].
    end if
  end for
```

4.2.2 Gusfield-update

This variation of the Gusfield algorithm is used to build the Gomory-Hu tree T' for graph $G \setminus v$. Its behavior is identical to that of the first variation described above except that prior to computing a minimum cut between vertices s and t, it uses the provided flow paths from the construction of T to build a collection of (s,t)-paths in G. Those (s,t)-paths which make use of v are discarded, and the remaining paths are used by a subsequent call to the Ford-Fulkerson algorithm (a common maximum flow algorithm) to warm start its calculation.

In order to assemble the (s,t)-paths, we begin by identifying the unique (s,t)-path in the Gomory-Hu tree T. The flow paths associated with the edges on this path in T are then stitched together using the join-paths subroutine to form a set of edge disjoint (s,t)-paths in G.

4.2.3 Join-Paths

The join-paths subroutine takes as input a list of edge disjoint (a,b)-paths, and a list of edge disjoint (b,c)-paths, and returns a list of edge disjoint (a,c)-paths. We use this subroutine to preserve the $f_{s,t}$ (s,t)-paths for every edge $(s,t) \in T$ while such edges may be redirected as the algorithm iterates. The subroutine is again used to produce paths in G between vertices which may not be adjacent in T.

Subroutine Description

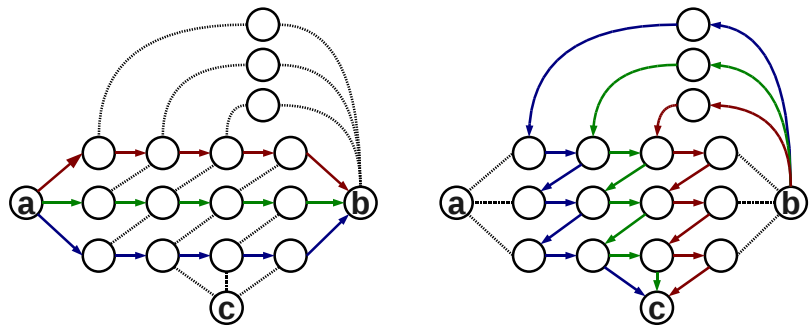
Let $F = f_{a,b} \leq f_{b,c} = F'$, $n = |V|$, and $m = |E|$. We begin by describing the data structures used in the algorithm. The inputs are a set of (a, b)-paths P_i $(1 \leq i \leq F)$ and a set of (b, c)-paths Q_j $(1 \leq j \leq F')$. The paths are represented as doubly linked lists of vertices. We assume the paths contain no cycles (we can remove them otherwise).

Other data structures used in the algorithm include:

- g_i $(1 \leq i \leq F)$ is a boolean indicating whether the ith (a, b)-path has been completed, yielding an (a, c)-path. All g_i are initially `false`.
- h_j $(1 \leq j \leq F')$ is an integer $1 \leq h_j \leq F$ indicating that the path Q_j is being used to complement the path P_i. Initially, all $h_j = 0$.
- p_i $(1 \leq i \leq F)$ is a pointer to some vertex in P_i. All p_i are initially set to a.
- q_j $(1 \leq j \leq F')$ is a pointer to some vertex in Q_j. All q_j are initially set to b.
- M is a hash table which maps edges to the index j associated with path Q_j using that edge.

The algorithm's execution proceeds as follows:

```
while ∃g_i = false do
  for i = 1 to F do
    if g_i = true then
      continue to next iteration
    end if
    g_i = true
    while the edge leading away from p_i's target in P_i is not in the table
    M and p_i does not point to the last entry of P_i do
      Advance p_i to the next entry in P_i.
    end while
    if the edge leading away from p_i's target in P_i has an entry in M
    then
      let j be the index returned by M
      if h_j ≠ 0 then
        g_{h_j} = false
      end if
      h_j = i
```



(a) (b)

Figure 4.1: These figures depict an example of input to the join-paths subroutine. Figure 4.1a shows 3 edge disjoint (a, b)-paths, and Figure 4.1b shows 3 edge disjoint (b, c)-paths.

```
        while q_j ≠ p_i do
          let q'_j be the next vertex in Q_j
          remove the edge (q_j, q'_j) from M
          q_j = q'_j
        end while
      end if
    end for
  end while
  for i = 1 to F do
    if p_i = b then
      choose j such that q_j = 0
      form R_i by concatenating P_i with Q_j
    else
      choose j = M[e] where e is the edge leading away from p_i's target in
      P_i.
      form R_i by concatenating the prefix of P_i ending with p_i with Q_j's
      suffix beginning with q_j.
    end if
  end for
  return the completed paths R_i
```

Sample Execution

Consider the graph depicted in Figure 4.1. If we process the three paths in the order (red, green, blue), then the join-paths subroutine would begin by connecting the red (a, b)-path to the blue (b, c)-path as depicted in Figure 4.2a. Subsequently, the process will attempt to use the green (a, b)-path, and find that it conflicts with the path assembled earlier. Since the green (a, b)-path intersects the blue (b, c)-path on a later edge than the red path, it gains possession of the (b, c)-path in Figure 4.2b. Similarly, when the blue (a, b)-path is advanced, it takes possession of the blue (b, c)-path as shown in Figure 4.2c.

In Figure 4.2d, we return to the red (a, b)-path which advances until it intersects the green (b, c)-path, and in Figure 4.2e, the green (a, b)-path gains possession of this (b, c)-path. Finally, the red (a, b)-path intersects the red (b, c)-path as depicted in Figure 4.2f, and the subroutine terminates since every path is complete.

Proof of Correctness

We claim that at the end of every iteration of the outermost `for` loop, every g_i set to `true` corresponds with a complete path between a and c that is edge disjoint from all other such paths. For a given starting path P_i, if p_i points to b, a complete path is formed by selecting some j where $h_j = 0$, and concatenating P_i with Q_j. Otherwise, the edge on P_i leading away from p_i's target vertex is shared with some Q_j, so the complete path involves the concatenation of the prefix of P_i ending with p_i's target and Q_j's suffix beginning with (but not including) q_j's target. We refer to P_i and $P_{i'}$ as the prefix and Q_j and $Q_{j'}$ as the suffix of S and T respectively,

Since P_i and $P_{i'}$ are derived from the same flow assignment, they are clearly edge disjoint. This implies that the prefixes of S and T are edge disjoint, and an identical argument implies that the suffixes are also edge disjoint.

Assume to the contrary that a suffix of one complete path shares an edge with the prefix of another path. Without loss of generality, assume P_i shares an edge (u, v) with $Q_{j'}$. The algorithm cannot reach such a state because at some iteration, p_i must equal u, and at this point, every edge preceding (u, v) in $Q_{j'}$ would have been removed from M thereby preventing any (a, b)-path from attaching to $Q_{j'}$ at an earlier edge. Furthermore, any (a, b)-path $P_{i''}$ previously attached to $Q_{j'}$ is "released" by reseting $g_{i''}$ to `false`. Since the intersection of these paths contradicts the state of the algorithm,

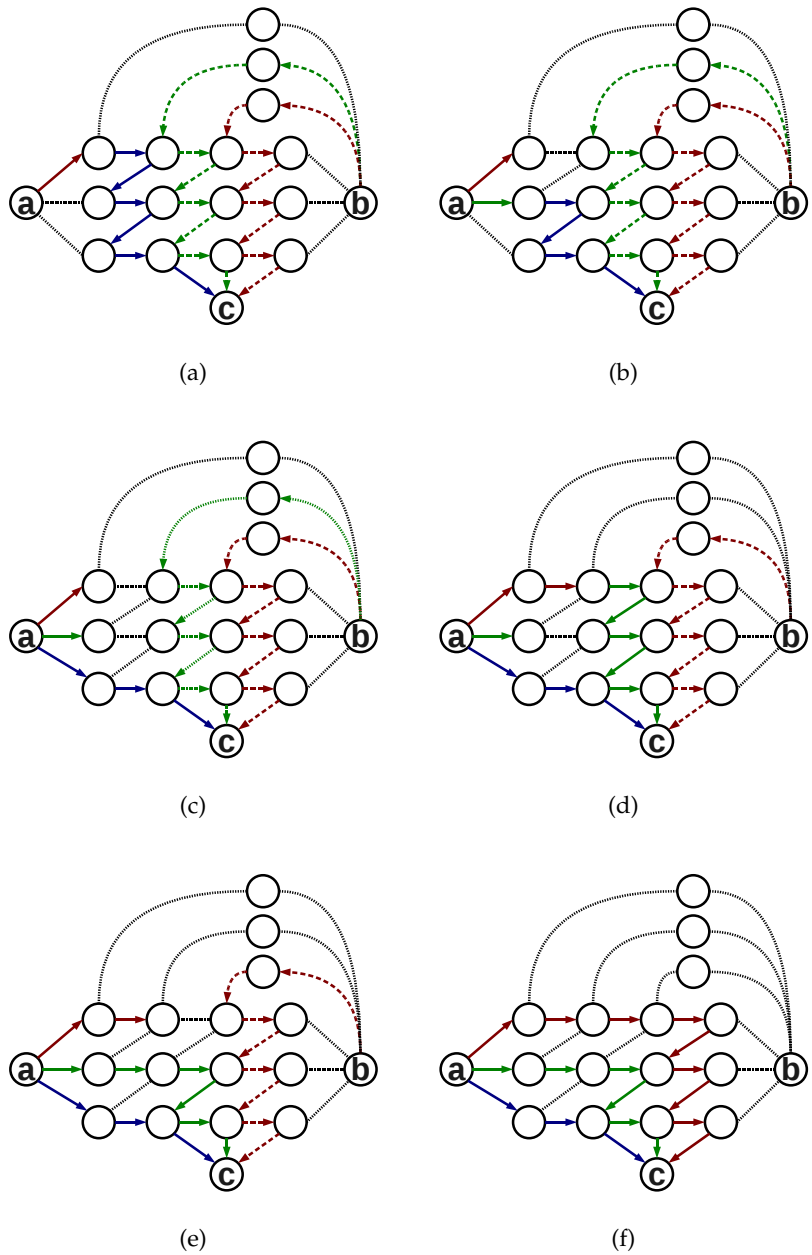


Figure 4.2: Starting from the collection of (a, b) and (b, c) edge disjoint paths from Figure 4.1, these figures demonstrate how the join-paths subroutine arrives at a set of edge disjoint (a, c) paths. The solid arcs indicate complete or in-progress (a, c)-paths.

it follows that the prefix of any complete path does not share an edge with the suffix of another, and thus that completed paths are edge disjoint.

Runtime

Each iteration of an inner-most `while` loop is associated with an edge in the graph. Furthermore, the first loop visits every edge in $\cup_i P_i$ at most once, and the second visits the edges of $\cup_j Q_j$ at most once. Thus the asymptotic, worst-case runtime is $O(\sum_i |P_i| + \sum_j |Q_j|) = O(m)$.

4.3 Experimental Performance

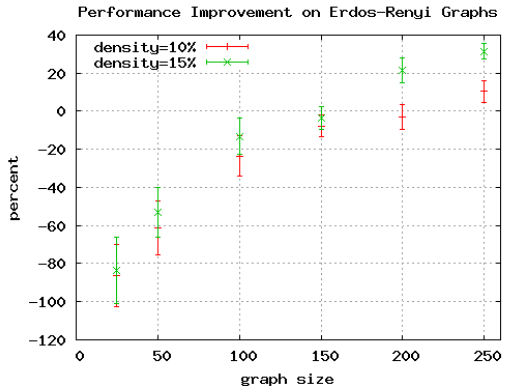
The worst-case asymptotic runtime of the warm-start algorithm described above is not better than that of the naïve brute force algorithm since in the worst case, all of the preserved edge disjoint paths are lost following the vertex deletion, and have to be rebuilt. Thus we sought to analyze expected runtime. We were unable to reach an analytically derived asymptotic expected performance, so the two algorithms were implemented and compared experimentally.

Both algorithms were implemented in Python, and tested on the random graph models proposed by Erdős and Rényi (1960), Barabási and Albert (1999), and Watts and Strogatz (1998). Erdős-Rényi graphs were tested with $p = 0.1$ and $p = 0.15$. The m parameter (which indicates the degree of a newly introduced vertex) for the Barabási-Albert model was varied with graph size to produce graphs with constant density, and m_0 (the size of the initial graph) was always chosen to be $m + 1$. For the Watts-Strogatz model we used $\beta = 0.1$ and k was varied with the graph size according to the listed density.

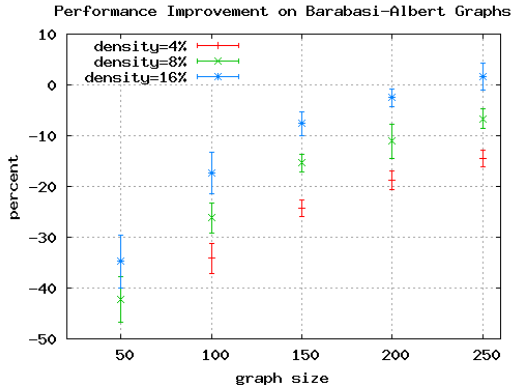
The experiments were performed on commodity personal computers with 3.0 GHz Intel Core 2 Duo processors and 4GB of RAM (note that the Python implementations were not multi-threaded and therefore did not make use of both processor cores). Each algorithm was tasked with simulating a brute force search for the load maximizing single vertex removal. Thirty graphs were generated for each size/density combination, and the algorithms were run once on each graph.

The plots in Figure 4.3 depict the results of these experiments. We observe that the proposed algorithm produces the greatest performance increase on larger graphs with greater density. Graphs produced by the Erdős-Rényi and Watts-Strogatz models benefited most from the new algo-

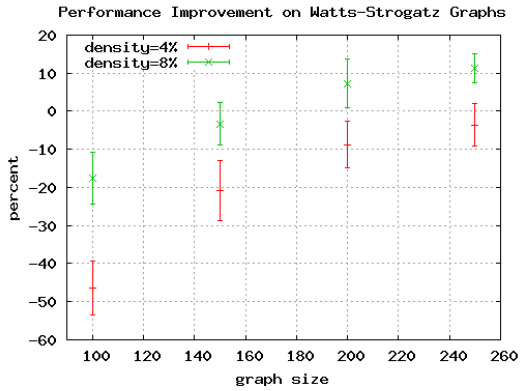
rithm, while those produced by the Barabási-Abert model required larger graphs with greater density to show a positive performance improvement.



(a)



(b)



(c)

Figure 4.3: These figures depict the warm-start algorithm's experimental performance on Erdős-Rényi (Fig. 4.3a), Barabási-Abert (Fig. 4.3b), and Watts-Strogatz (Fig. 4.3c) random graphs. The y-axis indicates the time difference scaled by the time required by the naïve algorithm, and the error bars indicate one standard deviation. Each data point was acquired from searches, on 30 different graphs, for the load maximizing single vertex removal.

Chapter 5

Future Work and Conclusion

We propose two alternative approaches to efficient Gomory-Hu tree reconstruction, and suggest the use of meta-heuristics to achieve good, if not optimal, solutions to the NP-complete problem of identifying the load maximizing subset removal.

5.1 Alternative Gomory-Hu Tree Reconstruction Algorithm

An alternative strategy for the recalculation of Gomory-Hu trees may work by using the prior tree to determine whether the newly deleted vertex is incident to any edge in a minimum cut between a pair of vertices. We prove that in the process of constructing a Gomory-Hu tree, the Gusfield algorithm encounters every minimum cut in the graph. This suggests that an algorithm which produces the set of vertices incident to an edge that is a member of a minimum cut could lead to an improved algorithm.

Picard and Queyranne (1980) describe an algorithm which generates minimum cuts, but since the number of minimum cuts between vertices may grow exponentially, such a process is not sufficient.

5.1.1 Proof that Gusfield's Algorithm Encounters All Minimum Cuts

A Gomory-Hu tree, T, only guarantees to encode one minimum cut between every pair of vertices (specifically, the minimum cut represented by the lightest edge on the unique path between those vertices in T). Consider again the Gomory-Hu tree of a cycle graph. As demonstrated in the proof

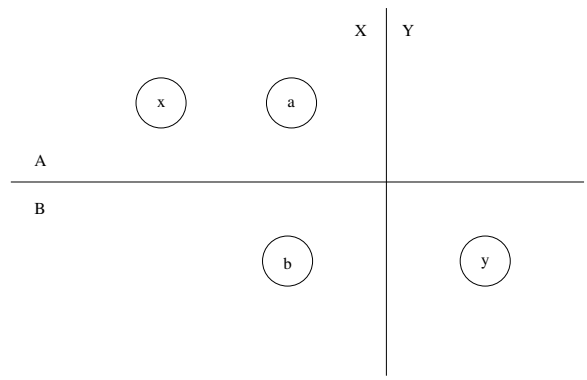


Figure 5.1: This figure depicts the setup for the second case in Proposition 5.1 where vertices x and y are placed in separate supervertices as a result of a minimum cut between a and b.

of 3.1, a star graph is a valid Gomory-Hu tree for this graph, but note that between the center vertex of the star, and any other vertex, only one minimum cut is encoded despite the existence of many more. We now prove that every minimum cut in a graph is encoded in the flow assignments obtained while running the Gusfield algorithm.

Proposition 5.1. *A single run of the Gusfield algorithm for constructing a Gomory-Hu tree will encounter every minimum cut in the graph.*

Proof. We say that the algorithm has encountered a minimum cut, (S, T), of weight w if it computes a maximum flow between some vertex $s \in S$ and another vertex $t \in T$ and $f_{s,t} = w$.

Let $G = (V, E)$ be a graph, and consider a minimum cut (X, Y) between vertices x, y. At some point in the algorithm's execution, two vertices in the same supervertex as x and y (potentially x and y themselves) will be selected, a minimum cut computed between them, and two new supervertices S_1 and S_2 will be formed with $x \in S_1$ and $y \in S_2$. In the case where these vertices are x and y, the (x, y)-cut will clearly encounter (X, Y), but the algorithm may split x and y when computing the minimum cut between some other vertices a and b yielding a cut (A, B) such that $x \in A$ and $y \in B$. Alternatively, x and y may be split into separate supervertices by a cut between x and some b. We will show that in either of these cases, there will continue to exist a pair of vertices in some supervertex of the growing Gomory-Hu tree for which (X, Y) is a minimum cut until the cut (X, Y) is encountered.

In the latter case, let the minimum (x, b)-cut which separates x and y be (A, B) where $x \in A$ and $b \in B$. We first note that if $b \in Y$, (X, Y) is an (x, b) cut and (A, B) is an (x, y)-cut, so $f_{x,y} = f_{b,x}$, and thus the flow calculation between x and b encounters the cut (X, Y). If $b \in X$, note that (X, Y) is a (b, y) cut, so

$$f_{b,y} \leq f_{x,y}. \tag{5.1}$$

Additionally, (A, B) is an (x, y)-cut, so

$$f_{x,y} \leq f_{x,b}. \tag{5.2}$$

Now we will apply an inequality introduced in Gomory and Hu's paper which states

$$f_{v_1,v_k} \geq \min\{f_{v_1,v_2}, f_{v_2,v_3}, \cdots, f_{v_{k-1},v_k}\}.$$

This implies that $f_{b,y} \geq \min\{f_{b,x}, f_{x,y}\}$. Together with Equation 5.2, this yields that $f_{x,y} \leq f_{b,y}$, and this combines with Equation 5.1 to conclude that $f_{b,y} = f_{x,y}$. Thus, following the (a, b)-cut, y and b will share the same supervertex, and have (X, Y) as a minimum cut between them. So while our cut of interest has not been encountered in this iteration, it is guaranteed to be encountered in a later iteration.

Now we consider the case depicted in Figure 5.1 where x and y are separated by a cut between two other vertices a and b. Notice that (A, B) is then an (x, y)-cut, so it follows that $f_{x,y} \leq f_{a,b}$ where $f_{u,v}$ is the maximum flow value between u and v. Equality would imply that the (a, b) flow computation encountered (X, Y), so we only consider the case where

$$f_{x,y} < f_{a,b}. \tag{5.3}$$

Furthermore, placing a and b on different sides of (X, Y) would also imply that the cut has been encountered by the (a, b) flow calculation, so without loss of generality, let $a, b \in X$.

We wish to show that one of the new supervertices obtained by the (a, b)-cut will contain a pair of vertices which have (X, Y) as a minimum cut. In particular, these vertices will be b and y, so we wish to show that $f_{b,y} = f_{x,y}$. Since $b \in X$ and $y \in Y$, (X, Y) is a (b, y)-cut and it follows that $f_{b,y} \leq f_{x,y}$, so now we need only demonstrate the reverse inequality

$$f_{x,y} \leq f_{b,y}. \tag{5.4}$$

Assume to the contrary that

$$f_{b,y} < f_{x,y}. \tag{5.5}$$

We begin by noting two additional inequalities. Since (X, Y) is an (a, y)-cut, it follows that

$$f_{a,y} \leq f_{x,y}, \tag{5.6}$$

and since (A, B) is a (b, x)-cut, it follows that

$$f_{b,x} \leq f_{a,b}. \tag{5.7}$$

We again apply the inequality

$$f_{v_1,v_k} \geq \min\{f_{v_1,v_2}, f_{v_2,v_3}, \cdots, f_{v_{k-1},v_k}\}.$$

- Equation 5.7 and $f_{a,x} \geq \min\{f_{a,b}, f_{b,x}\}$ together imply

$$f_{a,x} \geq f_{b,x}. \tag{5.8}$$

- Equation 5.6, Equation 5.3, and $f_{b,y} \geq \min\{f_{a,b}, f_{a,y}\}$ together imply

$$f_{b,y} \geq f_{a,y}. \tag{5.9}$$

- Equation 5.5 and $f_{b,x} \geq \min\{f_{b,y}, f_{x,y}\}$ together imply

$$f_{b,x} \geq f_{b,y}. \tag{5.10}$$

- Equation 5.5, Equation 5.3, and $f_{b,y} \geq \min\{f_{a,b}, f_{a,x}, f_{x,y}\}$ together imply

$$f_{b,y} \geq f_{a,x}. \tag{5.11}$$

- Equation 5.7, Equation 5.5, Equation 5.11, Equation 5.8, and $f_{a,y} \geq \min\{f_{a,b}, f_{b,x}, f_{x,y}\}$ together imply

$$f_{a,y} \geq f_{b,x}. \tag{5.12}$$

These inequalities together yield

$$f_{a,x} = f_{a,y} = f_{b,x} = f_{b,y} < f_{x,y} < f_{a,b} \tag{5.13}$$

Let $C(U)$, where $U \subseteq \{a, b, x, y\}$, denote the weight of the minimum cut which disconnects each vertex in U from every vertex in $\{a, b, x, y\} - U$.

In general,

$$f_{x,y} \leq \min\{C(a, b, y), C(a, b, x), C(a, x), C(a, y)\}, \tag{5.14}$$

but due to our earlier stipulation that $a, b \in X$, we find that $f_{x,y} = C(a, b, x)$. Similarly,

$$f_{a,b} \leq \min\{C(a), C(a, x), C(a, y), C(a, x, y)\} \tag{5.15}$$

in general, and in our case, $f_{a,b} = C(a, x)$.

Equation 5.3 along with equations 5.14 and 5.15 then imply that

$$f_{x,y} \leq c, c \in \{C(a), C(a, x), C(a, y), C(a, b, x), C(a, b, y), C(a, x, y)\}. \tag{5.16}$$

Equation 5.13 implies that none of the cuts in Equation 5.16 have weight $f_{a,x} = f_{a,y} = f_{b,x} = f_{b,y}$ leaving the conclusion that

$$f_{a,x} = f_{a,y} = f_{b,x} = f_{b,y} = C(a, b). \tag{5.17}$$

Gomory and Hu proved the following lemma in their 1961 paper.

Lemma 5.1. *Let (S, T) be a minimum cut in G separating vertices $s \in S$ and $t \in T$. Let $u, v \in S$, and let (U, V) be a minimum (u, v)-cut. If $t \in U$, then $(U', V') = (U \cup T, V \cap S)$ is another minimum (u, v)-cut. If $t \in V$, $(U', V') = (U \cap S, V \cup T)$ is another minimum (u, v)-cut.*

We can apply this lemma to the crossing (a, b) and (b, y) cuts. Let $s, t = b, a$, and $u, v = b, y$. Notice then that $u, v \in S$ (since $b, y \in B$), and that $t \in U$ (since we proved that $f_{a,x} = C(a, b)$). It then follows that $(A \cup U, B \cap V)$ is a minimum (b, y)-cut, and since this cut isolates y from a, b, and x, $f_{b,y} = C(a, b, x)$. Finally, this contradicts equations 5.5 and 5.14.

□

5.2 Additional Future Work

The algorithm described in this thesis performs the Gomory-Hu tree calculation using an algorithm based on minimum cuts, but recent work by Bhalgat et al. (2007) shows that Gomory-Hu tree calculations could be performed with expected time $\widetilde{O}(mn)$ using a randomized algorithm that applies an efficient tree packing algorithm to compute Steiner edge connectivity as its main subroutine . This algorithm may be better suited for Gomory-Hu tree recalculation.

Since identifying the optimally load maximizing subset of vertices to remove is NP-complete, a polynomial-time algorithm is unlikely to exist for the problem as currently defined. Meta-heuristics such as simulated annealing or genetic algorithms may prove successful at achieving locally maximal results.

5.3 Conclusion

This thesis sought to explore the computational feasibility of identifying the load maximizing vertex set removal. We began by proving that the problem is NP-complete, and therefore is not likely to have a polynomial time solution. We then sought to improve the efficiency of a brute force algorithm for identifying the load maximizing single vertex removal.

To do so, we considered designing an algorithm which efficiently reconstructs Gomory-Hu trees following a vertex deletion. When considering the feasibility of using Gomory-Hu trees which are most easily preserved under vertex deletions, we proved that the Gomory-Hu tree of a graph produced by Gusfield's algorithm is not uniquely defined by the chosen permutation of vertices since different choices in minimum cuts can also produce different Gomory-Hu trees. This implied that such an approach was unlikely to succeed.

Finally, we proposed an algorithm which uses flow assignments from a Gomory-Hu tree construction for G to provide a warm start to the maximum flow subroutines in the construction of a new Gomory-Hu tree for $G \setminus \{v\}$. An implementation of this algorithm was tested against an implementation of the naïve brute force algorithm and performance improvements were observed on sufficiently large and dense Erdős-Rényi, Barabási-Abert, and Watts-Strogatz random graphs.

Appendix A

The Gusfield Algorithm for Gomory-Hu Tree Construction

The Gusfield (1990) algorithm for constructing Gomory-Hu trees improves on the original algorithm proposed by Gomory and Hu by avoiding maximum flow calculations in graphs with contracted edges. Thus the new algorithm is both easier to implement and to describe.

Given a graph $G = (V, E)$ with an arbitrary ordering on the vertices $V = \{v_1, v_2, \ldots, v_n\}$. Begin by creating a new graph T containing only one supervertex, S_1, represented by v_1 which contains all vertices in V. As the algorithm progresses, we will divide this supervertex until all supervertices are made up of exactly one vertex of G, and the resulting tree will be the returned Gomory-Hu tree.

Until each supervertex contains only one vertex, we arbitrarily select the lowest-valued supervertex $S_x \in V(T)$ (represented by vertex x) and split it as follows. Select the lowest-valued non-representative vertex in S_x, y, and compute a minimum cut, (X, Y), between x and y in the original graph G. The old supervertex S_x now divides into $S'_x = X \cap S_i$ which is represented by x, and $S_y = Y \cap S_i$ which is represented by y. After splitting S_x, all edges incident to this supervertex must be updated. Let $e = \{S_x, S_z\}$ be such an edge, and let z be the representative vertex of S_z. If $z \in X$, replace e with $\{S'_x, S_z\}$, and if $z \in Y$, replace e with $\{S_y, S_z\}$.

Gusfield also provides an alternative representation of this algorithm described in the pseudocode below.

The n-vector `tree` encodes the topology of the resulting Gomory-Hu tree. Initially, $\texttt{tree}[i] = 1$ for all i. Ultimately, the edges of the Gomory-Hu tree will be the pairs $(i, \texttt{tree}[i])$.

The n-vector `weights` stores the weight of edge $(i, \texttt{tree}[i])$ in entry i.
for $s = 2$ to n **do**
  Compute a minimum cut between s and $t = \texttt{tree}[s]$. Let X be the set of vertices on s's side of the cut, and let $f_{s,t}$ be the resulting flow value.

  $\texttt{weights}[s] = f_{s,t}$
  for $i = 1$ to n **do**
    if $i \neq s$ and $i \in X$ and $p[i] = t$ **then**
      reassign $\texttt{tree}[i] = s$
    end if
  end for
  if $\texttt{tree}[t] \in X$ **then**
    $\texttt{tree}[s] = \texttt{tree}[t]$
    $\texttt{tree}[t] = s$
    $\texttt{weights}[s] = \texttt{weights}[t]$
    $\texttt{weights}[t] = f_{s,t}$
  end if
end for

Bibliography

Altner, D.S. 2008. *Advancements on Problems Involving Maximum Flows*. Ph.D. thesis, Georgia Institute of Technology.

Barabási, A.L., and R. Albert. 1999. Emergence of scaling in random networks. *Science* 286(5439):509.

Bhalgat, A., R. Hariharan, T. Kavitha, and D. Panigrahi. 2007. An $\widetilde{O}(mn)$ Gomory-Hu tree construction algorithm for unweighted graphs. In *Annual ACM Symposium on Theory of Computing*, vol. 39, 605.

Carpenter, T., G. Karakostas, and D. Shallcross. 2002. Practical issues and algorithms for analyzing terrorist networks. *Proceedings of the Western Simulation MultiConference* .

Erdős, P., and A. Rényi. 1960. On the evolution of random graphs. *Publ Math Inst Hung Acad Sci* 5:17–61.

Goldberg, A.V., and K. Tsioutsiouliklis. 2001. Cut tree algorithms: an experimental study. *Journal of Algorithms* 38(1):51–83.

Gomory, R.E., and T.C. Hu. 1961. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics* 551–570.

Gusfield, D. 1990. Very simple methods for all pairs network flow analysis. *SIAM Journal on Computing* 19:143.

Martonosi, S.E., D.S. Altner, M. Ernst, and S. Plott. 2009. Disrupting terrorist networks. Working Paper.

Picard, J.C., and M. Queyranne. 1980. On the structure of all minimum cuts in a network and applications. *Combinatorial Optimization II* 8–16.

Watts, D.J., and S.H. Strogatz. 1998. Collective dynamics of 'small-world' networks. *Nature* 393(6684):440–442.