

# Algorithms for Connectivity Problems in Undirected Graphs: Maximum Flow and Minimum k-Way Cut

by

Matthew S. Levine

S.M. Computer Science  
Massachusetts Institute of Technology, 1997

A.B. Computer Science  
Princeton University, 1995

Submitted to the Department of Electrical Engineering and Computer Science in  
Partial Fulfillment of the Requirements for the Degree of

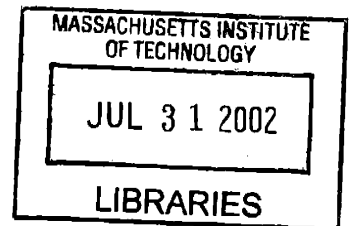
Doctor of Philosophy in Computer Science  
at the

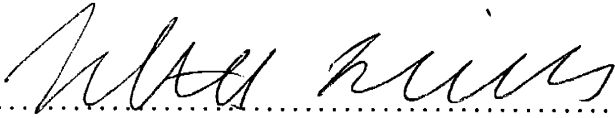
Massachusetts Institute of Technology

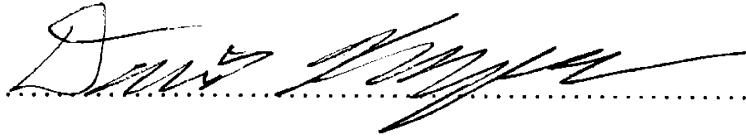
June 2002

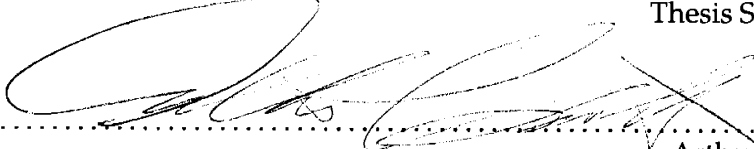
©2002 Massachusetts Institute of Technology  
All Rights Reserved

ARCHIVES



Signature of Author .....   
Department of Electrical Engineering and Computer Science  
May 3, 2002

Certified by .....   
David R. Karger  
Associate Professor of Electrical Engineering and Computer Science  
Thesis Supervisor

Accepted by .....   
Arthur C. Smith  
Chairman, Department Committee on Graduate Students

# Algorithms for Connectivity Problems in Undirected Graphs: Maximum Flow and Minimum k-Way Cut

by

Matthew S. Levine

Submitted to the Department of Electrical Engineering and  
Computer Science on May 3, 2002 in Partial Fulfillment of the  
Requirements for the Degree of Doctor of Philosophy in Computer Science

## ABSTRACT

We consider two connectivity problems on undirected graphs: maximum flow and minimum k-way cut.

The maximum flow problem asks about the connectivity between two specified nodes. A traditional approach is to search for augmenting paths. We explore the possibility of restricting the set of edges in which we search for an augmenting path, so that we can find each flow path in sub-linear time. Consider an  $n$ -vertex,  $m$ -edge, undirected graph with maximum flow value  $v$ . We give two methods for finding augmenting paths in such a graph in amortized sub-linear time, based on the idea that undirected graphs have sparse subgraphs that capture connectivity information. The first method sparsifies unused edges by using a spanning forest. It is deterministic and takes  $O(n\sqrt{v})$  time per path on average. The second method sparsifies the entire residual graph by taking random samples of the edges. It takes  $\tilde{O}(n)$  time per path on average. These results let us improve the  $O(mv)$  time bound of the classic augmenting path algorithm to  $O(m + nv^{3/2})$  (deterministic) and  $\tilde{O}(m + nv)$  (randomized). For simple graphs, the addition of a blocking flow subroutine yields a deterministic  $O(nm^{2/3}v^{1/6})$ -time algorithm.

A minimum k-way cut of an  $n$ -vertex,  $m$ -edge, weighted, undirected graph is a partition of the vertices into  $k$  sets that minimizes the total weight of edges with endpoints in different sets. We give new randomized algorithms to find minimum 3-way and 4-way cuts, which lead to time bounds of  $O(mn^{k-2} \log^3 n)$  for  $k \leq 6$ . Our key insight is that two different structural properties of k-way cuts, both exploited by previous algorithms, can be exploited simultaneously to avoid the bottleneck operations in both prior algorithms. The result is that we improve on the best previous time bounds by a factor of  $\tilde{\Theta}(n^2)$ .

Thesis Supervisor: David R. Karger

Title: Associate Professor of Electrical Engineering and Computer Science

# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	The Maximum Flow Problem . . . . .	8
1.1.1	History . . . . .	8
1.1.2	Our Contribution . . . . .	10
1.2	The Minimum k-Way Cut Problem . . . . .	13
1.2.1	History . . . . .	13
1.2.2	Our Contribution . . . . .	14
<b>2</b>	<b>Flow Background</b>	<b>17</b>
2.1	Maximum Flow Fundamentals . . . . .	17
2.2	Residual Graphs and Augmenting Paths . . . . .	19
2.3	Shortest Augmenting Paths, Blocking Flow and Simple Graphs . . . . .	20
2.4	Undirected Graph Sparsification . . . . .	21
2.5	Random Sampling in Undirected Graphs . . . . .	22
<b>3</b>	<b>Spanning Forests of Undirected Edges</b>	<b>25</b>
3.1	Small Flows Use Few Edges . . . . .	26
3.2	A Conceptually Easy Algorithm for Fast Augmenting Paths . . . . .	28
3.3	A Better Algorithm for Fast Augmenting Paths . . . . .	31
3.4	Deterministic Algorithms Using Fast Augmenting Paths . . . . .	33
<b>4</b>	<b>Sampling in Residual Graphs</b>	<b>35</b>
4.1	The Algorithm . . . . .	36
4.2	The Analysis . . . . .	36
4.2.1	Supporting Lemmas . . . . .	38
4.2.2	Proof of the Main Theorem . . . . .	39

<b>5</b>	<b>Minimum k-Way Cuts</b>	<b>43</b>
5.1	Background . . . . .	44
5.2	Minimum 3-Way Cuts . . . . .	45
5.2.1	Structural Results . . . . .	45
5.2.2	The Algorithm . . . . .	47
5.2.3	Correctness . . . . .	48
5.2.4	Running Time Analysis . . . . .	48
5.3	Minimum 4-Way Cuts . . . . .	50
5.3.1	Structural Results . . . . .	50
5.3.2	The Algorithm . . . . .	51
5.3.3	Correctness . . . . .	52
5.3.4	Running Time Analysis . . . . .	52
5.4	Minimum 5-Way Cuts, 6-Way Cuts and Beyond . . . . .	53
<b>6</b>	<b>Conclusion</b>	<b>55</b>
6.1	Maximum Flow . . . . .	55
6.2	Minimum k-Way Cut . . . . .	57
<b>A</b>	<b>Randomized Algorithms Using Fast Augmenting Paths</b>	<b>59</b>
A.1	New Tricks for an Old DAUG . . . . .	59
A.2	$\tilde{O}(m + nv^{5/4})$ - and $\tilde{O}(m + n^{11/9}v)$ -Time Algorithms . . . . .	61
<b>B</b>	<b>Compression of Residual Graphs</b>	<b>63</b>
B.1	Supporting Definitions and Lemmas . . . . .	64
B.2	The Proof . . . . .	65

# List of Figures

1.1	Picture of the best deterministic bounds, for simple graphs on the left and for general graphs on the right . . . . .	12
1.2	Pictures of the best randomized bounds . . . . .	13
2.1	A flow that blocks all $s$ - $t$ paths but is not maximum . . . . .	19
3.1	A graph with an acyclic flow that uses $O(n\sqrt{v})$ edges . . . . .	28
3.2	Basic algorithm for fast augmenting paths . . . . .	30
3.3	First algorithm for fast augmenting paths based on sparse certificates . . . . .	31
3.4	Final algorithm for fast augmenting paths . . . . .	32
4.1	Maximum flow algorithm based on sampling the residual graph . . . . .	36
5.1	The tree corresponding to a laminar collection . . . . .	45
5.2	Crossing cuts . . . . .	46
5.3	Minimum 3-way cut algorithm . . . . .	48
5.4	Attempt to insert $X = \{b, c, d\}$ . Shaded nodes are "subsets of $X$ ". They do not have a common parent because $X$ overlaps $Y$ . Observe that for a different $X$ , such as $\{a, b, c\}$ , where there are containments but no overlap, the subsets of $X$ would have a common parent. . . . .	49
5.5	Either $c(X_1)$ or $c(X - X_1)$ has value at most $c_2(X) + c_2/2$ . . . . .	52
5.6	Minimum 4-way cut algorithm . . . . .	53
5.7	Small cuts can cross $V_1$ for $k \geq 5$ . . . . .	54
6.1	With edges directed from left the right, this graph is a flow problem representing a maximum bipartite matching problem; if the edge directions are removed then the flow value increases, no longer corresponding to a matching. . . . .	57
A.1	The original "divide and augment" algorithm . . . . .	60



# Chapter 1

## Introduction

In this thesis we consider two connectivity problems on undirected graphs: maximum flow and minimum  $k$ -way cut.

The maximum flow problem asks about the connectivity between two specified nodes. In the unit-capacity case, this question reduces to finding a maximum number of edge-disjoint paths between the two nodes. The classic algorithm for flow does a linear time search for each path. We explore the possibility of restricting the set of edges in which we search for a flow, so that we can find each path in sub-linear time. For an  $n$ -vertex,  $m$ -edge, undirected graph with maximum flow value  $v$ , we give a deterministic algorithm that finds flow in  $O(n\sqrt{v})$  amortized time per path by reducing unused edges down to a spanning forest, and a randomized algorithm that finds flow in  $\tilde{O}(n)$ <sup>1</sup> amortized time per path by considering only a random sample of the edges of the residual graph.

A minimum  $k$ -way cut of an  $n$ -vertex,  $m$ -edge, weighted, undirected graph is a partition of the vertices into  $k$  sets that minimizes the total weight of edges with endpoints in different sets. We give new randomized algorithms to find minimum 3-way and 4-way cuts, which lead to time bounds of  $O(mn^{k-2} \log^3 n)$  for  $k \leq 6$ . Our key insight is that two different structural properties of  $k$ -way cuts, both exploited by previous algorithms, can be exploited simultaneously to avoid the bottleneck operations in both prior algorithms. The result is that we improve on the best previous time bounds by a factor of  $\tilde{\Theta}(n^2)$ .

The plan of the thesis is as follows. In the remainder of this chapter we give an overview of the history of the problems, and summarize our contribution in greater detail. Chapter 2 is background. It includes our notation, but otherwise summarizes some relevant past results and can easily be skipped by readers who are already familiar with those results. Chapter 3 gives our deterministic maximum flow algorithms based on summarizing the portion of the graph not carrying flow with a spanning forest. Chapter 4 gives our randomized maximum flow algorithm based on taking random samples of the edges of the residual graph. Chapter 5 gives our minimum  $k$ -way cut algorithms. Chapter 6 wraps up, summarizing our results and reviewing some open problems. Finally, two minor results that are of conceivable interest are in appendices. Appendix A has some randomized algorithms that result from combining our

---

<sup>1</sup>We write  $f(n) = \tilde{O}(g(n))$  if  $\exists c$  such that  $f(n) = O(g(n) \log^c n)$

spanning forest techniques with some past algorithms; Appendix B has an extension of the key theorem used in random sampling from residual graphs.

## 1.1 The Maximum Flow Problem

### 1.1.1 History

The maximum flow problem is a fundamental optimization problem and one of the older problems in computer science, having been studied extensively since the seminal paper of Ford and Fulkerson in 1956 [FF56], and reviews of the various results now fill chapters of textbooks (*cf.* [AMO93]). Yet it remains unclear whether we truly understand how to solve instances quickly.

The problem is normally stated for a directed graph with capacities on the edges, and is a generalization of the problem of finding edge-disjoint paths from a specified start node (called the source) to a specified end node (called the sink). Specifically, a flow is an assignment to the directed edges such that (1) the flow on an edge is less than its capacity and (2) for any node other than the source or sink, the flow entering the node equals the flow leaving the node. The value of a flow is the net flow leaving the source (which is the same as the net flow entering the sink), and the goal is to maximize the value. When we say we will study the problem on undirected graphs, what we mean is that we require that the edge capacities be symmetric: the capacity of edge  $(u, v)$  always equals the capacity of edge  $(v, u)$ . As a result we will sometimes discuss the capacity of an edge without reference to the direction, because the direction does not matter. Note that we do not change the definition of flow at all—the flow is certainly still directed. In order to state the running times of algorithms, we will use  $n$  for the number of nodes,  $m$  for the number of edges and  $v$  for the value of the maximum flow (we will restrict our attention to integer capacities<sup>2</sup>, so  $v$  is an integer).

The original Ford-Fulkerson algorithm relies on the concept of augmenting paths—basically a path from source to sink on which more flow can be sent. In pure form, in the worst case the algorithm does a  $\Theta(m)$ -time search to find each unit of flow, resulting in an  $O(mv)$  time bound. This running time is good if the flow value is small, but it is obviously not so good if capacities are large. In 1972 Edmonds and Karp [EK72] showed that simply using shortest augmenting paths leads to a strongly polynomial running time bound. By 1980, a sequence of improvements had led to an algorithm that runs in  $\tilde{O}(mn)$  time. Over the next decade various results whittled away at the logarithmic factors on the time bounds, leading to the best known strongly polynomial time bound of  $O(nm \log_{m/(n \log n)} n)$ , achieved by an algorithm of King, Rao and Tarjan [KRT94]. Meanwhile there was one other result that focused on small flows: Even and Tarjan [ET75] and Karzanov [Kar73] showed that on *simple* graphs—unit-capacity graphs with no parallel edges—the blocking flow method of Diniz [Din70] runs in  $O(m \min\{n^{2/3}, m^{1/2}, v\})$  time.

So by 1994 the maximum flow problem seemed reasonably well understood. The best

---

<sup>2</sup>Rational capacities can always be converted to integers by multiplying them up. Real capacities are somewhat different, although they can be approximated by integers



running time in general was not quite as small as  $O(mn)$ , but the bound was very close, and it was known that faster algorithms existed if the graph was simple or the flow value was very small.

Then new results started to change the picture. Goldberg and Rao gave an algorithm that goes a long way toward closing the gap between simple graphs and capacitated graphs in the directed case [GR97a]. Their algorithm runs in  $\tilde{O}(m \min\{n^{2/3}, m^{1/2}\} \log v)$  time. It is only weakly polynomial, so while they just about close the gap between unit capacities and integer capacities in directed graphs, there remains a gap between weakly polynomial algorithms and strongly polynomial algorithms.

Meanwhile, researchers began to consider undirected graphs. First, Nagamochi and Ibaraki [NI92b] gave a linear time algorithm that finds a set of  $nv$  edges that are sufficient for a flow of value  $v$ . This “sparsification” technique allows one to substitute  $nv$  for  $m$  in the time of any flow algorithm. In particular, it allows one to improve the running time of augmenting paths from  $O(mv)$  to  $O(m + nv^2)$ . For the worst case of simple graphs, this is still  $O(n^3)$  time, but the idea turned out to be important.

At this point Karger really opened the question of how much easier it might be to find flows in undirected graphs, first by giving an  $\tilde{O}(mv/\sqrt{c})$ -time algorithm for graphs with edge connectivity  $c$  [Kar99], and then by giving an  $\tilde{O}(m^{2/3}n^{1/3}v)$ -time algorithm for simple graphs [Kar97]. The latter result opened a gap between what was known for directed graphs and undirected graphs and raised the question of whether randomization can help.

Shortly afterward, Goldberg and Rao discovered that a combination of Even-Tarjan and Nagamochi-Ibaraki gives a deterministic algorithm for simple, undirected graphs that runs in  $O(n\sqrt{nm})$  time [GR97b]. This algorithm introduced a gap between directed graphs and undirected graphs, even without randomization.

Karger followed Goldberg and Rao with another randomized algorithm that runs in  $\tilde{O}(v\sqrt{nm})$  time, thereby reestablishing a gap between randomized algorithms and deterministic algorithms [Kar98].

Our results improve the time bounds for small flows in undirected graphs, widening several existing gaps. In particular, we widen the gap between randomized algorithms and deterministic algorithms for small flow values, and we widen the gaps between algorithms for small flow values and algorithms for large flow values for both deterministic and randomized algorithms.

Thus the current state of affairs looks far more unsettled than it did a decade ago. For simple undirected graphs, better randomized algorithms are known than deterministic algorithms. For small flow values, better algorithms for undirected graphs are known than for directed graphs. Yet unlike directed graphs, results for larger flow values in undirected graphs are not as good as those known for small flow values. And for both directed and undirected graphs there is a substantial gap in running time between weakly polynomial algorithms and strongly polynomial algorithms.

### 1.1.2 Our Contribution

Our contribution to maximum flow is two methods for finding augmenting paths in undirected graphs in amortized sub-linear time. Finding an augmenting path normally takes linear ( $O(m)$ ) time, because one does a graph search, for example, a breadth-first search. The underlying idea of both of our methods is that in an undirected graph, information about which nodes are connected can easily be captured by a small subgraph, so it should not be necessary to search all  $m$  edges. In particular, in an undirected graph, a spanning forest has  $O(n)$  edges, yet contains a path between any two nodes that are connected in the original graph. Likewise a suitable random sample of  $O(n \log n)$  edges will preserve connectivity information.

It would be useless to us that a spanning forest is small and contains a path if  $O(m)$  time was required to find a new spanning forest after we augmented on one path, but there are data structures that can be used to avoid this problem. Specifically, a spanning forest can be maintained under edge insertion and deletion in  $O(\log^2 n)$  time per update [HdLT98]. Similarly, a new random sample can be chosen in  $O(\log n)$  time per edge selected.

At this point it may sound like the result will be trivial, but there remains a serious obstacle: we need to find paths in a residual graph (loosely the original graph minus the flow), and since the flow is directed the residual graph is directed. So as soon as we find one path, the graph in which we want to find another path is directed. Since both techniques are for undirected graphs, it is not clear that either technique is still of any use.

For spanning forests, we handle the problem of directed edges by only using a spanning forest on the “undirected part” of the residual graph. More precisely, in order to look for an augmenting path we take all of the directed edges of the residual graph, but only a spanning forest of the undirected edges (those for which the capacity is still symmetric). It makes sense that this approach would be correct, because there is no harm in reducing undirected edges to a spanning forest, but it is not obvious that the method should be fast. At first glance one might think that all edges are liable to become directed, and then we would not gain anything. We show that it is always possible to restrict a flow to using  $O(n\sqrt{v})$  edges, so only  $O(n\sqrt{v})$  edges become directed and we can find augmenting paths in  $O(n\sqrt{v})$  time per path.

For random samples, we handle the problem by showing that the asymmetry the flow creates in the residual graph causes little enough damage to the cut structure of the graph that we can continue to prove good results about samples. Thus we iterate on the residual graph, sampling all edges, whether they have symmetric capacities or not. Our proof gets us the result we hoped for— $\tilde{O}(n)$  time per path on average.

### Spanning Forests of Undirected Edges

A spanning forest of an undirected graph has two desirable properties as far as flow is concerned: it has at most  $n - 1$  edges, and it contains a path between any two nodes that are connected in the original graph. Based on these properties, it is natural to hope that it would be possible to find a maximum flow in  $O(n)$  time per augmenting path, which would give a total time of  $O(m + nv)$ . The obstacle is that as soon as some edges are used by a flow

they become directed in the residual graph, so we no longer have an undirected graph in which to find a spanning forest. So instead, we only use a spanning forest on the edges that are still undirected (for which capacities are still symmetric). That is, when we go to find an augmenting path, we search in the directed edges plus a spanning forest on the undirected edges.

The key to this method being fast is that we show that a flow need not use more than  $O(n\sqrt{v})$  edges. This bound allows an algorithm in which each search for an augmenting path only looks at  $O(n\sqrt{v})$  edges, and therefore only takes that much time per path.

Replacing undirected edges with a spanning forest gives simple deterministic algorithms that are faster than all previous ones for the most difficult values of  $m$  and  $v$  on simple graphs. First, we can find flow by augmenting paths in  $O(m + nv^{3/2})$  time (substituting  $O(n\sqrt{v})$  for  $m$  in the classic  $O(mv)$ -time algorithm). Second, by incorporating a blocking flow subroutine, we can find flow in  $O(nm^{2/3}v^{1/6})$  time. The first algorithm is the best known deterministic algorithm for dense graphs with small  $v$ ; the second algorithm is the best known deterministic algorithm for dense graphs with large  $v$ . The second time bound is also at least as good as the Goldberg-Rao time bound of  $O(n^{3/2}m^{1/2})$  for all values of  $m$  and  $v$ . Both algorithms are clearly practical to implement, so only experiments will tell which methods are actually best for practical purposes. The first algorithm works for the capacitated case as well, running in  $\tilde{O}(m + nv^{3/2})$  time.

When combined with some prior algorithms, our sparsification technique also gives some randomized algorithms that run in  $\tilde{O}(m + nv^{5/4})$  time and  $\tilde{O}(m + n^{11/9}v)$  time. At the time these were discovered, they were the best randomized algorithms for finding small flows in undirected graphs for certain values of  $m$  and  $v$ . They are now dominated by the algorithm given in Chapter 4. Since it is always possible that an idea used by these algorithms could be useful in improving on the results of Chapter 3, we describe them in Appendix A.

We have so far described our results in terms of trying to restrict the search for an augmenting path to a small subgraph. Another way to look at our results is as follows. We prove that a flow of value  $v$  never needs to use more than  $O(n\sqrt{v})$  edges. This result suggests that we should be able to restrict attention to these "important" edges, thereby effecting a replacement of  $m$  by  $O(n\sqrt{v})$  in the time bound of any flow algorithm. For example, our  $\tilde{O}(m + nv^{5/4})$  time bound is achieved by applying this substitution to Karger's  $\tilde{O}(v\sqrt{mn})$ -time algorithm. Unfortunately, we do not know how to identify the right  $O(n\sqrt{v})$  edges without finding a flow. Nevertheless, we devise methods to achieve all or part of this speed-up on undirected graphs. It is interesting to compare our result to the sparsification results of Nagamochi and Ibaraki [NI92b]. They give a linear time algorithm that finds a set of  $nv$  edges that are sufficient for a flow of value  $v$ . Their result does allow one to substitute  $nv$  for  $m$  in the time of any flow algorithm, but obviously  $nv$  is substantially larger than  $n\sqrt{v}$ .

Note that Galil and Yu [GY95] previously proved that flows need only use  $O(n\sqrt{v})$  edges on simple graphs, but they did not show how to exploit that fact. Their proof was also somewhat complex. Henzinger, Kleinberg and Rao [HKR97] independently simplified the proofs of Galil and Yu, using an argument similar to but weaker than the one we use. Our stronger result shows that *any* acyclic flow uses few edges, even on *capacitated* graphs.

In order to clearly show which algorithms have the best performance for different values

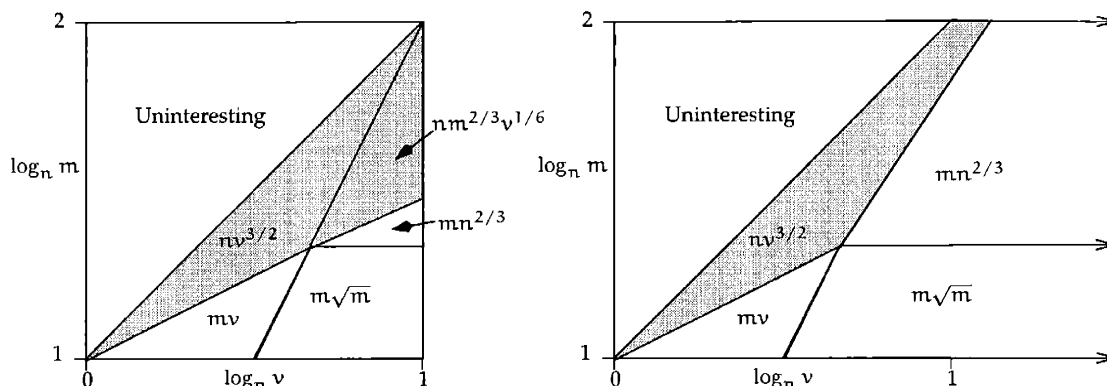


Figure 1.1: Picture of the best deterministic bounds, for simple graphs on the left and for general graphs on the right

of  $m$  and  $v$  relative to  $n$ , we have drawn the pictures in Figure 1.1. A point in the picture represents the value of  $m$  and  $v$  relative to  $n$ . Specifically,  $(a, b)$  represents  $v = n^a, m = n^b$ . Each region is labeled by the best time bound that applies for values of  $m$  and  $v$  in that region. Note that the region  $m > nv$  is uninteresting, because, as mentioned above, the sparsification algorithm of Nagamochi and Ibaraki can always be used to make  $m \leq nv$  in  $O(m)$  time (See Section 2.4 for further explanation). The shaded regions correspond to algorithms given in this thesis. Note that some time bounds only apply to simple graphs, so we have made two pictures: one for simple graphs, one for general graphs. The complexity of these diagrams suggests that more progress can be made.

### Sampling in Residual Graphs

The main result of Chapter 4 is a randomized algorithm that runs in  $\tilde{O}(m + nv)$  time. All of the recent papers on flows in undirected graphs (including Chapter 3) make some attempt to avoid looking at all of the edges all of the time, so as to reduce the amortized time per augmenting path to  $o(m)$ . Our work closes a chapter in this research, finally achieving amortized time per augmenting path of  $\tilde{O}(n)$ . Since a flow path can have as many as  $n - 1$  edges,  $\tilde{O}(n)$  is the result (up to logarithmic factors) that one would hope to achieve.

Our key advance is taking random samples of the edges of residual graphs. Benczúr and Karger [BK96] showed that sampling each edge of an undirected graph with probability inversely proportional to a quantity called strength yields a connected graph with only  $O(n \log n)$  edges with high probability. This sampling would be a good way to find a first augmenting path quickly, but once you have a non-zero flow the residual graph is directed, so it is no longer helpful to know that you can sample from an undirected graph. We extend their technique so that sampling in a residual graph will work. One way to interpret our result is as a proof that a residual graph remains similar to the original undirected graph, which means that the Benczúr-Karger sampling can be applied iteratively. With the exception of computing the edge strengths at the beginning, our algorithm is just this simple iteration: sample according to strengths and find an augmenting path, repeating until done.

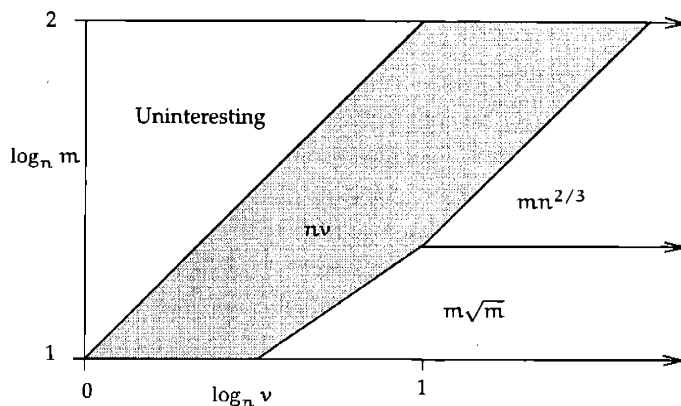


Figure 1.2: Pictures of the best randomized bounds

It is interesting to compare this sampling result to our other maximum flow result. At first glance, using Benczúr-Karger sampling to find a single path seems foolish—the resulting edge set is larger than that of a spanning forest, and extra effort is required to compute suitable sampling weights. But whereas finding a spanning forest in the residual graph by ignoring edge directions would not capture the path information we need, sampling a residual graph by ignoring edge directions does. Our first result works with an obvious simple structure, a spanning forest, and deals with the problem of directed edges by treating them separately. This approach turns out to be worse than starting with what looks like a sloppier structure, a random sample of edges, because a random sample is better able to accommodate directed edges.

In order to show which algorithms have the best performance for different values of  $m$  and  $v$  relative to  $n$ , we have drawn Figure 1.2. The interpretation of this figure is the same as for Figure 1.1. Observe that this diagram is much simpler, suggesting that our understanding of what can be done with randomized algorithms is better than our understanding of what can be done with deterministic ones.

## 1.2 The Minimum $k$ -Way Cut Problem

A *minimum  $k$ -way cut* of an  $n$ -vertex,  $m$ -edge, weighted, undirected graph is a partition of the vertices into  $k$  sets that minimizes the total weight of edges with endpoints in different sets. Asking for a minimum  $k$ -way cut is equivalent to asking for the edge set of minimum total weight whose removal would break the graph into at least  $k$  connected components.

### 1.2.1 History

Our main motivations for studying minimum  $k$ -way cuts are that they are a natural property of graphs that has received considerable attention in the past. Nagamochi and Ibaraki [NI99] also point to a number of applications, including finding cutting planes for the traveling salesman problem, VLSI design, task allocation in distributed computing and network

reliability.

Goldschmidt and Hochbaum [GH94] showed that the minimum  $k$ -way cut problem is NP-hard if  $k$  is part of the input, but that it can be solved in polynomial time if  $k$  is fixed. In particular, they gave a deterministic algorithm with a running time of  $O(n^{k^2/2-3k/2+4}F(n, m))$ , where  $F(n, m)$  is the time to compute a minimum  $s$ - $t$  cut (the minimum 2-way cut that separates specified vertices  $s$  and  $t$ ), which can be solved in  $\tilde{O}(mn)$  time by a maximum flow computation. Karger and Stein [KS96] improved on this algorithm with a randomized one that runs in  $O(n^{2(k-1)} \log^3 n)$  time. These two results are still the best known deterministic and randomized algorithms that work for general  $k^3$ .

Better results are known for several special cases. The minimum 2-way cut problem—more commonly known as simply the minimum cut problem, or the problem of computing edge-connectivity—can be solved in  $\tilde{O}(mn)$  time by the deterministic algorithms of Hao and Orlin [HO94] or Nagamochi and Ibaraki [NI92a] and in  $\tilde{O}(n^2)$  or  $\tilde{O}(m)$  time by the randomized algorithms of Karger and Stein [KS96] and Karger [Kar00], respectively.

Improvements were made for the minimum 3-way cut problem as well. Hochbaum and Shmoys [HS85] showed that on an unweighted planar graph the problem can be solved in  $O(n^2)$  time. Kapoor [Kap96] and Kamidoi, Wakabayashi and Yoshida [KWY97a] showed that the problem can be solved in  $O(n^3F(n, m))$  time. Burlet and Goldschmidt [BG97] improved the bound to  $O(mn^3)$  time.

For the minimum 4-way cut problem, Kamidoi, Wakabayashi and Yoshida [KWY97a] gave an  $O(n^4F(n, m))$ -time algorithm. More recently, Nagamochi and Ibaraki [NI99] gave an  $\tilde{O}(mn^k)$ -time algorithm for  $k \leq 4$ , and Nagamochi, Katayama and Ibaraki [NKI99] extended this result to work for  $k \leq 6$ .

Note that the special cases of small  $k$  are relevant to approximation of larger  $k$ . Saran and Vazirani [SV95] were the first to give an approximation algorithm for the general  $k$ -way cut problem. Their algorithm achieves an approximation ratio of  $(2 - 2/k)$  and runs in  $O(nF(n, m))$  time. Zhao, Nagamochi and Ibaraki [ZNI01] show that it is possible to guarantee a slightly better approximation ratio with an algorithm that uses  $k/2$  calls to a minimum 3-way cut algorithm. The approximation ratio is  $2 - 3/k$  for odd  $k$  and  $2 - (3k - 4)/(k^2 - k)$  for even  $k$ .

## 1.2.2 Our Contribution

Our results, which improve on Nagamochi, Katayama and Ibaraki's results by a factor of  $\tilde{\Theta}(n^2)$ , are very much grounded in past results. In particular, the 3-way cut algorithm makes use of Burlet and Goldschmidt's structural result, Karger's minimum 2-way cut algorithm and Nagamochi and Ibaraki's structural result and algorithm framework. The high level view is that Burlet and Goldschmidt gave a way to break the computation up into a number of subproblems, which could be identified quickly, but took some time to solve. Nagamochi and Ibaraki took a different approach that created far fewer subproblems, but the subproblems

<sup>3</sup>At least, they are the best ones that are fully written up. Nagamochi and Ibaraki [NKI99] say "a deterministic  $O(n^{2k-3}m)$  time algorithm is claimed in [[KWY97b]] (where no full proof is given)."

were much harder to identify. Our algorithm is a hybrid that manages to keep only the fast part of each algorithm.

Our 4-way cut algorithm is similar to the 3-way cut one, except that we needed to find an appropriate extension of Burlet and Goldschmidt's results to 4-way cuts, and we ended up needing to use Karger and Stein's minimum 2-way cut algorithm instead of Karger's. The improvement for 5-way and 6-way cuts follows simply by plugging the result for 4-way cuts into the algorithm of Nagamochi, Katayama and Ibaraki.





## Chapter 2

# Flow Background

In this chapter we review some background information for the maximum flow problem. The first section is largely basic definitions and notation. The remainder of the sections cover the context in which we see our work. In other words, they review the prior concepts that we use. Readers who are familiar with these results should not need to read these sections in order to understand later chapters, although seeing what history we consider relevant probably helps clarify the way in which we look at the problem. In the rest of the thesis we provide pointers back to sections of this chapter when relevant, to make it easy for readers to use this chapter only as reference.

We organize the sections in roughly chronological order, starting with the definition of flow and then discussing augmenting paths, shortest augmenting paths, blocking flow, sparse connectivity certificates and random sampling. So readers who are not very familiar with the maximum flow problem should be able to get a reasonable sense of the sequence of results that has lead to this thesis. Note, however, that our history of flow is heavily biased toward unit-capacity graphs. In particular, we skip major results in the development of flow algorithms for capacitated graphs, such as dynamic trees [ST83] and the push-relabel method [GT88]. Readers who want a more complete survey of maximum flow results should refer to a textbook (*e.g.* [AMO93]).

### 2.1 Maximum Flow Fundamentals

The following is a summary of the notation we use when discussing the maximum flow problem.

$G = (V, E, u)$  refers to a directed graph, where  $u(x, y)$  is an integer representing the capacity of edge  $(x, y)$ . Note that many authors define the capacity to be a real number, but we will only consider integer capacities. We will always assume that either both or neither of  $(x, y)$  and  $(y, x)$  are in  $E$ . If one has a graph that contains  $(x, y)$  but not  $(y, x)$ , the edge  $(y, x)$  can be added with  $u(y, x) = 0$  without changing the set of feasible flows. This addition at most doubles the number of edges, which will have no effect on our asymptotic running time bounds. When we talk about a graph being undirected, we mean that  $u(x, y) = u(y, x)$  for all

edges.

The source is  $s$  and the sink is  $t$ . We use  $n$  as shorthand for  $|V|$ , the number of nodes, and  $m$  as shorthand for  $|E|$ , the number of edges.

A *flow* is an assignment to the edges  $f$ , such that

$$\begin{aligned} \forall(x, y) \in E & \quad f(x, y) \geq 0 && \text{(non-negativity constraint)} \\ \forall(x, y) \in E & \quad f(x, y) \leq u(x, y) && \text{(capacity constraint)} \\ \forall x \in V - \{s, t\} & \quad \sum_{(x, y) \in E} f(x, y) = \sum_{(y, x) \in E} f(y, x) && \text{(conservation constraint)} \end{aligned}$$

The *value* of a flow  $f$ , denoted  $|f|$ , is the net flow leaving the source:

$$\sum_{(s, y) \in E} f(s, y) - \sum_{(y, s) \in E} f(y, s)$$

Because of the conservation constraint, the value is always the same as the net flow entering the sink.

Given  $G$ ,  $s$  and  $t$ , the *maximum flow problem* is to find a flow  $f$  of maximum value over all possible flows. We denote the value of the maximum flow as  $v$ .

An important related concept (see next section) is a *cut*, a non-empty proper subset of the vertices. In an undirected graph with edge capacities, the *value* of a cut  $X$ , written  $c(X)$ , is the total capacity of edges with precisely one endpoint in  $X$ . In a directed graph, the *value* of a cut is the total capacity of edges leaving  $X$ . A *minimum cut* of a graph is a cut of minimum value. An  $\alpha$ -*minimum cut* is a cut whose value is at most  $\alpha$  times the minimum cut value. A graph is said to be *c-edge connected* if it has minimum cut value at least  $c$ . An  $s$ - $t$  *cut* is a cut that contains  $s$  but does not contain  $t$ .

Note that many authors define flow slightly differently. Instead of requiring that flow be non-negative, they require that it be anti-symmetric ( $f(x, y) = -f(y, x)$ ). The choice between the definitions is largely arbitrary. We chose non-negativity because we prefer to think of an edge having non-zero flow as specifically meaning that the flow is on that edge, rather than on the reverse edge.

It is sometimes convenient to allow  $E$  to be a multiset; that is, to allow there to be parallel edges. In general, it is undesirable to work with a graph that has parallel edges because doing so just makes  $m$  larger. Furthermore, in the context of maximum flow, restricting a capacitated graph to have no parallel edges is no restriction at all, because in time linear in the input we can merge parallel edges into one edge with capacity equal to the sum of the capacities of the edges that make it up, and at the end we can split the flow on such an edge among the edges that make it up. So we will always want our input graphs to be free of parallel edges. Nevertheless, in certain instances we will define concepts in a unit-capacity graph (a graph in which all edges have capacity 1), in which case we may want to think about capacities as parallel edges, even though we would never want to actually convert our input graphs in an implementation.

## 2.2 Residual Graphs and Augmenting Paths

An obvious way to try to find a maximum flow is by using a greedy algorithm: find a path from source to sink, delete the capacity used up, and repeat until there are no more source-sink paths. Unfortunately, this algorithm does not work. For example, in Figure 2.1, the flow (thick lines) blocks all source-sink paths but is not maximum.

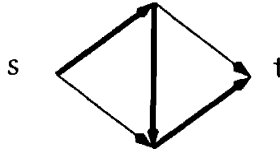


Figure 2.1: A flow that blocks all  $s$ - $t$  paths but is not maximum

Fortunately, Ford and Fulkerson [FF56] showed that a very similar algorithm does work. They defined a *residual graph* of  $G$  with respect to  $f$ :  $G_f = (V, E, u_f)$ , where  $u_f(x, y) = u(x, y) - f(x, y) + f(y, x)$ . Observe that the residual graph not only decreases the capacity of an edge that has flow on it, but increases the capacity of an edge whose reverse has flow on it. The point is that even if  $u(x, y) = 0$ , if  $f(y, x)$  is positive, we can think about sending flow on  $(x, y)$ , because what we will really do is remove flow from  $(y, x)$ . Accordingly, Ford and Fulkerson called a path in the residual graph an *augmenting path*, because we can effectively send more flow on the path, even if we will not actually increase the flow on each edge (because we might decrease the flow on a reverse edge instead).

Since an augmenting path can easily be found by depth-first search or breadth-first search in the residual graph in  $O(m)$  time, the running time of the augmenting paths algorithm is  $O(mv)$ .

Ford and Fulkerson proved several important fundamental theorems about flows:

**Theorem 2.2.1 (Flow Decomposition)** *A flow can always be decomposed into at most  $m$  cycles and source-sink paths.*

This theorem says that it is always reasonable to think about flow as a collection of source-sink paths, although one must keep in mind that there might be cycles. In Chapter 3 we will be interested in how many edges a flow uses, and as such will want to avoid cycles, because they use edges but do not contribute to the value.

Thinking of a flow as paths and cycles, one can see the effect of flow on cut values in the residual graph. A cycle can have no effect on a cut value in the residual graph, because the same amount of flow crosses the cut in each direction. An  $s$ - $t$  path decreases the value of an  $s$ - $t$  cut by precisely the amount of flow on the path, because the number of times the path can cross from  $s$  side to  $t$  side minus the number of times it can cross from  $t$  side to  $s$  side is always precisely 1. So the maximum flow value can never be larger than the minimum  $s$ - $t$  cut value. Ford and Fulkerson proved something even stronger, that the maximum flow value always equals the minimum  $s$ - $t$  cut value:

**Theorem 2.2.2 (max-flow min-cut)** *The following are equivalent*

1.  $f$  is a maximum flow in  $G$
2. there are no augmenting paths in  $G_f$
3.  $|f| = c(X)$  for some  $s$ - $t$  cut  $X$

It is worth pointing out that since the maximum flow value can never be larger than the minimum  $s$ - $t$  cut value, the following corollary follows immediately:

**Corollary 2.2.3** *The value of a maximum flow always equals the value of an  $s$ - $t$  minimum cut.*

Going back to the max-flow min-cut theorem, the equivalence of 1 and 2 demonstrates the correctness of the augmenting paths algorithm. But the equivalence of 2 and 3 will be more relevant for us: as long as every  $s$ - $t$  cut has some capacity left there will be an augmenting path. Our algorithms want to eliminate edges from consideration in order to speed the process of finding augmenting paths. The concern will always be that we might eliminate too many edges, in particular, all residual edges from some  $s$ - $t$  cut, thereby failing to find some flow. So our proofs of correctness will work by showing that until we are done we always have capacity on every  $s$ - $t$  cut.

## 2.3 Shortest Augmenting Paths, Blocking Flow and Simple Graphs

Edmonds and Karp [EK72] gave the first strongly polynomial flow algorithm by analyzing an augmenting paths algorithm where one always augments on the shortest available path (shortest here means having the fewest edges). They show that only  $O(nm)$  paths are required, so their algorithm runs in  $O(nm^2)$  time.

Dinitz [Din70] introduced the idea of a blocking flow. In full generality, a blocking flow is a flow found by the greedy algorithm, as discussed at the beginning of the previous section. It is not known how to compute blocking flows quickly in general, but Dinitz showed that they could be computed in  $O(mn)$  time on an acyclic graph (or  $O(m)$  time if all edges have unit capacity). Further, he showed that if one takes as an acyclic graph all of the edges on shortest paths from source to sink, then the result of adding a blocking flow is to increase the source sink distance by at least 1 in the residual graph. Since the source and sink can only be  $n$  apart, this means that  $n$  iterations of finding a blocking flow suffices to find a maximum flow. Observe that this method gives an improved running time in general of  $O(n^2m)$ , or  $O(nm)$  on a unit-capacity graph.

Even and Tarjan [ET75] and Karzanov [Kar73] analyzed Dinitz's algorithm on simple graphs (unit capacity with no parallel edges). Their first argument is that since there are only  $m$  edges, when the source-sink distance in the residual graph is  $d$ , each path uses at least  $d$  edges so only  $m/d$  paths can possibly remain. Therefore, after  $\sqrt{m}$  blocking flows, only  $\sqrt{m}$  flow remains to be found. Since each blocking flow certainly finds at least one augmenting path, a further  $\sqrt{m}$  blocking flows will complete the computation. So the total time is  $O(m^{3/2})$ .

Their second argument is somewhat more complicated. Let  $V_i$  be the set of nodes at distance  $i$  from the sink. So  $t$  is in  $V_0$  and  $s$  is in  $V_d$ . Therefore the cut separating  $\cup_{j>i} V_j$  (nodes at distance more than  $i$ ) from  $\cup_{j\leq i} V_j$  (nodes at distance at most  $i$ ) is an  $s$ - $t$  cut for any  $i < d$ . A residual edge leaving a node in  $V_{i+1}$  cannot connect to a node at distance less than  $i$ , because that edge would be the first on a path from the starting node to the sink that was shorter than  $i + 1$ . Since we do not allow parallel edges, and residual edges can have capacity at most 2, the total capacity of edges crossing the so-called canonical cut separating  $V_{i+1}$  from  $V_i$  is at most  $2|V_{i+1}||V_i|$ .

Now consider the  $V_i$  in pairs:  $V_0 \cup V_1, V_2 \cup V_3, \dots$ . There are  $\lfloor (d + 1)/2 \rfloor$  such pairs, and they are vertex disjoint, so some pair has at most  $2n/d$  vertices in it. The canonical cut separating this pair has capacity at most  $\max_x 2(x)(2n/d - x) = 2(n/d)^2$ , so the maximum flow value in  $G$  is at most  $2(n/d)^2$ .

We can now apply the same analysis as we did with  $m/d$  paths remaining. After  $n^{2/3}$  blocking flows, only  $2n^{2/3}$  paths remain, so the total time is  $O(mn^{2/3})$ . We will give a related algorithm in Chapter 3 that runs blocking flow for a while and then uses a new method to find augmenting paths at the end, thereby improving on the time bound given here.

Even and Tarjan also observed that as a corollary, one can bound the total length of the augmenting paths. If one runs shortest augmenting paths, then when  $x$  flow remains, the length of the path can be at most  $m/x$  or  $n\sqrt{2}/\sqrt{x}$ . Adding this quantity up as the remaining flow goes from  $v$  down to 1, we see that the total length of the augmenting paths is at most  $O(m \log v)$  (from first analysis) or  $O(n\sqrt{v})$  (from second analysis). We will use this result in Chapter 3 to show that acyclic flows do not use many edges.

## 2.4 Undirected Graph Sparsification

If one is interested in connectivity information in an undirected graph, it is sufficient to consider a spanning forest. That is, two nodes will have a path between them in the spanning forest if and only if they have a path between them in the original graph. Since a spanning forest can have at most  $n - 1$  edges, whereas the original graph could have as many as  $n(n - 1)/2$ , it could easily be advantageous to work with a spanning forest.

Likewise, if one is interested in  $k$ -edge connectivity in an undirected unit-capacity graph, one would prefer to work with a sparse  $k$ -certificate.

**Definition 2.4.1** *A sparse  $k$ -certificate of an undirected graph  $G$  is a subgraph  $G_k$  such that the value of any cut is at least the smaller of  $k$  and the value of the cut in  $G$ .*

It is possible to construct a sparse  $k$ -certificate by taking a spanning forest, then taking another spanning forest on the remaining edges, and so on,  $k$  times. This process yields at most  $nk$  edges. Of course it might take a substantial amount of time to find all those spanning forests.

Nagamochi and Ibaraki [NI92b] gave an algorithm that finds all such forests in one linear time pass. That is, in  $O(m)$  time they assign to each edge a label, where having label  $i$  means

that the edge belongs to the  $i^{\text{th}}$  spanning forest. So a sparse  $k$ -certificate is easily obtained by taking every edge with label at most  $k$ . We refer to the labeling as a *sparse connectivity certificate*, since it contains sparse  $k$ -certificates for all  $k$ .

If the edges have capacities, one would like to think about an edge of capacity  $x$  as  $x$  parallel unit-capacity edges, but  $O(m)$  time in such a graph would not be fast. In a later paper, Nagamochi and Ibaraki [NI92a] show how to handle capacities, assigning each edge two labels, representing the first and last forest of which the edge is a part. If the capacities are  $O(\log n)$  bit integers, their algorithm can be implemented to run in  $O(m)$  time. For general capacities, the running time is  $O(m + n \log n)$ .

This sparsification algorithm can easily be applied to flows. If the maximum flow value is  $v$ , it is sufficient to use a sparse  $v$ -certificate. So in  $O(m)$  time, we can restrict attention to  $O(nv)$  edges, and effectively substitute  $nv$  for  $m$  in the running time of any flow algorithm. In particular, sparsification plus augmenting paths is an  $O(m + nv^2)$ -time algorithm.

Goldberg and Rao's [GR97b]  $O(n\sqrt{mn})$ -time algorithm for maximum flow in simple, undirected graphs goes a bit further. They use blocking flows to compute flow. As the source-sink distance increases, the Even-Tarjan results (described in the previous section) give bounds on how much flow could possibly remain. So after each iteration, they take a  $2(n/d)^2$ -sparse certificate of the unused (and therefore undirected) edges, and throw out any remaining undirected edges, because they will never be needed. Sparsifying again and again gets the better time bound. We go even further in Chapter 3, only keeping a few spanning forests of undirected edges at any time, and bringing more back in when we need them.

## 2.5 Random Sampling in Undirected Graphs

The results on random sampling in undirected graphs can be summarized fairly easily: random sampling does a good job of finding sparse subgraphs that preserve connectivity information. As such, random samples can sometimes be used as sparse certificates. In Chapter 4, we improve our sparse certificate based results of Chapter 3 by doing exactly that.

But random samples can also be more powerful, in that they can find sparse subgraphs that represent the original cut structure fairly well. To see what we mean, compare to a Nagamochi-Ibaraki sparse  $k$ -certificate. Their structure is sparse, preserves all information about cuts smaller than  $k$  exactly and destroys information about cuts larger than  $k$ . A random sample can be sparse and approximately preserve cut values of all cuts. So one trades exactness for better information about larger cuts.

The basic result is as follows: if the edges of an undirected graph are sampled such that for every cut the expected number of edges chosen is at least a certain constant times  $\log n$ , then the sample will be connected with high probability [Kar99]. The simplest version of this result considers a unit-capacity graph and samples uniformly. The minimum cut will clearly have the minimum expected value in the sample, so for minimum cut  $c$  the acceptable probability is  $\Omega(\log n/c)$ . Benczúr and Karger [BK96] observe that if there are vertex induced subgraphs with higher edge connectivity, then edges in them can be sampled with smaller probability. That motivates the following definition.

**Definition 2.5.1** [BK96] *The strength of an edge  $\{x, y\}$ , denoted  $k(x, y)$ , is the maximum value of  $k$  such that a  $k$ -edge connected vertex-induced subgraph of  $G$  contains  $\{x, y\}$ . We say an edge is  $k$ -strong if its strength is  $k$  or more, and  $k$ -weak otherwise.*

In a unit-capacity graph, sampling each edge with probability proportional to  $\log n/k(x, y)$  causes the minimum expected number of edges in a cut to be  $\Omega(\log n)$ . To see why, consider any cut and its highest strength edge, of strength  $k$ . By the definition of strength there must in fact be at least  $k$  edges of strength  $k$  in the cut (because strength  $k$  came from a  $k$ -edge connected vertex induced subgraph), so the expected number of edges chosen of that strength alone is  $\Omega(\log n)$ .

What makes strength more important, though, is the fact that  $\sum_{\{x,y\} \in E} 1/k(x, y) \leq n$  [BK96]. So with high probability, sampling each edge with probability proportional to  $\log n/k(x, y)$  gives a sample of size  $O(n \log n)$  that is connected. The only problem is that it is not known how to compute strengths quickly. Worse, it is not known how to find constant factor approximations quickly. So instead Benczúr and Karger introduce and show how to compute what we will call *modified strengths*, denoted as  $k'$ . Modified strengths are lower bounds on strengths that are good on average. That is, they are always lower bounds, and the sum of inverses is still small,  $4n$ . So using them in place of strengths only improves the probability that the resulting sample will be connected, and the sample will still be small.

In Chapter 4, we do sampling in a residual graph. Our approach is similar to that of Benczúr and Karger and we manage to achieve similar results, even though we sample a directed graph.

As mentioned above, beyond just being connected, it is possible to show that cut values stay close to their expected values:

**Theorem 2.5.2** [Kar99] *If  $G$  is  $c$ -edge connected and edges are sampled with probability  $p$ , then with high probability all cuts in the sampled graph are within  $(1 \pm \sqrt{8 \ln n / pc})$  of their expected values.*

Of course if sampling probabilities are non-uniform, as in the case of using strengths, then it is of less interest to talk about the expected values of cuts, because they are very different from original values. In the specific case of strength, what Benczúr and Karger do is sample with probability proportional to  $\log n/k(x, y)$ , and increase the capacity of chosen edges by one over the sampling probability. That way the expected value of every edge is its original capacity. They call this technique *compression*, because they get a new graph that has fewer edges but similar cuts. In particular, they show that with appropriate constants, one will get (with high probability) a graph that has the same cut values to within a  $1 \pm \epsilon$  factor, but only  $O(n \log n / \epsilon^2)$  edges.

Note that for the purposes of our algorithm in Chapter 4, we only need to show that connectivity is preserved. It is also possible to prove a compression theorem for residual graphs, although we do not know what to do with it. Accordingly, the proof of compression is in Appendix B.





## Chapter 3

# Spanning Forests of Undirected Edges

In this chapter we show how to find augmenting paths quickly when finding flow in an undirected graph by reducing the undirected edges of the residual graph down to a spanning forest. We then show how to apply this technique to get faster maximum flow algorithms for undirected graphs. A preliminary version of this work appeared in a conference paper [KL98].

The basic idea is easy to state: use a spanning forest on the edges that are still undirected (for which capacities are still symmetric). That is, when we go to find an augmenting path, we search in the directed edges plus a spanning forest on the undirected edges. Since spanning forests capture the connectivity information of undirected graphs, it should make sense that this method is correct. And clearly whenever there are more than  $n - 1$  undirected edges it will reduce the size of the graph in which we search for an augmenting path.

The obvious potential problem is that all the edges might become directed, such that we get no benefit. We will begin by proving that this problem can be avoided, and then give details of a complete algorithm for finding augmenting paths that is conceptually simple but has some excess logarithmic factors in the running time. We then give details of a more complicated method that avoids the logarithmic factors, and finish the chapter by discussing complete flow algorithms.

Since we want to treat undirected edges differently from directed edges, some additional notation is useful. For a flow problem  $G = (V, E, u)$ , we use  $E^u$  to denote the “undirected edges”,  $\{(x, y) : u(x, y) = u(y, x)\}$ , and  $E^d$  to denote the “directed edges”,  $\{(x, y) : u(x, y) \neq u(y, x)\}$ .

The point of these definitions is that an undirected graph  $G$  and a non-zero flow  $f$  will give rise to a residual graph  $G_f$  in which the edge capacities are no longer symmetric. That is,  $E_f^d$  will always be non-empty. So we cannot hope to exploit undirected graph properties in  $G_f$ ; we can, however, exploit undirected graph properties in  $E_f^u$ .

We will want to talk about running algorithms defined only for undirected graphs on  $E^u$ . Whenever we say to run such an algorithm on  $E^u$ , we mean that it should be run on an undirected graph with the same vertices and the edge set  $\{(x, y) : (x, y) \in E^u\}$ . If the algorithm expects capacities, then  $u(\{x, y\}) = u(x, y)$ , which of course is the same as  $u(y, x)$ . When we

talk about using the output edge set  $E'$  from such an algorithm in the context of flow, we mean  $\{(x, y) : \{x, y\} \in E'\}$ . So in general this means that the number of edges we use in our flow context is double the number of undirected edges. For example, a spanning tree of an undirected graph has at most  $n - 1$  edges, so in our flow context when we talk about using a spanning tree, there will be  $2(n - 1)$  directed edges.

### 3.1 Small Flows Use Few Edges

Separating directed edges and undirected edges and exploiting the good properties of undirected graphs when possible would be worthless if all of the edges were liable to become directed. As such the bound on the number of directed edges that we give in this section is the foundation for the rest of the chapter. With one definition, the theorem is easy to state:

**Definition 3.1.1** *A flow  $f$  is acyclic if there is no directed cycle on which every edge has positive flow in the cycle direction.*

**Theorem 3.1.2** *Any integral acyclic flow  $f$  uses at most  $2n\sqrt{|f|}$  edges.*

Note that this theorem is very close to a theorem proved by Galil and Yu [GY95] and simplified by Henzinger, Kleinberg and Rao [HKR97] that says there exists a maximum flow that uses only  $O(n\sqrt{v})$  edges in simple graphs. Our result is stronger in that we show that any integral acyclic flow uses few edges, even on capacitated graphs. We also have tighter constants. Our proof is a stronger version of the one used by Henzinger, Kleinberg and Rao, although we proved it independently.

The proof given by Henzinger, Kleinberg and Rao is easy to state, given the results proved by Even and Tarjan [ET75] about blocking flows in simple graphs (see Section 2.3). Even and Tarjan showed that the total length of augmenting paths in the blocking flow algorithm is only  $O(n\sqrt{v})$ . Since the total length of augmenting paths is an upper bound on the number of edges used, it is clear that there always exists a maximum flow that uses only  $O(n\sqrt{v})$  edges.

We refine this proof in two ways. We show that it applies equally well when there are edge capacities, and we show that *any* acyclic flow uses few edges, not just that a flow using few edges exists.

In order to account for capacities, we consider a slightly different algorithm. Recall that the Even-Tarjan argument is based on shortest augmenting paths. We would like to avoid over-counting edges that have high capacity and consequently are in many augmenting paths. So what we will do is define a length function on the edges and consider finding shortest augmenting paths with respect to the lengths. The length of unit capacity edges will be 1, and the length of any higher capacity edges will be 0. The idea is that an edge will only contribute to the length of an augmenting path the last time it is used.

As an aside, observe that while it might sound like the combination of this idea and blocking flows would make for a fast flow algorithm for capacitated graphs, it does not. The problem is that when zero length edges are present, the set of edges that are on shortest paths

is not necessarily acyclic, and we only know how to find blocking flows quickly on acyclic graphs. Note that the Goldberg-Rao algorithm that achieves Even-Tarjan like time bounds on capacitated graphs does use a 0-1 length function, but in a more complicated way. For our proof we only care about the bound on the total length of augmenting paths, not the running time of the algorithm, so we did not need to do anything as complicated as Goldberg-Rao.

The extended version of the Even-Tarjan canonical cut argument is as follows:

**Lemma 3.1.3** *Given a graph  $G = (V, E, u)$  with no parallel edges, let  $l_u(x, y)$  be 0 if  $u(x, y) > 1$ , 1 if  $u(x, y) \leq 1$ , and  $\infty$  if  $u(x, y) = 0$ . Let  $d_u(x)$  be the distance with respect to  $l_u$  from node  $x$  to  $t$ , or  $\infty$  if there is no path from  $x$  to  $t$ . If  $0 < d_u(s) < \infty$ , then  $v < (n/d_u(s))^2$ .*

**Proof.** (Note that this proof is really only different from the Even-Tarjan proof in that it is stated with the length function.)

Let  $V_i$  be the set of nodes with  $d_u(x) = i$ . Since  $s$  is in  $V_{d_u(s)}$  and the sink is in  $V_0$ , the cut separating  $\cup_{j>i} V_j$  from  $\cup_{j\leq i} V_j$  is an  $s$ - $t$  cut for any  $i < d_u(s)$ . Call this cut the *canonical cut* separating  $V_{i+1}$  from  $V_i$ . Observe that edges with positive capacity leaving a node in  $V_{i+1}$  can never go to a node at distance  $i$  or less, and cannot even go to a node at distance  $i$  if they have length 0 (capacity exceeding 1), or else there would be a path to the sink from the starting node that is shorter than  $i + 1$ . Since only unit capacity edges can cross from  $V_{i+1}$  to  $V_i$  and we do not allow parallel edges, the total capacity of edges crossing the canonical cut separating  $V_{i+1}$  from  $V_i$  is at most  $|V_{i+1}||V_i|$ .

Now consider the  $V_i$  in pairs:  $V_0 \cup V_1, V_2 \cup V_3, \dots$ . There are  $\lfloor (d_u(s) + 1)/2 \rfloor$  such pairs, and they are vertex disjoint, so some pair has at most  $2n/d_u(s)$  vertices in it. The canonical cut separating this pair has capacity at most  $\max_x(x)(2n/d_u(s) - x) = (n/d_u(s))^2$ , so the maximum flow value in  $G$  is at most  $(n/d_u(s))^2$ . ■

Now, the other component we want to add to our final proof is that any acyclic graph uses few edges. We get that extra strength from the following lemma, which says that we can construct a graph in which the unique maximum flow is the acyclic flow we are interested in, so that existence of a maximum flow with few edges in this graph will imply that the original flow uses few edges.

**Lemma 3.1.4** *Let  $f$  be an acyclic flow in  $G = (V, E, u)$ . Then  $f$  is the unique maximum flow in  $G' = (V, E, u')$ , where  $u'(x, y) = f(x, y)$ .*

**Proof.** (See Section 2.2 for relevant background.) First, observe that  $f$  is a flow in  $G'$ , and it uses up the capacity of every edge, so it saturates a minimum cut, so it must be maximum. Now, to show uniqueness, suppose that there exists  $f'$ , a maximum flow in  $G'$  that is not the same as  $f$ . Consider  $f'' = f - f'$ . It is clear that  $f''$  will satisfy capacity and conservation constraints. Since  $f$  uses up every edge,  $f''$  must also satisfy non-negativity. The value of  $f''$  is clearly 0. By flow decomposition,  $f''$  can be decomposed into paths from source to sink and cycles. It must be that  $f''$  has only cycles, because it has no paths from source to sink. But the only edges with non-zero capacity are those on which  $f$  was positive, and those edges do not contain cycles—a contradiction. ■

We can now prove the theorem.

**Proof of Theorem 3.1.2.** Consider  $G' = (V, E, u')$ , where  $u'(x, y) = f(x, y)$ . By Lemma 3.1.4, if we find a maximum flow in this graph, it will be  $f$ . So consider building a flow  $f'$  in  $G'$  by repeatedly finding and augmenting one unit of flow on a shortest path in  $G'_{f'}$ , where shortest is defined by the length function of Lemma 3.1.3 in  $G'_{f'}$ . When no paths are left we will have  $f' = f$ . Lemma 3.1.3 says that when  $d_{u'_f}(s) > 0$ ,  $|f - f'| \leq (n/d_{u'_f}(s))^2$ , which we can restate as  $d_{u'_f}(s) \leq n/\sqrt{|f - f'|}$ . As we find paths,  $|f - f'|$  takes on each value from  $|f|$  to 1. Since we always augment on shortest paths, the length of each path is  $d_{u'_f}(s)$ . Therefore the total length of the paths is at most

$$\sum_{x=1}^{|f|} d_{u'_f}(s) \leq \sum_{x=1}^{|f|} \frac{n}{\sqrt{x}} \leq 2n\sqrt{|f|}$$

Since  $u'$  is integral, and we only reduce the capacity of an edge by 1 unit at a time, and every edge is reduced to 0 capacity at the end, it must be the case that every edge has length 1 at least one of the times it is on an augmenting path, so that edge is counted when we count the length of that path in the sum. It follows that the total length of the augmenting paths is an upper bound on the number of edges used by  $f$ . ■

Observe that Theorem 3.1.2 is tight up to constant factors. Figure 3.1 gives an example of a simple graph with an acyclic maximum flow that uses all  $\Theta(n\sqrt{v})$  edges.

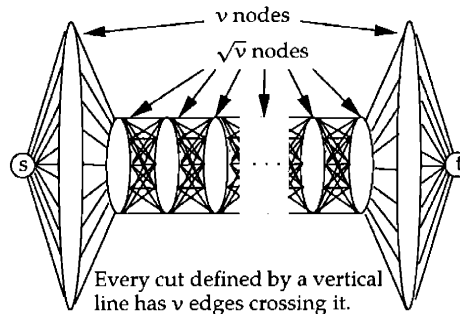


Figure 3.1: A graph with an acyclic flow that uses  $O(n\sqrt{v})$  edges

### 3.2 A Conceptually Easy Algorithm for Fast Augmenting Paths

In this section we give a full algorithm based on the idea of maintaining a spanning forest on the undirected edges. We will maintain the spanning forest using a dynamic connectivity data structure, and we will take steps to prevent flow cycles so that we can apply the bounds of the previous section. This algorithm is conceptually simple, and will find augmenting paths in  $\tilde{O}(n\sqrt{v})$  amortized time per path. The price of the conceptual simplicity will be several logarithmic factors. In the next section we will describe another method that is less conceptually simple (although possibly easier to implement) and avoids logarithmic factors most of the time.

We can get efficient dynamic maintenance of spanning forests by simply applying a data structure built for that purpose:

**Lemma 3.2.1** [HdLT98] *It is possible to maintain a spanning forest of an undirected graph under edge insertions and deletions in  $O(\log^2 n)$  amortized time per operation.*

We also need to worry about keeping our flow acyclic, because Theorem 3.1.2 only applies if it is. Fortunately, using a procedure given by Sleator and Tarjan [ST83], it is easy to remove all cycles from a flow (we will refer to this procedure as *decycling*):

**Lemma 3.2.2** [ST83] *It is possible to take a flow  $f$  that is non-zero on  $x$  edges and find an acyclic flow  $f'$  of the same value ( $|f'| = |f|$ ) in  $O(x \log n)$  time.*

Since our algorithms are only interesting for small flow values, and unit-capacity graphs are a frequently studied case of graphs with small flow values, we observe that a simplification of the Sleator-Tarjan algorithm works a little faster in a unit-capacity graph:

**Lemma 3.2.3** *In a unit-capacity graph, it is possible to take a flow  $f$  that is non-zero on  $x$  edges and find an acyclic flow  $f'$  of the same value ( $|f'| = |f|$ ) in  $O(x)$  time.*

**Proof.** One natural way to remove cycles from a flow would be to repeatedly find a cycle by doing a depth first search on edges carrying flow (in the direction of the flow) and remove it by sending flow backward on the cycle. Removing cycles will not change the flow value. If no cycle is found, then obviously we are done, and we will reach this point because in every iteration we reduce the number of edges carrying flow. However we might end up doing too much work this way, because we have to do many searches. But when we find a cycle, any work done looking at edges that did not reveal cycles need not have been wasted—removing flow will never create flow cycles, so we can just restart the search from the first node of the cycle that we found and only search on edges that were not previously explored. Since we explore each edge at most once and delete each edge at most once, and there are only  $x$  edges that we are interested in, the running time is  $O(x)$ . (Note that the capacitated case solved by Sleator and Tarjan is far more difficult—an edge will not necessarily be deleted when flow is removed from it, so a given edge might need to be explored many times. They solve this problem with an amazing data structure called a dynamic tree.) ■

The basic algorithm for fast augmenting paths appears in Figure 3.2. To show that this algorithm is correct, we just need to know that  $G'$  contains an augmenting path if and only if  $G_f$  does, and then we can apply the max-flow min-cut theorem (see Section 2.2).

**Theorem 3.2.4** *Let  $T$  be any spanning forest of  $E_f^u$ . Then  $T \cup E_f^d$  has an augmenting path if and only if  $G_f$  does.*

**Proof.** Let  $G' = T \cup E_f^d$ . Since  $G'$  is a subgraph of  $G_f$ , it is clear that if  $G'$  has an augmenting path then  $G_f$  does. For the other direction, suppose that there is an augmenting path in  $G_f$ ,

```

SparseAugment1(G, f)
  insert all undirected edges ( $E_f^u$ ) into a dynamic connectivity data structure, and use it
  to maintain a spanning forest  $T$ 
  repeat:
    look for an augmenting path in  $E_f^d \cup T$ 
    if no such path exists
      return f
    else
      augment  $f$  using the path
       $f \leftarrow \text{decycle}(f)$ 
      update the connectivity structure as appropriate to the augmentation:
        delete any edge no longer in  $E_f^u$  (because flow added)
        insert any edge newly a member of  $E_f^u$  (because flow removed)

```

Figure 3.2: Basic algorithm for fast augmenting paths

but not in  $G'$ . By the max-flow min-cut theorem, we can restate this condition as follows: every  $s$ - $t$  cut of  $G_f$  has an edge with positive capacity crossing it (from the  $s$  to the  $t$  side), but some cut of  $G'$  does not. Thus, there is an  $s$ - $t$  cut  $C$  that has a positive capacity edge  $(x, y)$  crossing it in  $G_f$ , but no edges crossing it in  $G'$ . If  $(x, y) \in E_f^d$ , then it is in  $G'$ —a contradiction. So  $(x, y)$  must be in  $E_f^u$ . But  $T$  is a spanning forest of  $E_f^u$ , which means that it contains an edge from every nonempty cut of  $E_f^u$ . (If there were an edge across a cut where  $T$  did not have any edges, then the edge could be added to  $T$ , increasing its span and contradicting the fact that  $T$  is spanning to begin with.) Since  $C$  is nonempty in  $E_f^u$  ( $(x, y)$  crosses it), some edge of  $T$ , and thus of  $G'$ , crosses  $C$ . This contradicts our (restated) original assumption. ■

We now consider the running time:

**Theorem 3.2.5** *SparseAugment1 runs in  $\tilde{O}(m + rn\sqrt{v})$  time on an undirected graph, where  $r = v - |f|$  is the number of augmenting paths that need to be found.*

**Proof.** First consider the work done finding augmenting paths. Since we decycle the flow in each iteration, every augmenting path search takes place in a graph with  $O(n\sqrt{v})$  edges and therefore takes  $O(n\sqrt{v})$  time. Similarly, every decycling takes  $\tilde{O}(n\sqrt{v})$  time. Since there are  $r$  iterations, the total time is  $\tilde{O}(rn\sqrt{v})$ .

It remains to account for the dynamic connectivity operations. First consider deletions. An edge is deleted from the data structure when we place flow on it. This happens to at most  $n$  edges in any one augmenting path, for a total of  $nr$  deletions taking  $\tilde{O}(nr)$  time. Now consider insertions. Initially we insert all edges in the structure in  $\tilde{O}(m)$  time. Later, edges are inserted in the data structure when flow is removed from them. Note, however, that flow cannot be removed from an edge until flow has been added to the edge. We have already counted the cost of deleting edges when we add flow to them; this cost can also absorb the equal cost of inserting those edges when the flow is removed. ■

### 3.3 A Better Algorithm for Fast Augmenting Paths

The algorithm in the previous section spends more time maintaining the edge set in which it will search for augmenting paths than it spends actually searching. One way to reduce this overhead cost is to find several spanning forests at once and use them to find several augmenting paths. In this section we apply an algorithm given by Nagamochi and Ibaraki [NI92b] to do just that (see Section 2.4 for background).

```

SparseAugment2(G, f)
  k ← ⌈√m/n⌉
  if f has cycles, f ← decycle(f)
  repeat:
    Gk ← a sparse k-certificate of Efu
    G' ← Efd ∪ Gk
    run augmenting paths on G' until k paths are found or no more paths exist
    f ← decycle(f)
  if the augmenting paths step found less than k paths, return f

```

Figure 3.3: First algorithm for fast augmenting paths based on sparse certificates

Our first algorithm using sparse certificates appears in Figure 3.3. We terminate when  $G'$  has fewer than  $k$  paths. So to prove that the algorithm is correct we only need to know that  $G'$  always has at least  $k$  paths when  $G_f$  does, and that when  $G_f$  has fewer than  $k$  augmenting paths  $G'$  has the same number.

**Theorem 3.3.1** *Let  $G_k$  be a sparse  $k$ -certificate of  $E_f^u$ . Then  $E_f^d \cup G_k$  contains  $i < k$  augmenting paths if and only if  $G_f$  has  $i$  augmenting paths, and  $E_f^d \cup G_k$  contains at least  $k$  paths if  $G_f$  contains at least  $k$ .*

**Proof.** The idea here is the same as that of Theorem 3.2.4, except that now we have several spanning forests instead of one. Again  $G' = E_f^d \cup G_k$  is a subgraph of  $G_f$ , so it can have no more augmenting paths than  $G_f$ . For the other direction, consider a minimum  $s$ - $t$  cut of  $G'$ . Suppose  $G_f$  has more capacity crossing this cut. It is impossible for the extra capacity to be in  $E_f^d$ , because  $G'$  contains all edges of  $E_f^d$ . So there must be more capacity crossing the cut in  $E_f^u$  than in  $G_k$ . But by definition of a sparse  $k$  certificate, this can only happen if more than  $k$  capacity crosses the cut in  $E_f^u$ , in which case at least  $k$  capacity must cross the cut in  $G_k$ . This completes the proof. ■

We now consider the running time.

**Lemma 3.3.2** *Let  $r$  be the number of augmenting paths that need to be found. The running time of  $\text{SparseAugment2}(G, f)$  on a unit-capacity undirected graph is  $O(m + r(n\sqrt{v} + \sqrt{mn}))$ . With capacities the running time is slowed by an  $O(\log n)$  factor.*

**Proof.** We begin with the unit capacity case. By Lemma 3.2.3, the initial decycling takes  $O(m)$  time. The sparse certificate computation at the beginning of the loop takes  $O(m)$  time per

iteration. The cost of the augmenting paths step is  $O(m'k)$ , where  $m'$  is the number of edges in  $G'$ . The decycling step in the loop takes  $O(m')$  time. By definition of a sparse  $k$ -certificate and Theorem 3.1.2,  $m' \leq 2nk + n\sqrt{v} = 2\sqrt{mn} + n\sqrt{v}$ . The number of iterations is  $\lceil r/k \rceil$ , so the total time is  $O((m + m'k) \lceil r/k \rceil) = O(m + r(n\sqrt{v} + \sqrt{mn}))$ .

With capacities, the only change is that the decycling steps may be slower by an  $O(\log n)$  factor, so the total running time is also slowed by no more than that. ■

This bound is somewhat unsatisfactory, in that the cost per augmenting path becomes  $\sqrt{mn}$  when  $m \geq nv$ . But if we knew  $v$  at the beginning, we could find a sparse  $v$ -certificate and ensure that we only worked with  $nv$  edges for the rest of the algorithm. This would give the amortized  $O(n\sqrt{v})$  time per path that we want. It turns out that we can effectively simulate knowing  $v$  by taking a small guess and doubling it until we are correct. The pseudocode for this improvement appears in Figure 3.4.

```

SparseAugment3(G, f)
  compute a sparse connectivity certificate of unused edges of G
  (we will now use  $G_w$  to denote the first  $w$  forests of this sparse certificate)
   $w \leftarrow |f|$ 
  repeat:
     $w \leftarrow$  minimum  $w'$  such that  $|G_{w'}| > 2|G_w|$ 
    SparseAugment2( $G_w$ , f), stopping when  $|f| \geq w$ 
  until  $|f| < w$ 
  return f

```

Figure 3.4: Final algorithm for fast augmenting paths

Notice that  $G_w \subset G_{2w}$ , so we need not start over each iteration of the loop, but can simply continue with more of the edges from  $G$ . This is irrelevant to the time bound, but it means that in practice work is not being wasted.

**Theorem 3.3.3** *The running time of SparseAugment3( $G, f$ ) on a unit-capacity undirected graph is  $O(m + rn\sqrt{v})$ , where  $r$  is the number of augmenting paths that need to be found. With capacities the running time is slowed by an  $O(\log n)$  factor.*

**Proof.** Again, we begin with the unit capacity case. The running time of the first step is  $O(m)$ . The running time of the  $i^{\text{th}}$  iteration is  $O(m_i + r_i(n\sqrt{v} + \sqrt{m_i n}))$  by Lemma 3.3.2. (Here the notation  $x_i$  is used to mean the value of  $x$  in the  $i^{\text{th}}$  iteration.) Since  $m_i$  doubles with each iteration, the sum over iterations of the first term is  $O(m)$ . Let  $k$  be the number of iterations. It must be the case that  $w_{k-1} \leq v$  in order for the  $(k-1)^{\text{st}}$  iteration to not terminate. Thus  $m_{k-1} \leq 2nv$ . Since we attempt to double  $m_i$ , ending up with at most one tree too many,  $m_k \leq 4nv + 2n = O(nv)$ . Since  $\sum r_i = r$ , the sum over iterations of the second term is  $O(rn\sqrt{v})$ . The total is  $O(m + rn\sqrt{v})$ .

Since almost all the work is done by SparseAugment2, it is clear that the same  $O(\log n)$  slowdown with capacities applies. ■



### 3.4 Deterministic Algorithms Using Fast Augmenting Paths

The results of the previous sections can be used in several ways to give fast flow algorithms. Most obviously, simply using `SparseAugment3` to find all the necessary augmenting paths gives a simple, deterministic  $O((m + nv^{3/2}) \log n)$ -time flow algorithm. On simple graphs the time bound is slightly better:  $O(m + nv^{3/2})$ . In the worst case, when  $m = \Theta(n^2)$  and  $v = \Theta(n)$ , this gives an  $O(n^{5/2})$  time bound, which is as good as the worst-case bounds of all previous known algorithms. For smaller  $v$  this is the best deterministic algorithm known. Note that ours is the first deterministic algorithm to achieve this bound without blocking flows, and unlike previous blocking flow approaches it benefits from small  $v$ . For large  $v$ , we can do better by running blocking flows until not much flow remains and then finishing with fast augmenting paths. (See Section 2.3 for background.)

**Lemma 3.4.1** *On a simple graph, if we run blocking flows  $k$  times and then finish with fast augmenting paths, the running time is  $O(mk + n^3 \sqrt{v}/k^2)$ .*

**Proof.** Finding a blocking flow takes  $O(m)$  time, so computing  $k$  of them takes  $O(mk)$  time. After  $k$  blocking flows on a simple graph the source-sink distance is at least  $k$ , so the remaining flow is  $O((n/k)^2)$ . Therefore the time for fast augmenting paths is  $O(n^3 \sqrt{v}/k^2)$ . ■

It is important to point out that this result is limited to simple graphs because it uses the Even-Tarjan [ET75] argument that when the source-sink distance is at least  $k$ , the remaining flow is only  $O((n/k)^2)$ . Note that we used similar arguments in capacitated graphs to prove that flows use few edges, but we did not show anything that would imply that little flow would be left in a capacitated graph after  $k$  blocking flows.

We can pick  $k = nv^{1/6}/m^{1/3}$  to balance the terms and get an algorithm that runs in  $O(nm^{2/3}v^{1/6})$  time. This algorithm also takes  $O(n^{5/2})$  time in the worst case, but it is better when the graph is sparse but the flow value is large. It is always at least as good as the bound of  $O(n^{3/2}m^{1/2})$  given by Goldberg and Rao [GR97b], and in general better by a factor of  $(n^3/mv)^{1/6}$ .

Note however that, unlike the Even-Tarjan improvement, where the better running time arose by changing the *analysis* of the algorithm to augmenting paths at a certain point, we must explicitly change the *execution* of the algorithm at a certain point to achieve our bounds. The reason for this difference is that we do not know how to combine our sparsification techniques and blocking flows. Obviously we could take  $E_f^d$  plus a sparse  $k$ -certificate and find a blocking flow in it, but the good thing about blocking flow is that it finds many augmenting paths at once, quantified by the fact that it increases the source-sink distance. If we sparsify, we may prevent the blocking flow from finding all the paths it might otherwise find, and then later, when we added a new sparse certificate, we might reduce the source-sink distance, thereby losing track of any benefit we might have obtained.

This issue of changing analysis versus execution is important, because the fact that our algorithm must change its actions means that we need to be able to compute the switchover point. That is, we need to know  $v$  in advance in order to achieve our bound.

There are several ways to avoid this problem. One is to use the iterative doubling trick employed in `SparseAugment3`. Another possibility is to run blocking flows until the source is far enough from the sink. In particular, if  $d_f$  is the smallest number of positive capacity edges in  $E_f$  on a path from source to sink, then we should stop blocking flows when  $md_f \geq \left(\frac{n}{d_f}\right)^2 n\sqrt{|f| + \left(\frac{n}{d_f}\right)^2}$  and then finish with fast augmenting paths. At that point there can be only  $\left(\frac{n}{d_f}\right)^2$  flow remaining, so  $|f| + \left(\frac{n}{d_f}\right)^2$  is an upper bound on  $v$ . Therefore, when the condition is true we have  $md_f \geq \left(\frac{n}{d_f}\right)^2 n\sqrt{v}$ . Solving for  $d_f$ , we get that it has to be at least  $nv^{1/6}/m^{1/3}$ , which means that the remaining flow is at most  $O(m^{2/3}/v^{1/3})$ . Thus the time spent running fast augmenting paths is  $O(n\sqrt{v}m^{2/3}/v^{1/3}) = O(nm^{2/3}v^{1/6})$ .

As for the time spent finding blocking flows, we do one blocking flow computation beyond the point where  $md_f < \left(\frac{n}{d_f}\right)^2 n\sqrt{|f| + \left(\frac{n}{d_f}\right)^2}$ . So it is either the case that  $md_f < \left(\frac{n}{d_f}\right)^2 n\sqrt{2|f|}$  or that  $md_f < \left(\frac{n}{d_f}\right)^2 n\sqrt{2\left(\frac{n}{d_f}\right)^2}$ , depending on which term inside the square root is larger. In the former case, since  $|f| < v$ , we again get that  $d_f = O(nv^{1/6}/m^{1/3})$ . In the latter case, solving for  $d_f$  we get  $d_f < n/m^{1/4}$ . Since  $d_f$  is an upper bound on the number of blocking flows we have to compute, the total time is  $O(nm^{2/3}v^{1/6} + nm^{3/4})$ . This is not quite as good, but  $nm^{2/3}v^{1/6}$  is only better than  $nv^{3/2}$  when  $m < v^2$ , and in this case  $nm^{3/4} = nm^{2/3}m^{1/12} = O(nm^{2/3}v^{1/6})$ . In other words, in the case where we would want to use blocking flows at all (when it is faster than the augmenting paths running time of  $O(m + nv^{3/2})$ ), this method achieves the full benefit.

## Chapter 4

# Sampling in Residual Graphs

In this chapter we present a maximum flow algorithm for undirected graphs based on finding augmenting paths in random samples of edges from the residual graph. A preliminary version of this work appeared in a conference paper [KL02].

The initial idea for the algorithm in this chapter is to use random sampling instead of spanning forests. With high probability, a suitable random sampling scheme will yield a sparse graph that has the same property of being connected. At first glance it might seem that replacing trees with random samples is of little use, because the samples are slightly larger and have a chance of error. But whereas directed edges are a problem for spanning trees, it is easy to consider sampling directed edges. It turns out that this difference allows us to stop separating directed and undirected edges and get a substantially improved algorithm.

We start with the sampling scheme given by Benczúr and Karger [BK96] (See Section 2.5 for background.) The hope is that we can use a sample of size  $O(n \log n)$  for each augmenting path. But of course we need to account for the effect of the flow in the residual graph. To see the basic idea, consider a residual graph when  $|f| = v/2$ . The cuts are the same as in the original graph, but each cut may have lost as much as half of its capacity. If we were sampling with uniform probability, the expected number of edges chosen would be smaller by half, but we could easily compensate by doubling the sampling probability. More generally, we can compensate by increasing the probability by one over the fraction of flow remaining:  $\frac{v}{v-|f|}$ . This argument is insufficient for non-uniform probabilities, because the flow might take away the high probability edges and leave the low probability ones, thereby greatly changing the expected value. For example, if a cut has  $v/2$  edges of strength 1 and  $v/2$  edges of strength  $v/2$ , the initial expected value is roughly  $v/2 \log n$ . If the flow uses the  $v/2$ -strong edges, then doubling the sampling probability just makes the expected value larger—roughly  $v \log n$ . But if the flow uses the strength 1 edges, then even with the doubling the expected value drops to roughly  $2 \log n$ , which is a huge reduction. Notice, however, that the expected value is still at least  $\log n$ , so all hope should not be lost. We show that the method works anyway, essentially because, as mentioned above, the highest strength edges in a cut are sufficient to give a high expected value. So if the flow uses low strength edges we will have enough high strength edges to make the argument work anyway, and if the flow uses low strength edges then with the extra probability the expected values will simply increase.

Having given the idea, it is now time to state the algorithm itself, which is far simpler than the reasons why it works. We will then give the analysis in full detail.

## 4.1 The Algorithm

Our algorithm is easy to state. With the possible exception of the first step, it ought to be correspondingly easy to implement. It is, however, difficult to tell whether it might perform well in practice without actually implementing it.

```

SampleResidual(G)
  compute modified strengths,  $k'$ 
   $\alpha = 1$ 
  while  $\alpha n < m$ 
    sampling according to weights  $u_f(x, y)/k'(x, y)$ , pick a sample of  $\alpha n$  edges
    search for an augmenting path in the sample
    if no path is found, double  $\alpha$ 
  repeatedly search for augmenting paths in the residual graph until no more are found

```

Figure 4.1: Maximum flow algorithm based on sampling the residual graph

The pseudocode appears in Figure 4.1. The correctness of the algorithm is obviously guaranteed by the last step.

One important point about the algorithm is that we are sampling directed edges. The background section discussion of Benczúr-Karger only discusses the idea for undirected unit-capacity graphs, because it is simpler to understand that way. Our algorithm must take into account the directed edges and capacities of the residual graph.

## 4.2 The Analysis

The foundation of our analysis is the following theorem, closely related to the main theorem of Benczúr and Karger [BK96]:

**Theorem 4.2.1** *Let  $\beta = \frac{4v \ln n}{v - |f|}$ . If a sample of  $8\beta n$  residual edges is chosen according to weights  $u_f(x, y)/k'(x, y)$ , then with high probability there is an augmenting path in the sample.*

**Corollary 4.2.2** *The running time of SampleResidual is  $O(m \log^3 n + nv \log^2 n \log m/n)$ .*

**Proof.** Given the theorem, we can easily add up the times for each step to prove the corollary. The first step runs in  $O(m \log^3 n)$  time [BK96]. In each iteration of the loop we need to select  $O(\beta n)$  edges and search for an augmenting path. If we order the edges arbitrarily, put them all in a binary search tree and maintain at each tree node the sum over edges in the subtree of  $u_f(x, y)/k'(x, y)$ , we can easily pick an edge in  $O(\log n)$  time. (Start at the root. Pick a subtree

to descend according to total weight (pick a subtree of weight  $x$  over a subtree of weight  $y$  with probability  $\frac{x}{x+y}$ ). Continue down the tree this way until reaching a leaf. Consider the leaf chosen and remove it from the tree.) So the random sampling will take only  $O(\beta n \log n)$  time. Searching for an augmenting path in  $O(\beta n)$  edges takes only  $O(\beta n)$  time. In order to be able to sample in the next iteration, we need to put the chosen edges back in the tree, with their capacities updated as appropriate. This takes another  $O(\log n)$  time per edge chosen. So the total time per iteration is  $O(\beta n \log n)$ . Thus the time to halve the remaining flow—the time to find  $(v - |f|)/2$  more paths—is  $O(nv \log^2 n)$ , at which time we will be due to double  $\alpha$ . We can double  $\alpha$  only  $\lg(m/n)$  times before  $\alpha n \geq m$ , so the total time for the loop is  $O(nv \log^2 n \log m/n)$ . When  $\alpha = m/n$ , it must be the case that  $v/(v - |f|) = \Omega(m/n \log n)$ , so  $v - |f| = O((nv \log n)/m)$ , which means that the time for the last step is only  $O(nv \log n)$ . Therefore the total running time is  $O(m \log^3 n + nv \log^2 n \log m/n)$ . ■

Observe that unlike in the previous chapter, we would get no benefit from running blocking flow for a while and then switching to our new algorithm to find the remaining paths. The reason is that it always takes  $\tilde{O}(nv)$  time to halve the remaining flow, regardless of how much flow is left. So we would get the same running time even if another algorithm had been used to find a lot of flow initially.

The proof of the theorem turns out to be a bit tricky. We would obviously like to simply use Karger's result [Kar99] that sampling such that the minimum expected number of edges is  $\Omega(\log n)$  will preserve connectivity. But that result relies crucially on the graph being undirected. Fortunately, it works by taking a union bound over all the cuts. So what we do is bound the probability of each cut surviving our sampling scheme (*i.e.* not losing all of its edges) by the probability of the same cut surviving the Benczúr-Karger sampling scheme in the original graph. We can then bound the probability our sampling fails to produce an augmenting path by the probability that Benczúr-Karger fails.

This plan is not as easy to execute as it sounds, because of the fact that the flow may use up low strength edges first, thereby greatly reducing the probability that a sampled edge crosses the cut. To see how dramatic the difference can be, consider a cut that has a few edges with strength one. Choosing these edges is a near certainty, so the probability of failure is practically zero. But the flow can quickly use them up and then, while the probability of failure may still be small, it can be far larger than it used to be. It is also worth noting that flow can cause damage without using up edges—cycles have no net effect on the capacity crossing a cut, but they can easily move capacity from low strength edges to high strength edges, increasing the chance that no edge crossing the cut will be chosen.

We escape these problems by arguing that for every cut there is always a  $k$ -strong component in which the flow has not caused damage. That is, for each cut, for some  $k$ , the sampling weights of  $k$ -strong residual edges still compare reasonably to the original weights. So we in fact make our argument by analyzing sampling not just in the original graph, but in each  $k$ -strong component of the original graph. Fortunately the  $k$ -strong components nest, so there can only be  $2n$  of them, which means that we can tolerate an extra union bound over components.

### 4.2.1 Supporting Lemmas

Before we can prove the main theorem, we need some supporting results. The first such is the result by Benczúr and Karger [BK96] that sampling an undirected graph with probability inversely proportional to strength preserves cut values well. The result that we need is slightly different from what they state, so we will state precisely what we need and provide a proof of it.

Note that it is common to refer to a cut as a non-empty set of vertices  $Y$  that is a proper subset of the vertices. However, when we discuss cuts our interest will be in the edges crossing the cut, and only the edges crossing in one direction at a time. So, for convenience, in our proofs in this chapter we use a shorthand of referring to a cut by the set of edges that cross the cut in a given direction, for which we have used the symbol  $X$ . In particular, in several cases where we wish to sum over the edges that cross a cut in a given direction, we write  $\sum_{(x,y) \in X}$ . Observe that the direction means that when we sum over all cuts in an undirected graph, we are counting each cut twice.

**Theorem 4.2.3** *In a connected undirected graph,*

$$\sum_{\text{cuts } X} e^{-\sum_{(x,y) \in X} \frac{(d+2)u(x,y) \ln n}{k(x,y)}} < \frac{2(d+2)}{dn^d}$$

To see how this theorem relates to sampling, consider sampling every edge with probability  $4u(x,y) \log n / k(x,y)$ . For a given cut  $X$ , the probability that no edge is chosen is

$$\prod_{(x,y) \in X} \left(1 - \frac{4u(x,y) \log n}{k(x,y)}\right) \leq \prod_{(x,y) \in X} e^{-\frac{4u(x,y) \log n}{k(x,y)}} = e^{-\sum_{(x,y) \in X} \frac{4u(x,y) \log n}{k(x,y)}}$$

Taking a union bound over all cuts, we get the quantity in the theorem and find that the probability that any cut has no edges chosen is at most  $4/n^2$ .

To prove this theorem we also need a theorem proved by Karger and Stein [KS96]:

**Theorem 4.2.4** *In an undirected graph with minimum cut value  $c$ , the number of undirected cuts of value  $\alpha c$  is at most  $n^{2\alpha}$ .*

We can now prove our restatement of the Benczúr-Karger result.

**Proof.** Consider the weighted graph with the same vertices and edges as  $G$  and weight  $w(x,y) = (d+2)u(x,y) \ln n / k(x,y)$  assigned to edge  $(x,y)$ . Order the undirected cuts of this graph in increasing order by value,  $c_1, c_2, \dots, c_{2^{n-1}-1}$ . Since we want to bound the sum over directed cuts, our goal is to bound

$$2 \sum_{i=1}^{2^{n-1}-1} e^{-c_i}$$

For any cut of the original graph, consider the maximum  $k$  of an edge crossing it. By definition of strength there must be at least  $k$  capacity crossing the cut, so the value of the cut in the weighted graph is at least  $(d+2) \ln n$ . Since this is true of every cut (note that no cut has no edges), the minimum cut of the weighted graph must be at least  $(d+2) \ln n$ . By Theorem 4.2.4,  $c_i \geq \frac{c_1 \ln i}{2 \ln n}$ , so we can bound our sum as

$$\begin{aligned}
&\leq 2 \sum_{i=1}^{n^2} e^{-c_1} + 2 \sum_{i=n^2}^{2^{n-1}-1} e^{-\frac{c_1 \ln i}{2 \ln n}} \\
&\leq 2 \sum_{i=1}^{n^2} e^{-(d+2) \ln n} + 2 \sum_{i=n^2}^{\infty} e^{-\frac{(d+2) \ln i}{2}} \\
&\leq 2n^{2-(d+2)} + 2 \sum_{i=n^2}^{\infty} i^{-1-d/2} \\
&\leq 2/n^d + 2 \int_{x=n^2}^{\infty} x^{-1-d/2} dx \\
&\leq 2/n^d + 2 \frac{n^{2(-d/2)}}{d/2} \\
&\leq \frac{2(d+2)}{dn^d}
\end{aligned}$$

■

The other supporting result we need is a little combinatorial lemma. This lemma will be used to argue that the flow cannot wreck our argument by using up low strength edges first.

**Lemma 4.2.5** *Given positive real numbers  $w_1 \dots w_l$  in decreasing order, for any real numbers  $x_1 \dots x_l$  such that  $\sum_{i=1}^l x_i \geq 0$ , there exists a  $j \in \{1 \dots l\}$  such that  $\sum_{i=j}^l w_i x_i \geq 0$ .*

**Proof.** Consider the largest  $j$  such that  $\sum_{h=j}^l x_h \geq 0$ .

$$\begin{aligned}
\sum_{i=j}^l w_i x_i &= \sum_{i=j}^l \left( w_i \sum_{h=i}^l x_h - w_i \sum_{h=i+1}^l x_h \right) \\
&= w_j \sum_{h=j}^l x_h + \sum_{i=j}^{l-1} \left( (w_{i+1} - w_i) \sum_{h=i+1}^l x_h \right)
\end{aligned}$$

The first term is non-negative by the choice of  $j$ . This choice also implies that for all  $i \geq j$  we have  $\sum_{h=i+1}^l x_h < 0$ , so the inner sum of the second term is always negative. Since the  $w_i$  are in decreasing order,  $w_{i+1} - w_i$  is non-positive, which means that the second term is a sum of non-negative numbers. So the entire expression is non-negative. ■

#### 4.2.2 Proof of the Main Theorem

We now have all the pieces necessary to prove the main theorem. The basic idea is to compare the residual graph to the original graph, which Theorem 4.2.3 gives us a handle on. As a

result of being a residual graph, at least a  $\frac{v-|f|}{v}$  fraction of the capacity of each cut is still available. What we are trying to say is that we can compensate for the missing capacity by increasing the sampling probability by a factor of  $\frac{v}{v-|f|}$ . Looking at an individual cut, this is not immediately obvious because the flow is not necessarily spread out evenly among the edges of different strengths. In particular, the flow might use up the high weight edges, in which case the probability that at least one edge crossing the cut is chosen can decrease significantly. However, if there is one edge that has low weight then there must be many, so if the flow only uses up the high weight edges, the many low weight edges that remain will be sufficient to make it very likely that some edge crossing the cut is chosen. We will use Lemma 4.2.5 to formalize this idea. We now give the full details.

**Proof.** Consider a cut  $X$  in the residual graph. Group the edges of the cut by strength, and arrange the groups in increasing order by strength. Associate with the  $i^{\text{th}}$  group

$$w_i = \frac{\beta}{k_i}$$

$$x_i = \sum_{(x,y) \in X: k(x,y)=k_i} \left( u_f(x,y) - \frac{v-|f|}{v} u(x,y) \right)$$

Thus the sum of the  $x_i$  is the residual capacity of the cut minus  $\frac{v-|f|}{v}$  times the original capacity of the cut. The residual capacity of every cut is at least a  $\frac{v-|f|}{v}$  fraction of the original capacity, so this quantity is always non-negative. Applying Lemma 4.2.5 we find that there exists a  $j$  such that  $\sum_{i=j}^l x_i w_i \geq 0$ ; or, rephrased in terms of the graph, we find that there exists a  $k$  such that

$$\sum_{(x,y) \in X: k(x,y) \geq k} \frac{\beta}{k(x,y)} \left( u_f(x,y) - \frac{v-|f|}{v} u(x,y) \right) \geq 0$$

$$\sum_{(x,y) \in X: k(x,y) \geq k} \frac{\beta u_f(x,y)}{k(x,y)} \geq \sum_{(x,y) \in X: k(x,y) \geq k} \frac{(v-|f|)\beta u(x,y)}{v k(x,y)}$$

$$= \sum_{(x,y) \in X: k(x,y) \geq k} \frac{4u(x,y) \ln n}{k(x,y)}$$

Since by definition of modified strengths the total weight of edges is at most  $8n$ , the probability that we fail to choose any of these  $k$ -strong edges is

$$\leq \left( 1 - \frac{\sum_{(x,y) \in X: k(x,y) \geq k} \frac{u_f(x,y)}{k'(x,y)}}{8n} \right)^{8\beta n}$$

$$\leq e^{-\sum_{(x,y) \in X: k(x,y) \geq k} \frac{\beta u_f(x,y)}{k'(x,y)}}$$

$$\leq e^{-\sum_{(x,y) \in X: k(x,y) \geq k} \frac{\beta u_f(x,y)}{k(x,y)}}$$

$$\leq e^{-\sum_{(x,y) \in X: k(x,y) \geq k} \frac{4u(x,y) \ln n}{k(x,y)}}$$



So with each cut we associate a  $k$ -strong component such that the probability we fail to choose an edge from the component is at most  $\exp(-\sum_{(x,y) \in X} \frac{4u(x,y) \ln n}{k(x,y)})$ . By Theorem 4.2.3, summing this quantity over all cuts that have the same component is at most  $4/n^2$ . Since there can be only  $2n$  distinct components, the probability that we fail to choose an edge from any cut is at most  $8/n$ .

If every  $s$ - $t$  cut in the residual graph has a residual edge crossing it, then by the max-flow min-cut theorem there is an augmenting path. ■

Observe that some corollaries to the main theorem are clear from the proof. In particular, if we had sampled every edge with probability  $\frac{\beta u_f(x,y)}{k'(x,y)}$  instead of randomly choosing a set of  $8\beta n$  edges, then the probability of failing to pick any  $k$ -strong edge from cut  $X$  would be

$$\prod_{(x,y) \in X: k(x,y) \geq k} \left(1 - \frac{\beta u_f(x,y)}{k'(x,y)}\right) \leq e^{-\sum_{(x,y) \in X: k(x,y) \geq k} \frac{\beta u_f(x,y)}{k'(x,y)}}$$

which we were already using as a bound. Likewise, in the proof we upper bound  $1/k'$  by  $1/k$ , so the theorem also holds if sampling is done according to the  $k$  instead of the  $k'$ .



## Chapter 5

# Minimum $k$ -Way Cuts

In this chapter we present better algorithms for finding minimum  $k$ -way cuts for  $k \leq 6$ . A preliminary version of this work appeared in a conference paper [Lev00].

A minimum  $k$ -way cut is a partition of the vertices into  $k$  sets that minimizes the total weight of edges with endpoints in different sets. For convenience, when talking about a  $k$ -way cut  $\{V_1, \dots, V_k\}$ , let us assume that  $V_1$  is the  $V_i$  with minimum  $c(V_i)$ . A common approach to the problem is to use the following observation: if the minimum  $k$ -way cut is  $\{V_1, \dots, V_k\}$  and one knows  $V_1$ , then  $\{V_2, \dots, V_k\}$  can be computed by finding the minimum  $(k-1)$ -way cut in the subgraph induced by the vertices  $V - V_1$ . More generally, if one can identify a collection  $\mathcal{X}$  of cuts that contains  $V_1$ , then  $|\mathcal{X}|$  minimum  $(k-1)$ -way cut computations suffice to find the minimum  $k$ -way cut. So one can approach the problem of finding  $k$ -way cuts by finding candidate 2-way cuts and then finding refinements of those 2-way cuts.

Burlet and Goldschmidt [BG97] show that for the minimum 3-way cut  $\{V_1, V_2, V_3\}$ , if  $X$  is the minimum cut,  $V_1$  must be a  $\frac{4}{3}$ -minimum cut in either the original graph, the vertex induced subgraph  $X$  or the vertex-induced subgraph  $V - X$ . That is,  $V_1$  is either a small cut, or it is contained in one side of a small cut and is small in that vertex-induced subgraph. A graph can have only  $O(n^2)$   $\frac{4}{3}$ -minimum cuts [NNI94], and we need only consider as candidates for  $V_1$  the  $\frac{4}{3}$ -minimum cuts of  $V$ , the  $\frac{4}{3}$ -minimum cuts of  $X$ , and the  $\frac{4}{3}$ -minimum cuts of  $V - X$ , so the total number of  $V_1$  candidates is  $O(n^2)$ . (Note that it is not a problem if the minimum cut of  $V$  is not unique. In that case, it suffices to only consider  $\frac{4}{3}$ -minimum cuts of  $V$ .) Near-minimum cuts can be computed efficiently, so Burlet and Goldschmidt's  $O(mn^3)$  running time follows from the time to compute  $V_2$  and  $V_3$  (with a minimum 2-way cut algorithm) for each of the  $O(n^2)$  candidate  $V_1$ . Notice that if they had been willing to give a randomized algorithm, they could have used Karger's  $\tilde{O}(m)$ -time algorithm for minimum 2-way cuts and claimed a time bound of  $\tilde{O}(mn^2)$ .

Nagamochi and Ibaraki [NI99] tighten this result by showing that if the 2-way cuts are sorted in increasing order of value, one only needs to consider (as candidates to be  $V_1$ ) cuts up until the point that the current cut *crosses* some previously considered cut. Deferring definition of crossing for a moment, the relevant point is that among any  $2n - 2$  cuts at least two cross, so their method only needs to consider  $O(n)$  cuts. Unfortunately, the best known

algorithm to find the first  $n$  cuts in order takes  $O(n^3m)$  time [VY92].

We use Burlet and Goldschmidt's result, together with fast algorithms for finding near-minimum cuts to find a superset of the cuts that Nagamochi and Ibaraki would consider, and then process them according to Nagamochi and Ibaraki's algorithm so that we need perform only  $O(n)$  minimum 2-way cut computations. In this way we avoid the bottleneck of both algorithms.

Note that some of the results in this chapter are effectively explained twice, once with pictures and once with math. To be convinced that the results are correct, read the math and skip the pictures. To just get the ideas, read the pictures and skip the math.

## 5.1 Background

The following is a summary of definitions, notation and background for  $k$ -way cuts.

A  $k$ -way cut is a partition of the vertices into  $k$  sets  $\{V_1, V_2, \dots, V_k\}$ . For convenience, we will always assume that the order of the indices is such that  $c(V_1) \leq c(V_2) \leq \dots \leq c(V_k)$ . The value of a  $k$ -way cut is  $\frac{1}{2}(c(V_1) + c(V_2) + \dots + c(V_k))$ . Observe that this value is the total weight of the edges whose endpoints are in different sets, so this definition is a generalization of the definition of the value of a cut. In fact, a "cut" and a "2-way cut" are the same thing.

The *minimum  $k$ -way cut* is the  $k$ -way cut of minimum value. We refer to this value as  $c_k$ . We will sometimes consider cuts in vertex-induced subgraphs, in which case we will write  $c_k(X)$  for the minimum  $k$ -way cut in the graph induced by  $X$ .

Notice that since  $c_k = \frac{1}{2}(c(V_1) + c(V_2) + \dots + c(V_k))$  and we have imposed an order on the  $V_i$ , it must always be the case that  $c_k \geq kc(V_1)/2$ . Or inverted,  $c(V_1) \leq 2c_k/k$ . Furthermore, since by definition  $c_2 \leq c(V_1)$ , we have  $c_k \geq kc_2/2$  and  $c_2 \leq 2c_k/k$ .

It is useful to notice that the cut value function is *symmetric*,  $c(X) = c(V - X)$ , and *submodular*,  $c(X) + c(Y) \geq c(X \cap Y) + c(X \cup Y)$  (cf. [NI99]). It follows immediately that  $c(X) + c(Y) \geq c(X - Y) + c(Y - X)$ .

Two sets are said to *overlap* if they intersect, and neither is a subset of the other. A collection of sets that do not overlap is called *laminar*. Laminar collections have some nice properties. Restating the definition, two sets in the collection can only intersect if one is a subset of the other. Furthermore, if  $X$  is a subset of  $Y$  and of  $Z$ , then since  $Y$  and  $Z$  intersect in  $X$  one must be a subset of the other. This means that if any sets contain  $X$ , there is a well defined set of smallest cardinality that contains  $X$ . It follows that the sets can be represented as a forest: the parent of  $X$  is the smallest cardinality set  $Y$  that contains it. See Figure 5.1 for an example. All the descendants of  $X$  are sets contained in  $X$ , and all of the ancestors are sets that contain  $X$ . If the ground set has  $s$  elements, then there can be at most  $s$  leaves in this forest, and since each node in the forest has at least two children, there can be at most  $2s - 1$  nodes. Therefore a laminar collection of sets on a ground set of size  $s$  can have at most  $2s - 1$  sets.

Cut  $X$  is said to *cross* cut  $Y$  if all of  $X \cap Y$ ,  $X - Y$ ,  $Y - X$  and  $V - (X \cup Y)$  are non-empty. In words, they intersect, neither is a subset of the other, and the union is not all of  $V$ . We say that a collection of cuts is *non-crossing* if no two cuts in the collection cross.

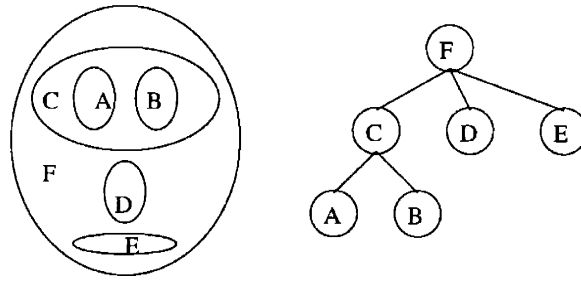


Figure 5.1: The tree corresponding to a laminar collection

Observe that if we wanted to store a cut  $X$ , we could instead store the other side,  $V - X$ , with no loss of information. Suppose we wanted to store a collection of non-crossing cuts. We could fix a node  $v_1$ , and plan to store the sides of the cuts in the collection that did not contain  $v_1$ . At this point, no two cuts in the collection could overlap, because neither contains  $v_1$ , so if they did overlap they would cross. Thus if we take a collection of non-crossing cuts, fix a vertex  $v_1$ , and choose the side of each cut that does not contain  $v_1$ , the collection is laminar. The ground set now has only  $n - 1$  nodes, so we see that a collection of non-crossing cuts can have cardinality at most  $2n - 3$ .

## 5.2 Minimum 3-Way Cuts

The key to our algorithm for minimum 3-way cuts is simultaneously exploiting the structural results of Burlet and Goldschmidt and those of Nagamochi and Ibaraki. We begin with those results, and then give the algorithm in detail.

### 5.2.1 Structural Results

The following lemmas restate the key claims proved by Nagamochi and Ibaraki [NI99]. The first one shows that two cuts of small value that cross give a 3-way cut of small value. The second one shows that either  $V_1$  of the minimum 3-way cut is itself a small cut, or a 3-way cut given by the crossing of two small cuts will be the minimum.

**Lemma 5.2.1** *For any two crossing cuts  $X$  and  $Y$  with  $c(X) \leq c(Y)$ , one of*

$$\begin{aligned} &\{V - X, X - Y, X \cap Y\} \\ &\{V - Y, Y - X, X \cap Y\} \\ &\{Y, X - Y, V - (X \cup Y)\} \\ &\{X, Y - X, V - (X \cup Y)\} \end{aligned}$$

*is a 3-way cut of value at most  $\frac{3}{4}(c(Y) + c(X)) \leq \frac{3}{2}c(Y)$ .*

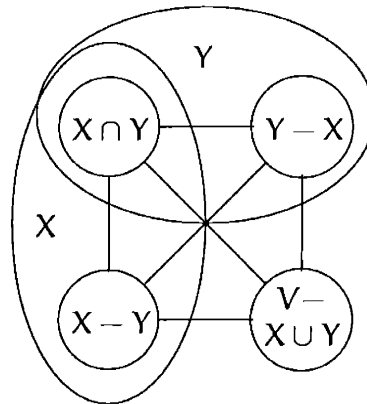


Figure 5.2: Crossing cuts

The idea behind this lemma is that the crossing of  $X$  and  $Y$  defines four non-empty pieces, and by merging two of the pieces into one, we get three pieces that define a small 3-way cut. Figure 5.2 shows the two cuts, the four pieces and the edges that cross among the four pieces. The 4-way cut defined by these pieces clearly has value at most  $c(X) + c(Y)$ . To get a 3-way cut we need to merge two pieces. If we pick the endpoints of the largest of the six "edges" to merge, we certainly eliminate  $\frac{1}{6}$  of the weight and therefore get a 3-way cut that has value at most  $\frac{5}{6}(c(X) + c(Y))$ . But we have overcounted, because the diagonal edges in the picture are counted twice. All this lemma really says is that since both cuts count the diagonal edges in the picture, we can pick two pieces to merge such that the 3-way cut has value only  $\frac{3}{4}(c(X) + c(Y))$ . Observe that this is now tight, because it is the best we can do for a 4-cycle.

**Proof.** Adding up the values of the four cuts listed above, and recalling that  $c(X) = C(V - X)$ , we get

$$\begin{aligned}
 & \frac{1}{2}(c(V - X) + c(X - Y) + c(X \cap Y)) \\
 & + \frac{1}{2}(c(V - Y) + c(Y - X) + c(X \cap Y)) \\
 & + \frac{1}{2}(c(Y) + c(X - Y) + c(V - (X \cup Y))) \\
 & + \frac{1}{2}(c(X) + c(Y - X) + c(V - (X \cup Y))) \\
 = & c(X) + c(Y) + c(X - Y) + c(Y - X) + c(X \cap Y) + c(V - (X \cup Y))
 \end{aligned}$$

Since the cut function is symmetric and submodular,  $c(X \cap Y) + c(V - (X \cup Y)) \leq c(X) + c(Y)$  and  $c(X - Y) + c(Y - X) \leq c(X) + c(Y)$ , so our total is at most  $3(c(Y) + c(X))$ . Therefore one of the four 3-way cuts listed above has value at most  $\frac{3}{4}(c(Y) + c(X)) \leq \frac{3}{2}c(Y)$ . ■

**Lemma 5.2.2** Consider all the cuts of the graph in non-decreasing order of cut value:  $X_1, X_2, \dots$  with  $c(X_1) \leq c(X_2) \leq \dots$ . Let  $r$  be the smallest index such that  $X_r$  crosses some  $X_q$  with  $q < r$ . The minimum 3-way cut is either

- given by  $\{V_1, V_2, V_3\}$ , where  $V_1 = X_s$  or  $V - X_s$  for some  $s < r$
- one of the cuts that follow from the crossing of  $X_r$  and  $X_q$  and lemma 5.2.1

**Proof.** If  $V_1 = X_s$  or  $V - X_s$  for some  $s \leq r$ , then we are done, so suppose not. This means that  $c(V_1) \geq c(X_r)$ . As  $c_3 \geq \frac{3}{2}c(V_1)$  by definition of  $V_1$ , this implies that  $c_3 \geq \frac{3}{2}c(X_r)$ . But by Lemma 5.2.1, one of the cuts listed in the statement of the lemma has value at most  $\frac{3}{4}(c(X_r) + c(X_q)) \leq \frac{3}{2}c(X_r)$ . Therefore this cut is a minimum one. ■

The following lemmas restate the key claims proved by Burlet and Goldschmidt [BG97]. The first says that  $V_1$  cannot cross a minimum cut unless it is a minimum cut. The second uses the first to say that  $V_1$  is either a small cut, or is contained in one side of the minimum cut and is a small cut in the subgraph induced by the side of the minimum cut.

**Lemma 5.2.3** *For a minimum 3-way cut  $\{V_1, V_2, V_3\}$ , no cut with value at most  $c(V_1)$  crosses any cut with value less than  $c(V_1)$ .*

**Proof.** Suppose there is a cut  $X$  such that  $c(X) \leq c(V_1)$  and  $X$  crosses a cut  $Y$  with value  $c(Y) < c(V_1)$ . By lemma 5.2.1, there exists a 3-way cut with value at most  $\frac{3}{4}(c(X) + c(Y)) \leq \frac{3}{4}(c(V_1) + c(Y)) < \frac{3}{2}c(V_1)$ —a contradiction to the fact that  $c_3 \geq \frac{3}{2}c(V_1)$ . ■

**Lemma 5.2.4** *Let  $X$  be a minimum cut. There exists a minimum 3-way cut  $\{V_1, V_2, V_3\}$  such that  $V_1$  is a  $\frac{4}{3}$ -minimum cut in one of  $V, X$  or  $V - X$ .*

The idea here is basically that  $c(V_1)$  can be large only if the minimum cuts of  $X$  and  $V - X$  are large, in which case  $V_1$  is contained in one of  $X$  or  $V - X$  and is still small in that vertex-induced subgraph.

**Proof.** Suppose  $V_1$  is not a  $\frac{4}{3}$ -minimum cut in the original graph ( $c(V_1) > \frac{4}{3}c_2$ ), because otherwise we are done. Recall from the definitions section that  $\frac{2}{3}c_3 \geq c(V_1)$ . It follows that  $c_3 > 2c_2$ . Let the smaller of the minimum cut in  $X$  and the minimum cut in  $V - X$  have value  $c'$ . Using this cut and the minimum cut we get a 3-way cut of value  $c_2 + c'$ , so  $c_3 \leq c_2 + c'$ . Since we already found that  $c_3 > 2c_2$ , it must be the case that  $c' > c_2$ . Now, since  $c(V_1) \leq \frac{2}{3}c_3 \leq \frac{2}{3}(c_2 + c')$  and  $c' > c_2$ , we have that  $c(V_1) < \frac{4}{3}c'$ .

So we now know that the value of  $V_1$  is small compared to the minimum cut of both  $X$  and  $V - X$ . Since the value of  $V_1$  in a vertex-induced subgraph can only be smaller than its value in the original graph, all we need to do to complete the proof is show that either  $V_1$  or  $V - V_1$  is actually contained in  $X$  or  $V - X$ . Alternately phrased, all we need to do is show that  $V_1$  does not cross  $X$ , which follows immediately from lemma 5.2.3 and the fact that we have already handled the case when  $c(V_1) = c_2$  (in fact when  $c(V_1) \leq \frac{4}{3}c_2$ ). ■

## 5.2.2 The Algorithm

The outline of the algorithm is given in Figure 5.3. We defer full specification of some of the steps to the analysis.

```

MIN3WAYCUT(G)
  find a minimum cut X and all the  $\frac{4}{3}$ -minimum cuts
  if  $|X| > 1$ , find all the  $\frac{4}{3}$ -minimum cuts in X
  if  $|V - X| > 1$ , find all the  $\frac{4}{3}$ -minimum cuts in  $V - X$ 
  for each cut Y found, compute the value  $c(Y)$  in the original graph
  sort all the cuts in non-decreasing order by value
  while not all cuts have been processed, process the next cut X in sorted order:
    if X crosses any cut Y previously processed
      look at the 4 cuts arising from lemma 5.2.1
      return best 3-way cut seen so far
    else
      if  $|X| > 1$ , find the minimum cut Y in X and look at  $\{V - X, Y, Y - X\}$ 
      if  $|V - X| > 1$ , find the minimum cut Y in  $V - X$  and look at  $\{X, Y, (V - X) - Y\}$ 
  return the best 3-way cut seen so far

```

Figure 5.3: Minimum 3-way cut algorithm

### 5.2.3 Correctness

**Theorem 5.2.5** *MIN3WAYCUT always returns the minimum 3-way cut.*

**Proof.** Our algorithm only considers as candidates for  $V_1$  the  $\frac{4}{3}$ -minimum cuts, the  $\frac{4}{3}$ -minimum cuts in  $X$  (a minimum cut) and the  $\frac{4}{3}$ -minimum cuts in  $V - X$ . By Lemma 5.2.4, this list of candidates is sufficient. That is, if the while loop processes all of the candidates, then we necessarily see the minimum 3-way cut.

Otherwise the while loop is stopped because we find a cut  $X$  that crosses a smaller cut. By lemma 5.2.2, either one of the four 3-way cuts given by this crossing is the minimum, or  $c(V_1) < c(X)$ . We look at all four of the cuts given by the crossing, so in the former case we succeed. In the latter case, since we processed cuts in sorted order, we have already seen the minimum, so we succeed. ■

### 5.2.4 Running Time Analysis

**Theorem 5.2.6** *MIN3WAYCUT can be implemented to run in  $O(mn \log^3 n)$  time.*

**Proof.** Karger's minimum cut algorithm can find all  $O(n^2)$   $\frac{4}{3}$ -minimum cuts in  $O(n^2 \log n)$  time [Kar00]. The algorithm obviously computes cut values for all of the cuts it finds as it is finding them; when we are finding cuts in a vertex-induced subgraph we need to add a step to compute cut values in the original graph. The value of a cut in a vertex-induced subgraph is different from its value in the original graph by the total weight of edges connecting nodes in the cut to nodes not in the subgraph. In linear time we can compute for each node of the subgraph the total weight of edges to nodes not in the subgraph. To compute the corrected cut values we just need to add up, for each cut, these values over the nodes in the cut. At first



glance it would seem that this will take  $n^3$  time, but we can do better for the same reason that Karger's algorithm does not take that long—the cuts found by Karger's algorithm can all be described by at most two subtrees of one of  $O(\log n)$  trees. We can easily compute the sum of node weights for all subtrees by a recursive postorder tree walk in  $O(n)$  time per tree. So in  $O(n \log n)$  time we can compute enough information that we can correct the value of each cut in  $O(1)$  time. Thus the entire operation of computing near minimum cuts and their values in the original graph can be done in  $O(n^2 \log n)$  time. Since there are only  $O(n^2)$  cuts, we can also sort them in  $O(n^2 \log n)$  time.

Testing crossing is the trickiest part. We will now explain how to do this in  $O(n)$  time per test (for a total of  $O(n^2)$  time). Recall from the definitions section that if we have a collection of cuts that do not cross, and we pick the side of each cut that does not contain some fixed vertex  $v_1$ , then the collection is laminar. Recall also that a laminar collection can be represented by a forest. We will accomplish our crossing test by explicitly maintaining this forest.

It is convenient to begin by pretending that each individual vertex (except  $v_1$ ) and the set of all vertices (except  $v_1$ ) are in the collection initially. This way the forest starts as a flat tree, with each vertex as a leaf and the set of all vertices as the root. None of these cuts can possibly cross any other cut, so this does no harm.

The nice thing about starting this way is that the vertices contained by a cut will now be precisely the leaves in its subtree. Inserting a cut is the only operation we need to implement. If the cut does not cross any previous cut it should be inserted, and if it does the insertion should fail. To attempt to insert  $X$ , we will first go through the list of vertices in  $X$  and mark the corresponding leaves in the tree “subset of  $X$ ”. The total time to do this marking is  $O(n)$ . We would now like to compute, for every node in the tree, whether it too is a subset of  $X$ . This can be easily accomplished with a recursive postorder tree walk (in  $O(n)$  time), because a node is a subset of  $X$  if and only if all of its children are.

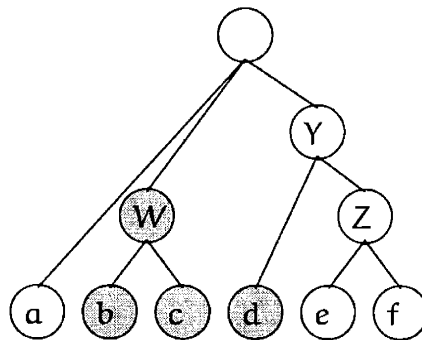


Figure 5.4: Attempt to insert  $X = \{b, c, d\}$ . Shaded nodes are “subsets of  $X$ ”. They do not have a common parent because  $X$  overlaps  $Y$ . Observe that for a different  $X$ , such as  $\{a, b, c\}$ , where there are containments but no overlap, the subsets of  $X$  would have a common parent.

At this point the key claim is that  $X$  overlaps some set in the tree if and only if the subtrees marked “subset of  $X$ ” do not have a common parent. Figure 5.4 has a picture of this situation. If  $X$  overlaps some set  $Y$ , then some of  $Y$ 's descendants will be subsets of  $X$ , but  $Y$  will not be. Furthermore, some other sets that are not descendants of  $Y$  will be subsets of  $X$ . Therefore

$Y$  prevents subtrees that are subsets of  $X$  from having a common parent. Now we argue the other direction. If the subtrees that are subsets of  $X$  do not have a common parent, then consider  $Y$ , the parent of one of these subtrees. Since  $Y$  is not itself a subset of  $X$ , but intersects  $X$ , and does not contain  $X$ ,  $Y$  overlaps  $X$ .

Our insertion procedure should now be clear. We do the marking as described above. We then check each node that is not a subset of  $X$  to see if it has children that are subsets of  $X$ . If we find only one such node  $Y$ , we add a new node corresponding to  $X$  as a child of  $Y$ , and make all of the children of  $Y$  that were subsets of  $X$  into children of  $X$ . Otherwise we fail. The total time per insertion is clearly  $O(n)$ .

Finally, Karger's minimum cut algorithm can find one minimum cut in  $O(m \log^3 n)$  time. We need to do this 2 times per iteration, and there are only  $O(n)$  iterations, so the total time spent on these operations is  $O(mn \log^3 n)$ . Thus the total time is  $O(mn \log^3 n)$ . ■

### 5.3 Minimum 4-Way Cuts

The minimum 4-way cut algorithm is an extension of the minimum 3-way cut algorithm. Nagamochi and Ibaraki's result covered 4-way cuts, but Burlet and Goldschmidt's result did not. Fortunately, however, there is a suitable extension.

#### 5.3.1 Structural Results

The following lemmas are extensions of those for 3-way cuts. The first two restate claims proved by Nagamochi and Ibaraki [NI99]. The rest are new.

**Lemma 5.3.1** *For any two crossing cuts  $X$  and  $Y$  with  $c(X) \leq c(Y)$ , the 4-way cut  $\{X \cap Y, X - Y, V - (X \cup Y), Y - X\}$  has value at most  $c(X) + c(Y) \leq 2c(Y)$ .*

**Proof.** It follows immediately from the fact that the cut function is symmetric and submodular that the value of this 4-way cut is at most  $c(X) + c(Y) \leq 2c(Y)$ . It is also trivial to see by looking at Figure 5.2. ■

**Lemma 5.3.2** *Consider all the cuts of the graph in non-decreasing order of cut value:  $X_1, X_2, \dots$  with  $c(X_1) \leq c(X_2) \leq \dots$ . Let  $\tau$  be the smallest index such that  $X_\tau$  crosses a  $X_q$  with  $q < \tau$ . Either the minimum 4-way cut is given by  $\{V_1, V_2, V_3, V_4\}$ , where  $V_1 = X_s$  or  $V - X_s$  for some  $s \leq \tau$ , or it is  $\{X_\tau \cap X_q, X_\tau - X_q, V - (X_\tau \cup X_q), X_q - X_\tau\}$*

In short, the minimum 4-way cut is either a refinement of a small 2-way cut, or it is given by the crossing of two small cuts.

**Proof.** If  $V_1 = X_s$  or  $V - X_s$  for some  $s \leq \tau$ , then we are done, so suppose not. This means that  $c(V_1) \geq c(X_\tau)$ . Since  $c_4 \geq 2c(V_1)$ , this implies that  $c_4 \geq 2c(X_\tau)$ . But by Lemma 5.3.1,

$\{X_r \cap X_q, X_r - X_q, V - (X_r \cup X_q), X_q - X_r\}$  has value at most  $c(X_r) + c(X_q) \leq 2c(X_r)$ . Therefore this cut is a minimum one. ■

We now generalize Burlet and Goldschmidt's results to 4-way cuts.

**Lemma 5.3.3** *For a minimum 4-way cut  $\{V_1, V_2, V_3, V_4\}$ , no cut with value at most  $c(V_1)$  crosses any cut with value less than  $c(V_1)$ .*

**Proof.** Suppose there is a cut  $X$  such that  $c(X) \leq c(V_1)$  and  $X$  crosses a cut  $Y$  with  $c(Y) < c(V_1)$ . By lemma 5.3.1, there exists a 4-way cut with value at most  $c(X) + c(Y) \leq c(V_1) + c(Y) < 2c(V_1)$ —a contradiction. ■

**Lemma 5.3.4** *Let  $X$  be a minimum cut such that the minimum cut of the subgraph  $X$  is less than the minimum cut of the subgraph  $V - X$ , that is, that  $c_2(X) \leq c_2(V - X)$ . Let  $X_1$  be a minimum cut in  $X$  such that  $c_2(X_1) \leq c_2(X - X_1)$ . There exists a minimum 4-way cut  $\{V_1, V_2, V_3, V_4\}$  such that  $V_1$  is a  $\frac{3}{2}$ -minimum cut in one of  $V, X, V - X, X_1, X - X_1$ .*

Note that Figure 5.5 may be helpful in keeping track of all the pieces in this lemma and proof.

**Proof.** If  $c(V_1) \leq \frac{3}{2}c_2$  then we are done, so suppose not. Recall from the definitions section that  $c(V_1) \leq \frac{1}{2}c_4$ . It follows that  $c_4 > 3c_2$ .

Let  $X$  be as in the statement of the lemma. By lemma 5.3.3, since  $c(V_1) > c_2$  we know that  $V_1$  does not cross the minimum cut, so either  $V_1$  or  $V - V_1$  is contained in one of  $X$  or  $V - X$ . We have defined  $X$  such that  $c_2(X) \leq c_2(V - X)$ ; therefore if  $c(V_1) \leq \frac{3}{2}c_2(X)$  we are done (recall that restricting to a vertex-induced subgraph can only reduce the value of a cut), so suppose not. It follows that  $c_4 > 3c_2(X)$ .

Taking the minimum cut in  $V$ , the minimum cut in  $X$ , and the smaller of the minimum cut in  $X_1$  and the minimum cut in  $V - X$ , we get a 4-way cut of value  $c_2 + c_2(X) + \min\{c_2(X_1), c_2(V - X)\}$ . Since we already found that  $c_4 > 3c_2$  and  $c_4 > 3c_2(X)$ , it must be the case that the third term is largest, that is, that  $c_4 \leq 3 \min\{c_2(X_1), c_2(V - X)\}$ . Furthermore, since  $c(V_1) \leq \frac{1}{2}c_4$ , we get that  $c(V_1) < \frac{3}{2} \min\{c_2(X_1), c_2(V - X)\}$ .

Either  $c(X_1)$  or  $c(X - X_1)$  has value at most  $c_2(X) + c_2/2$  in the original graph (this is easiest to see by looking at Figure 5.5). Since we have already restricted to the case where  $c(V_1) > \frac{3}{2}c_2$  and  $c(V_1) > \frac{3}{2}c_2(X)$ , it must be the case that  $V_1$  has a larger value than the smaller of  $c(X_1)$  and  $c(X - X_1)$ . So by lemma 5.2.3  $V_1$  cannot cross whichever of  $X_1$  or  $X - X_1$  has the smaller value. But neither can  $V_1$  cross the minimum cut, so  $V_1$  or  $V - V_1$  must be contained in one of  $X_1, X - X_1$  or  $V - X$ , and since  $V_1$  has value at most  $\frac{3}{2}$  times the minimum of the minimum cut values of these three sets, it is a  $\frac{3}{2}$ -minimum cut in one of them. ■

### 5.3.2 The Algorithm

The outline of the algorithm is given in Figure 5.6. We defer full specification of some of the steps to the analysis.

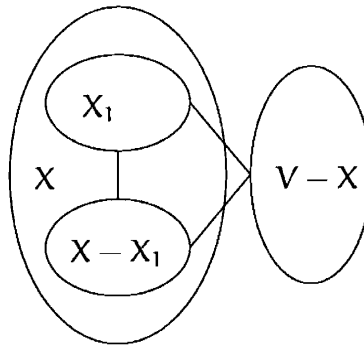


Figure 5.5: Either  $c(X_1)$  or  $c(X - X_1)$  has value at most  $c_2(X) + c_2/2$

Note that we switch algorithms for finding small 2-way cuts, now using Karger and Stein's algorithm [KS96] instead of Karger's [Kar00]. We made this change largely as a matter of clarity. Karger's algorithm ought to be able to find  $\alpha$ -minimum cuts in  $\tilde{O}(n^{\lfloor 2\alpha + \epsilon \rfloor})$  time, but no paper spells out how to do this for  $\alpha \geq 3/2$ . Karger and Stein's algorithm is clearly specified for finding  $\alpha$ -minimum cuts in  $\tilde{O}(n^{2\alpha})$  time. So Karger's algorithm is clearly preferable for  $\alpha = 4/3$ , which we needed for the minimum 3-way cut algorithm; however, the running times are comparable for  $\alpha = 3/2$ , which we need now, and since Karger's algorithm is not clearly specified for this case we felt it was preferable to state our algorithm with the Karger-Stein algorithm as a subroutine.

### 5.3.3 Correctness

**Theorem 5.3.5** *MIN4WAYCUT always returns the minimum 4-way cut.*

**Proof.** The cuts we consider as candidates to be  $V_1$  are only those specified in Lemma 5.3.4, but by that lemma those candidates always contain  $V_1$ . This means that if the algorithm processes all of the cuts found, then we must see  $V_1$ . Given  $V_1$ , the other sets  $V_2$ ,  $V_3$ , and  $V_4$  are clearly given by the minimum 3-way cut in  $V - V_1$ , so we have necessarily seen the minimum 4-way cut.

Otherwise the algorithm stops because it finds a cut  $X$  that crosses a smaller cut. By lemma 5.3.2, either the 4-way cut given by this crossing is the minimum, or  $c(V_1) < c(X)$ . In the former case we look at that cut, so we succeed. In the latter case, since we processed cuts in sorted order, we have already seen the minimum, so we succeed. ■

### 5.3.4 Running Time Analysis

**Theorem 5.3.6** *MIN4WAYCUT can be implemented to run in  $O(mn^2 \log^3 n)$  time.*

**Proof.** The Karger-Stein minimum cut algorithm can find all  $\frac{3}{2}$ -minimum cuts in  $O(n^3 \log^2 n)$  time. By keeping track of the original graph edges through the contractions, cut values in the

```

MIN4WAYCUT(G)
  find a minimum cut X and all the  $\frac{3}{2}$ -minimum cuts
  if  $|X| > 1$ , find all the  $\frac{3}{2}$ -minimum cuts in X
  if  $|V - X| > 1$ , find all the  $\frac{3}{2}$ -minimum cuts in  $V - X$ 
  if  $c_2(X) > c_2(V - X)$ , swap the names X and  $V - X$ 
  Let  $X_1$  be the minimum cut in X
  if  $|X_1| > 1$ , find all the  $\frac{3}{2}$ -minimum cuts in  $X_1$ 
  if  $|X - X_1| > 1$ , find all the  $\frac{3}{2}$ -minimum cuts in  $X - X_1$ 
  for each cut Y found, compute the value  $c(Y)$  in the original graph
  sort all the cuts in non-decreasing order by value
  while not all cuts have been processed, process the next cut X in sorted order:
    if X crosses any cut Y previously processed
      look at  $\{X \cap Y, X - Y, V - (X \cup Y), Y - X\}$ 
      return best 4-way cut seen so far
    else
      if  $|X| > 1$ , find the minimum 3-way cut  $\{V_2, V_3, V_4\}$  in X and look at
         $\{V - X, V_2, V_3, V_4\}$ 
      if  $|V - X| > 1$ , find the minimum 3-way cut  $\{V_2, V_3, V_4\}$  in  $V - X$  and look at
         $\{X, V_2, V_3, V_4\}$ 
  return the best 4-way cut seen so far

```

Figure 5.6: Minimum 4-way cut algorithm

original graph can be computed at the same time, with no asymptotic loss in performance. There are only  $O(n^3)$  cuts, so we can sort them in  $O(n^3 \log n)$  time. As we described for 3-way cuts, testing crossing can be done in  $O(n)$  time per check by maintaining the tree of cuts; this adds up to  $O(n^2)$  time. The algorithm from the previous section can find minimum 3-way cuts in  $O(mn \log^3 n)$  time. We need to do this  $O(1)$  times per iteration, and there are only  $O(n)$  iterations, so the total time spent there is  $O(mn^2 \log^3 n)$ . So the total is  $O(mn^2 \log^3 n)$  time. ■

## 5.4 Minimum 5-Way Cuts, 6-Way Cuts and Beyond

The extension of our results to 5-way and 6-way cuts follows trivially from the algorithm of Nagamochi, Katayama and Ibaraki [NKI99], because the time to find the first  $O(n)$  cuts in sorted order ceases to be the bottleneck. Rather, the bottleneck in their minimum 5-way cut algorithm is that it must call a minimum 4-way cut algorithm  $O(n)$  times. In particular, denoting the time to find a minimum  $k$ -way cut as  $C_k(n, m)$ , they prove the following result:

**Lemma 5.4.1** [NKI99] For  $k \leq 6$ ,  $C_k(n, m) = O(n^2 F(n, m) + n C_{k-1}(n, m) + n^3)$ .

Plugging in our results that  $C_4(n, m) = O(mn^2 \log^3 n)$ , we get that for  $k \leq 6$ ,  $C_k(n, m) = O(mn^{k-2} \log^3 n)$ .

Note that it is fortunate that finding the cuts ceases to be a bottleneck, because it does not seem like the Burlet-Goldschmidt type results would extend any further. Specifically, for  $k \geq 5$  it ceases to be true that no small cut can cross  $V_1$ . Figure 5.7 gives a counterexample. The cut defined by the triangle on the left has value 11, and crosses  $V_1$ , which has value 12.

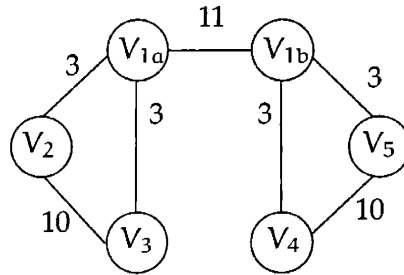


Figure 5.7: Small cuts can cross  $V_1$  for  $k \geq 5$

## Chapter 6

# Conclusion

### 6.1 Maximum Flow

We gave two algorithms for finding augmenting paths in undirected graphs, both based on exploiting the fact that undirected graphs have very sparse subgraphs that capture connectivity information.

It is natural to ask whether our techniques can be extended further. The best performance improvement we could hope for from the technique of Chapter 3 is reduction of  $m$  to  $n\sqrt{v}$ ; we achieve this reduction for augmenting paths, but only get part way when blocking flows are involved. It would be nice to find a way to sparsify for a blocking flow computation. In particular, if we could achieve a full reduction to  $n\sqrt{v}$  edges when blocking flows were involved, it would imply a deterministic  $O(n\sqrt{v}n^{2/3}) = O(n^{2.1\bar{6}})$ -time algorithm. It is interesting to note that the structure theorem—that a flow does not use many edges—holds for directed graphs, but this fact is of no use to us, because the point of the theorem was to show that not too many edges of the residual graph were directed. It would of course be wonderful to find a sparsification technique for directed graphs, but then one would not need our theorem.

Our result of  $\tilde{O}(m + nv)$  time given in Chapter 4 is a natural stopping point for algorithms for small maximum flows in undirected graphs, but it is not necessarily the end of progress on the problem. First, it is randomized, and better than our deterministic results given in Chapter 3, so there is still the question of how well a deterministic algorithm can do. Perhaps there is a better way to apply Nagamochi-Ibaraki sparse certificates [NI92b] in a residual graph than ignoring the edges that carry flow. Second, we showed in Chapter 3 that flows need only use  $O(n\sqrt{v})$  edges. Therefore, while some augmenting paths can require  $n - 1$  edges, most of them are much shorter. Thus  $\tilde{O}(m + n\sqrt{v})$  would be another natural time bound to hope to achieve. And of course one can always hope for linear time.

It is interesting to note that both of our algorithms are very simple, although the proofs that they have good running times are not so simple. Our work suggests that it may be worth looking for complicated reasons why other simple algorithmic ideas for finding flows might give performance improvements.

The major open questions at this point are whether it is possible to give faster algorithms for directed graphs or graphs with large flow values. Note that these questions are not entirely separate. It is possible to reduce the problem of finding a minimum  $s$ - $t$  cut in a directed graph to that of finding a minimum  $s$ - $t$  cut in an undirected graph [Que]. However, the cut and flow values will be larger. Therefore a faster algorithm for finding flows in capacitated, undirected graphs would immediately imply a faster algorithm for finding minimum  $s$ - $t$  cuts in capacitated, directed graphs. For reference, the idea of the transformation is to replace directed edge  $(x, y)$  of capacity  $u$  with undirected edges  $\{s, y\}$ ,  $\{x, y\}$  and  $\{x, t\}$  each of capacity  $u/2$ . This change increases all  $s$ - $t$  cuts by  $u/2$ , but has the desired property that cuts with  $x$  on the  $s$  side and  $y$  on the  $t$  side are  $u$  larger than other cuts. So cut values in the new graph are different (larger) than in the original, but the ordering of cuts by size is preserved. The idea can be refined slightly by observing that if one reduces the capacities of  $\{s, x\}$  and  $\{x, t\}$  by the smaller of the two, then all cuts lose the same amount of capacity, so the ordering of cut values is again preserved. This refinement means that when one converts from a directed graph to an undirected graph it is not necessary to increase all cut values by the total edge weight—it is sufficient to increase all cut values by the sum over nodes of the difference between total incoming capacity and total outgoing capacity. Thus in a directed graph where every node has the same incoming capacity as outgoing capacity, the minimum  $s$ - $t$  cut can be found by solving a maximum flow problem in an undirected graph with the same number of edges and the same flow value. Likewise, if one applies this transformation to a graph in which the capacities are symmetric (in other words, an undirected graph), nothing changes. And if one applies this transformation to the residual graph of an undirected graph, one recovers the original undirected graph.

In sampling from residual graphs, we have shown that random sampling in directed graphs is not entirely hopeless. Perhaps there is a suitable replacement for edge strength in a directed graph that would allow random sampling in directed graphs. Also, it is intriguing that a random sample, while sloppier than a structure like a spanning forest, might be preferable because it is more robust against changes to the graph. Perhaps there are other situations where a simple deterministic structure is not as useful as one would hope, and a random sample would do a better job.

As for trying to extend our results to capacitated graphs, one obvious goal would be to replace  $m$  by  $n$  in general. That is, what we did was show that it is possible to work with  $\tilde{O}(n)$  edges on average when searching for augmenting paths, thereby effectively replacing the  $m$  in the  $O(mv)$ -time augmenting paths algorithm with an  $n$ . Accordingly, for capacitated graphs, attempting to replace  $m$  by  $n$  in the running time of some algorithm seems like an appropriate target. We note that our sampling theorem applies perfectly well to capacitated graphs. As long as an  $\epsilon$  fraction of flow is left, a sample of size  $O(\frac{n \log n}{\epsilon})$  edges will have an augmenting path. It might be possible to construct an approximation algorithm from this result, but once the remaining flow is a small fraction of the original, we do not see how to proceed. We also note that it is possible to prove a compression theorem for a residual graph (see Appendix B): sampling with probabilities proportional to  $1/k_e$  and multiplying the capacity of sampled edges by  $k_e$  preserves all cut values reasonably well. It might be possible to use this result to give an approximation algorithm like that of Benczúr-Karger [BK96] that has a slightly better dependence on  $\epsilon$ , but again, we do not see how to get a good



exact algorithm for general capacities.

One other surprising implication of our results has to do with bipartite matching. A bipartite matching problem can be reduced to a flow problem in a directed graph, and this flow problem can actually be solved in  $O(m\sqrt{n})$ -time. This is better than what is known for flow in simple, directed graphs, so bipartite matching has long seemed a little easier than flow. Our results invert this, saying that flow in undirected graphs is easier than bipartite matching. The standard reduction from bipartite matching to flow does not work if the graph is not directed [Gol97] (see Figure 6.1), and the reduction mentioned above from directed  $s$ - $t$  cut problems to undirected  $s$ - $t$  problems would increase  $v$  to  $m$  if applied in this case, but our work opens the question of whether bipartite matching can be reduced to undirected flow or, more generally, whether the time for bipartite matching is really correct.

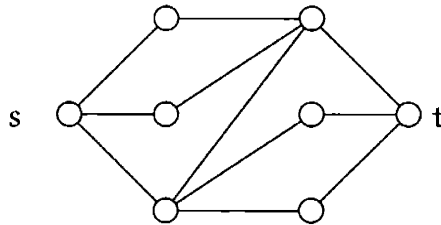


Figure 6.1: With edges directed from left the right, this graph is a flow problem representing a maximum bipartite matching problem; if the edge directions are removed then the flow value increases, no longer corresponding to a matching.

## 6.2 Minimum $k$ -Way Cut

There are also a number of open questions about the minimum  $k$ -way cut problem. For one, does  $C_k(n, m) = \tilde{O}(mn^{k-2})$  hold for any constant  $k$ ? for all  $k$ ? This would be true if Nagamochi, Katayama and Ibaraki's recurrence held for all  $k$ . It is also natural to wonder if minimum 3-way cuts can be found any faster. Since, up to logarithmic factors, minimum 2-way cuts cannot be found faster, this would seem to require either finding a way to consider fewer candidates for  $V_1$  or finding a way to reuse information, such that over the  $O(n)$  2-way cut computations, the amortized cost of each was less than  $O(m)$ . More generally, one can ask if a minimum  $(k+1)$ -way cut computation must really cost  $n$  times more than a minimum  $k$ -way cut computation. There is likely to be an exponential dependence on  $k$ , since the problem is NP-complete for general  $k$ , but it is not clear that it must be a function like  $n^k$ , as opposed to, say,  $2^k$ .



## Appendix A

# Randomized Algorithms Using Fast Augmenting Paths

As mentioned in the introduction, the fast augmenting paths algorithms given in Chapter 3 can be combined with prior techniques to obtain other randomized algorithms. At the time these algorithms were discovered, they were the fastest known for dense undirected graphs with small flow values. However, they are both slower and more complicated than the algorithm given in Chapter 4, so it is not clear that they are of much interest anymore. We include this appendix only because the techniques used in it might be of independent interest. To avoid unnecessary complication, we describe these algorithms for simple undirected graphs. Using Karger's graph smoothing technique [Kar98] they can immediately be extended to undirected graphs with capacities.

### A.1 New Tricks for an Old DAUG

Our fast augmentation can improve the running time of the "divide and augment" algorithm (DAUG) given by Karger [Kar99]. This result is of relatively minor interest in itself, but we make good use of it in the next section.

The idea of DAUG is that if we randomly divide the edges of a graph into two groups, then about half of the flow can be found in each group. So we can recursively find a maximum flow in each half, put the halves back together, and use augmenting paths to find any flow that was lost because of the division. In the original version, the time spent finding augmenting paths at the top level dominated the running time, so it is natural to expect an improvement with faster augmentations. The original algorithm is given in Figure A.1.

The key fact that makes DAUG work is that random sampling preserves cut values fairly well as long as all cuts are large enough. (See Theorem 2.5.2 in Section 2.5.)

Thus when we divide the edges into two groups (effecting  $p = 1/2$  in each group), the minimum  $s$ - $t$  cut in each group is at least  $\frac{v}{2}(1 - O(\sqrt{\log n/c}))$  with high probability. So the flow in each half has at least this value, giving us a flow of value at least  $v(1 - O(\sqrt{\log n/c}))$

```

DAUG(G)
  if G has no edges, return the empty flow
  randomly divide the edges of G into two groups, giving G1 and G2
  f1 ← DAUG(G1)
  f2 ← DAUG(G2)
  f ← f1 + f2
  (*) use augmenting paths to turn f into a maximum flow
  return f

```

Figure A.1: The original “divide and augment” algorithm

when we put the two halves together. This leaves only  $O(v\sqrt{\log n/c})$  augmenting paths to be found in Step (\*). It turns out that this step is the dominant part of the running time (the time bound for DAUG is  $O(mv\sqrt{\log n/c})$ ), so it makes sense to use our fast augmenting paths algorithm from Chapter 3. We refer to this new algorithm as *newDAUG*.

Now, by Theorem 3.3.3, the time to find the augmenting paths is  $O(m + nv\sqrt{v\log n/c})$ . So a recurrence for the running time of *newDAUG* is

$$T(m, v, c) = 2T(m/2, v/2, c/2) + O\left(m + nv\sqrt{v\log n/c}\right)$$

This solves to  $\tilde{O}(m + nv\sqrt{v/c})$ , but unfortunately, because of the randomization in the algorithm, the problem reduction is expected, not guaranteed, so solving this recurrence does not actually prove anything about the running time of *newDAUG*. We need to look at the recursion tree (Karger [Kar99] has a full discussion of this issue). This proof is more technical than interesting, and goes the same way as in the original [Kar99], so we just sketch it.

**Theorem A.1.1** *The running time of newDAUG on a  $c$ -edge connected graph is  $\tilde{O}(m + nv\sqrt{v/c})$ .*

**Proof.** (Sketch) As in the original algorithm, the depth of the recursion tree is  $O(\log m)$ , and the time spent looking unsuccessfully for augmenting paths is  $O(m \log m)$ . It remains to bound the time spent in successful augmentations. Consider a recursion node  $N$  at depth  $d$ . Each edge of the original graph ends up at  $N$  independently with probability  $2^{-d}$ , so the graph at this node is equivalent to one obtained by sampling with probability  $2^{-d}$ .

Consider the nodes at depths exceeding  $\log(c/\log n)$ . By Theorem 2.5.2, at these nodes the flow is  $\tilde{O}(v/c)$ . So by Theorem 3.3.3, the total time spent on successful augmenting paths is  $\tilde{O}(nv\sqrt{v/c})$ . At the nodes at depth  $d \leq \log(c/\log n)$ , Karger’s argument [Kar99] continues to apply, showing that the number of augmenting paths that need to be found is  $O(v\sqrt{\log n/2^d c})$ . Since the value of the flow is  $O(v/2^d)$ , the time taken is  $\tilde{O}((v\sqrt{1/c})n\sqrt{v/2^d}) = \tilde{O}(nv\sqrt{v/c/2^d})$ . Adding this up over the whole recursion, we get the claimed bound. ■

Note that this time bound is very good if  $v$  is not much bigger than  $c$ . In particular, we get the following easy corollary:

**Corollary A.1.2** *In a simple graph where  $v = \tilde{O}(c)$ , the running time of newDAUG is  $\tilde{O}(m + nv) = \tilde{O}(m)$ . (Note that  $m \geq nc/2$  in a  $c$ -edge connected simple graph.)*

## A.2 $\tilde{O}(m + nv^{5/4})$ - and $\tilde{O}(m + n^{11/9}v)$ -Time Algorithms

The algorithm of the previous section is only an improvement over the  $O(m + nv^{3/2})$ -time algorithm if  $c$  is large. Nevertheless, we can take advantage of it by using ideas from another paper by Karger [Kar98]. In that paper, a number of ideas are put together to get a fast flow algorithm, CompressAndFill, that runs in  $\tilde{O}(v\sqrt{mn})$  time on any undirected graph. For our purposes, that algorithm can be summarized with the following theorem:

**Theorem A.2.1** [Kar98] *Let  $T(m, n, v, c)$  denote the time to find a maximum flow of value  $v$  in a  $c$ -edge connected undirected graph with  $m$  edges and  $n$  nodes. Given flow algorithms  $A_1$  and  $A_2$  ( $A_1$  must handle capacities), with running times  $T_1$  and  $T_2$  respectively, it is possible to define a flow algorithm  $A_3$  with expected running time (up to log factors) given by*

$$T_3(m, n, v, c) \leq T_1(nk, n, v, c) + T_2(m, n, v, k) + T_2(m, n, k, k) \\ + \text{time to find } O(v/\sqrt{k}) \text{ augmenting paths}$$

(There is a technicality that the bound of  $T_2$  must be “reasonable”— $\Omega(m + n)$ —for this theorem to be true.)

CompressAndFill results from picking  $k \approx m/4n$ , using CompressAndFill (recursively) for  $A_1$  and using DAUG (with runtime  $\tilde{O}(mv/\sqrt{k})$ ) for  $A_2$ . Thus the recurrence for the running time is

$$T(m, n, v, c) \leq T(m/2, n, v, c) + \tilde{O}(mv\sqrt{k}) + \tilde{O}(m\sqrt{k}) + \tilde{O}(mv\sqrt{k}) \\ \leq T(m/2, n, v, c) + \tilde{O}(v\sqrt{mn}) \\ \leq \tilde{O}(v\sqrt{mn})$$

We improve on this algorithm by replacing the subroutines  $A_1$  and  $A_2$  and the augmenting path step appropriately. In particular, we use newDAUG instead of DAUG for  $A_2$  and we find augmenting paths at the end with SparseAugment. We also consider two possibilities for  $A_1$ : CompressAndFill and the  $\tilde{O}(mn^{2/3})$ -time algorithm of Goldberg and Rao. Note that we investigated using a recursive strategy again, but we were unable to get an improvement that way.

**Theorem A.2.2** *On undirected simple graphs, we can find a maximum flow in expected time  $\tilde{O}(m + nv^{5/4})$ .*

**Proof.** Use Theorem A.2.1 with  $A_1 = \text{CompressAndFill}$ ,  $A_2 = \text{newDAUG}$ , and SparseAugment to find the augmenting paths at the end. The resulting time bound is

$$\tilde{O}(v\sqrt{(nk)n}) + \tilde{O}(nv^{3/2}/k^{1/2}) + \tilde{O}(nv) + \tilde{O}(n\sqrt{v} \cdot v/\sqrt{k})$$

$$= \tilde{O}(vn\sqrt{k} + nv^{3/2}/k^{1/2})$$

Picking  $k = \sqrt{v}$  completes the proof. ■

**Theorem A.2.3** *On undirected simple graphs, we can find a maximum flow in expected time  $\tilde{O}(m + n^{11/9}v)$ .*

**Proof.** Use Theorem A.2.1 with  $A_1 =$  the  $\tilde{O}(mn^{2/3})$ -time algorithm of Goldberg and Rao [GR97a],  $A_2 =$  newDAUG, and SparseAugment to find the augmenting paths at the end. The time is

$$\begin{aligned} \tilde{O}((nk)n^{2/3}) + \tilde{O}(nv^{3/2}/k^{1/2}) + \sum \tilde{O}(nv) + \tilde{O}(n\sqrt{v} \cdot v/\sqrt{k}) \\ = \tilde{O}(kn^{5/3} + nv^{3/2}/k^{1/2}) \end{aligned}$$

Picking  $k = v/n^{4/9}$  completes the proof. ■

## Appendix B

# Compression of Residual Graphs

As discussed in Section 2.5, the original Benczúr and Karger [BK96] result showed compression. That is, they showed that by sampling with probabilities inversely proportional to strength and increasing the capacity of a chosen edge by one over its sampling probability, with high probability the cuts in the sample have capacity close to their original value. In Chapter 4 we do similar sampling in a residual graph, but prove a weaker result—we only show that the sample contains an augmenting path. In this section we show that it is also possible to compress a residual graph. This result is stronger than what we state in Chapter 4, but the proof is somewhat messier and not necessary for that algorithm, so we have just included it in this appendix for reference.

Compression is more difficult to show for two reasons. First, recall that our proof in Chapter 4 worked by arguing that there would be an edge in some  $k$ -strong component. Naturally we would also expect cut values to be preserved in that  $k$ -strong component, but that is not sufficient. We want to show that cut values are preserved everywhere. The second difficulty is that we cannot appeal to the undirected graph results as cleanly. It would be nice to say that the probability of cut value deviation looks like  $e^{-\sum 1/k(x,y)}$ , as the probability of picking no edges did. But whereas it was easy to analyze the probability of picking no edge even when the sampling probabilities were different, it is less easy to do so when analyzing deviation. Our likely tool for analyzing deviation is the Chernoff bound, and that does not accommodate different weights on the edges so smoothly.

The theorem is as follows.

**Theorem B.0.4** *Let  $\gamma = \frac{256 \ln n \lg^2 m}{\epsilon^2} \frac{v}{v-|f|}$ . Given a unit-capacity graph  $G$  and a flow  $f$ , if each edge  $(x, y)$  of  $G_f$  is chosen with probability  $p(x, y) = \min\{1, \frac{\gamma}{k(x,y)}\}$  and given capacity  $\frac{1}{p(x,y)}$  if chosen, then with high probability the value of every cut in the sample is within a  $(1 \pm 2\epsilon)$  factor of the value of the cut in  $G_f$ .*

Note that we only state the theorem for unit-capacity graphs. It can be applied to integer capacities by treating each edge of capacity  $x$  as  $x$  parallel unit-capacity edges. In fact, the theorem can reasonably be applied to real capacities, because one could round the real capacities off to integers, such that cut values are preserved to within  $\epsilon/2$ , and then sample. Of course

you would still not want to make  $x$  random choices for one edge of capacity  $x$ . But Karger [Kar99] shows how to overcome such problems by instead picking a number from 1 to  $x$  from a suitable distribution.

## B.1 Supporting Definitions and Lemmas

As was the case in Chapter 4, our foundation is Karger and Stein's [KS96] proof that the number of  $\alpha$ -minimum cuts is only  $n^{2\alpha}$ . We will show a related result that takes vertex induced subgraphs into account. We begin with a definition:

**Definition B.1.1** *Two cuts are  $k$ -indistinguishable if they have the same set of nodes incident to  $k$ -strong edges crossing the cut.*

Observe that  $k$ -indistinguishability is an equivalence relation.

**Definition B.1.2** *The value of an equivalence class of  $k$ -indistinguishable cuts is the number of  $k$ -strong edges that cross the cuts in the class.*

Note that the above makes sense, because any two cuts that are  $k$ -indistinguishable have the same  $k$ -strong edges crossing them.

**Lemma B.1.3** *The number of equivalence classes of  $k$ -indistinguishable cuts of value  $l$  is at most  $n^{2l/k}$ .*

This proof is very similar to the one given by Karger and Stein, except that now we look only at the  $k$ -strong edges. This means that we only have a collection of  $k$ -connected subgraphs, not a  $k$ -connected graph, which in turn means that the number of small cuts can increase dramatically. Suppose that  $n/2$  vertices were in their own component. Then for any cut of any size in the remaining vertices, there are  $2^{n/2}$  cuts of that size, because each individual vertex can go on either side. However, these cuts are all  $k$ -indistinguishable, so they are only one equivalence class.

**Proof.** Fix an equivalence class of  $k$ -indistinguishable cuts of value  $l$  and consider running a modified contraction algorithm on the graph of only the  $k$ -strong edges: while there are more than  $2l/k$  vertices, if there is an isolated vertex contract it into a random vertex, otherwise pick a random edge and contract it.

Each time we contract an edge we reduce the number of vertices by 1, and we run the risk of destroying the chosen class. Letting  $m'$  denote the number of remaining edges, the probability the class survives a step is  $(1 - r/m')$ . Contraction can never reduce connectivity, so at all times we know that the degree of each non-isolated vertex is at least  $k$ . This means that when the number of vertices is  $n'$  and none are isolated, we always have  $m' \geq kn'/2$ . So if we never came across the case of an isolated vertex, the probability that the class would survive all the contractions is



$$\prod_{i=n}^{2r/k+1} 1 - \frac{r}{ki/2} = \frac{1}{\binom{n}{2r/k}}$$

At this point only  $2^{2r/k-1}$  classes remain, and since a fixed class survives to this point with at least the above probability, there can only be  $\binom{n}{2r/k} 2^{2r/k-1} \leq n^{2r/k}$  such classes.

Now observe that when an isolated vertex occurs, we get to reduce the number of vertices without risking the equivalence class, so we get a step with survival probability 1. Thus the calculation above is still a lower bound on the survival probability, so the bound on the number of classes holds in general. ■

## B.2 The Proof

We are now ready to prove the theorem.

**Proof.** We will consider a slightly different sampling experiment for which the probability that cut values are preserved is obviously no better than the experiment given in the theorem. The modified experiment is as follows:

Let  $G_0$  be  $G_f$ . For  $i$  from 1 to  $l = \lfloor \lg(m/\gamma) \rfloor$ , let  $k = 2^{i-2}\gamma$  and construct  $G_i$  by taking all of the  $k$ -weak edges from  $G_{i-1}$  and a random sample (with  $p = 1/2$ ) of the  $k$ -strong edges from  $G_{i-1}$  with their capacities doubled.

Observe that in this experiment the probability that an edge is kept is  $2^{-\lfloor \lg(2k(x,y)/\gamma) \rfloor} \leq \gamma/k(x,y)$ , so if the probability is good that this experiment preserves cut values, then the probability is at least as good that our original experiment does.

We will now prove that the new experiment preserves cut values with high probability by induction.

Inductive hypothesis: with high probability, all cuts in  $G_i$  have capacity within  $(1 \pm \epsilon/\lg m)^i$  times their original capacity.

Observe that our final bound comes from  $G_l$ , which will have cut values within  $(1 \pm \epsilon/\lg m)^l$  of their original value. As long as  $\epsilon < 1/2$ , we have

$$\begin{aligned} \left(1 + \frac{\epsilon}{\lg m}\right)^{\lg \frac{m}{\gamma}} &\leq \left(1 + \frac{\epsilon}{\lg m}\right)^{\lg m} \leq e^\epsilon \leq (1 + 2\epsilon) \\ \left(1 - \frac{\epsilon}{\lg m}\right)^{\lg \frac{m}{\gamma}} &\geq \left(1 - \frac{\epsilon}{\lg m}\right)^{\lg m} \geq e^{-2\epsilon} \geq (1 - 2\epsilon) \end{aligned}$$

So cuts are preserved as desired.

We now proceed with the proof by induction.

Base case ( $i = 0$ ):  $G_0 = G_f$ , so it's trivially true.

Inductive step: Suppose the inductive hypothesis holds for  $G_i$ . It suffices to show that with high probability the capacity of every cut in  $G_{i+1}$  is within  $(1 \pm \epsilon/\lg m)$  times the capacity in  $G_i$ .

Observe that all remaining  $k$ -strong edges in  $G_i$  have capacity  $2^i$ . For each equivalence class of  $k$ -indistinguishable cuts in  $G_i$ , associate the cut that has the least  $k$ -weak residual out-bound capacity. The reason for doing this is that the cut with the minimum capacity is the one that would deviate first. That is, the amount of deviation we can tolerate in the sampling of our  $k$ -strong edges is dependent on how many  $k$ -weak edges we can count on being present. Specifically, consider a cut with  $s$  residual capacity and in a class with  $s_1$  residual edges. If the number of edges chosen is within  $(1 \pm \frac{\epsilon s}{2^i s_1 \lg m})$  of the expected value, then since the expected value is  $s_1/2$  and the new weight of each edge will be  $2^{i+1}$ , the deviation in cut value is only  $\epsilon s / \lg m$ . If we consider another cut that uses the same strong edges but has more weak edges, then it will have a larger  $s$ , so this deviation will be acceptable for it as well. Therefore, for a given class, we need only worry about satisfying the cut with the smallest residual capacity.

A Chernoff bound says that if we sample  $s_1$  edges with probability  $p$ , the probability of deviation by  $\delta$  is at most  $e^{-ps_1\delta^2/4}$ . Therefore, the probability that a cut with  $s$  residual edges in a class with  $s_1$  residual edges deviates too much is

$$e^{-\frac{\frac{1}{2}s_1\left(\frac{\epsilon s}{2^i s_1 \lg m}\right)^2}{4}} = e^{-\frac{s\epsilon^2}{8 \cdot 2^i \lg^2 m} \frac{s}{2^i s_1}} \leq e^{-\frac{s\epsilon^2}{8 \cdot 2^i \lg^2 m}}$$

We now have two cases.

### Case 1: $k < 2v$

Since every cut has value at least  $v - |f|$ , the probability of failure in any of the  $n^{4v/k}$  smallest classes is at most

$$n^{4v/k} e^{-\frac{(v-|f|)\epsilon^2}{8 \cdot 2^i \lg^2 m}}$$

Recalling that  $k = 2^{i-2}\gamma$ , and  $\gamma = \frac{256 \ln n \lg^2 m}{\epsilon^2} \frac{v}{v-|f|}$ , we can rewrite this bound as

$$n^{4v/k} e^{-\frac{8v \ln n}{k}} = n^{-4v/k} < 1/n^2$$

We know from inverting Lemma B.1.3 that the  $j^{\text{th}}$  largest class must have value at least  $\frac{k \ln j}{2 \ln n}$ . In particular, the  $n^{4v/k}$ th largest class must have value at least  $2v$ . So the number of residual edges in the associated cut is at least half the original number of edges, which is of course larger than the value of the class. Therefore the probability of failure over all large classes is at most

$$\begin{aligned} \sum_{j > n^{4v/k}} e^{-\frac{\epsilon^2 k \ln j}{32 \cdot 2^i \ln n \lg^2 m}} &= \sum_{j > n^{4v/k}} e^{-\frac{2v \ln j}{v-|f|}} = \sum_{j > n^{4v/k}} j^{-\frac{2v}{v-|f|}} \\ &\leq \int_{j=n^{4v/k}}^{\infty} j^{-\frac{2v}{v-|f|}} dj = \frac{-1}{-\frac{2v}{v-|f|} + 1} n^{\frac{4v}{k} (-\frac{2v}{v-|f|} + 1)} \end{aligned}$$

Since  $(\frac{2v}{v-|f|} - 1)$  is at least one, the total is at most  $n^{-4v/k}$ , which is at most  $1/n^2$ .

**Case 2:**  $k \geq 2v$ 

Since every class has at least  $k$  capacity in the original graph, it has at least  $k/2$  capacity in the residual graph. Since all the cuts we are sampling have at least half their original capacity, this case essentially follows from the fact that sampling in an undirected graph works.

For completeness, we state the details. The probability of failure in any of the  $n^2$  smallest classes is at most

$$n^2 e^{-\frac{k\epsilon^2}{16 \cdot 2^i \lg^2 m}}$$

Again substituting for  $k$  and  $\gamma$ , we get

$$n^2 e^{-\frac{4v \ln n}{v-|f|}} < n^2 n^{-4} = 1/n^2$$

The probability of failure in any of the larger classes is at most

$$\sum_{j>n^2} e^{-\frac{\epsilon^2 k \ln j}{32 \cdot 2^i \ln n \lg^2 m}} = \sum_{j>n^2} e^{-\frac{2v \ln j}{v-|f|}} = \sum_{j>n^2} j^{-\frac{2v}{v-|f|}} \leq \sum_{j>n^2} j^{-2} \leq \int_{j=n^2}^{\infty} j^{-2} dj = 1/n^2$$

Since each step of the induction fails with probability at most  $1/n^2$ , and there are only  $O(\log m)$  steps, the desired result holds with high probability. ■



# Bibliography

- [AMO93] Ravindra K. Ahuja, Thomas L. Magnanti and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.
- [BG97] Michel Buriel and Olivier Goldschmidt. A new and improved algorithm for the 3-cut problem. *Operations Research Letters*, 21:225–227, 1997.
- [BK96] András A. Benczúr and David R. Karger. Approximate s–t min-cuts in  $\tilde{O}(n^2)$  time. In *Proceedings of the 28<sup>th</sup> ACM Symposium on Theory of Computing*, pages 47–55, May 1996.
- [Din70] Efim A. Dinitz. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. *Soviet Mathematics Doklady*, 11:1277–1280, 1970.
- [EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.
- [ET75] Shimon Even and Robert E. Tarjan. Network Flow and Testing Graph Connectivity. *SIAM Journal on Computing*, 4(4):507–518, 1975.
- [FF56] Lester R. Ford, Jr. and Delbert R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.
- [GH94] Oliver Goldschmidt and Dorit Hochbaum. A polynomial algorithm for the k-cut problem for fixed k. *Mathematics of Operation Research*, 19:24–37, 1994.
- [Gol97] Andrew V. Goldberg. Personal communication, October 1997.
- [GR97a] Andrew V. Goldberg and Satish Rao. Beyond the flow decomposition barrier. In *Proceedings of the 30<sup>th</sup> Annual Symposium on the Foundations of Computer Science*, pages 2–11, October 1997.
- [GR97b] Andrew V. Goldberg and Satish Rao. Flows in undirected unit capacity networks. In *Proceedings of the 30<sup>th</sup> Annual Symposium on the Foundations of Computer Science*, pages 32–35, October 1997.
- [GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

- [GY95] Zvi Galil and Xiangdong Yu. Short length versions of Menger's theorem (extended abstract). In *Proceedings of the 27<sup>th</sup> ACM Symposium on Theory of Computing*, pages 499–508, May 1995.
- [HdLT98] Jacob Holm, Kristian de Lichtenberg and Mikkel Thorup. Poly-logarithmic deterministic fully-dynamic graph algorithms for connectivity, minimum spanning tree, 2-edge and biconnectivity. In *Proceedings of the 29<sup>th</sup> ACM Symposium on Theory of Computing*, pages 79–89, May 1998.
- [HKR97] Monika Rauch Henzinger, Jon Kleinberg and Satish Rao. Short-length Menger theorems. Technical Report 1997-022, Digital Systems Research Center, 1997.
- [HO94] Jianxiu Hao and James B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. *Journal of Algorithms*, 17(3):424–446, 1994.
- [HS85] Dorit S. Hochbaum and David B. Shmoys. An  $O(|V|^2)$  algorithm for the planar 3-cut problem. *SIAM Journal on Algebraic and Discrete Methods*, 6(4):707–712, 1985.
- [Kap96] Sanjiv Kapoor. On minimum 3-cuts and approximating k-cuts using cut trees. In *Proceedings of the 5<sup>th</sup> Conference on Integer Programming and Combinatorial Optimization*, pages 132–146, June 1996.
- [Kar73] Alexander V. Karzanov. O nakhozhenii maksimal'nogo potoka v setyakh spetsial'nogo vida i nekotorykh prilozheniyakh. In *Matematicheskie Voprosy Upravleniya Proizvodstvom*, volume 5. Moscow State University Press, Moscow, 1973. In Russian; title translation: On Finding Maximum Flows in a Network with Special Structure and Some Applications.
- [Kar97] David R. Karger. Using random sampling to find maximum flows in uncapacitated undirected graphs. In *Proceedings of the 29<sup>th</sup> ACM Symposium on Theory of Computing*, pages 240–249, May 1997.
- [Kar98] David R. Karger. Better random sampling algorithms for flows in undirected graphs. In *Proceedings of the 9<sup>th</sup> Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 490–499, January 1998.
- [Kar99] David R. Karger. Random sampling in cut, flow, and network design problems. *Mathematics of Operations Research*, 24(2):383–413, 1999.
- [Kar00] David R. Karger. Minimum cuts in near-linear time. *Journal of the ACM*, 47(1):46–76, 2000.
- [KL98] David R. Karger and Matthew S. Levine. Finding maximum flows in simple undirected graphs seems faster than bipartite matching. In *Proceedings of the 29<sup>th</sup> ACM Symposium on Theory of Computing*, pages 69–78, May 1998.
- [KL02] David R. Karger and Matthew S. Levine. Random sampling in residual graphs. In *Proceedings of the 33<sup>rd</sup> ACM Symposium on Theory of Computing*, May 2002.

- [KRT94] Valerie King, Satish Rao and Robert E. Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17(3):447–474, 1994.
- [KS96] David R. Karger and Clifford Stein. A new approach to the minimum cut problem. *Journal of the ACM*, 43(4):601–640, 1996.
- [KWY97a] Yoko Kamidoi, Shin'ichi Wakabayashi and Noriyoshi Yoshida. Faster algorithms for finding a minimum  $k$ -way cut in a weighted graph. In *Proceedings of the IEEE International Symposium on Circuits and Systems*, pages 1009–1012, June 1997.
- [KWY97b] Yoko Kamidoi, Shin'ichi Wakabayashi and Noriyoshi Yoshida. A new approach to the minimum  $k$ -way partition problem for weighted graphs. Technical Report COMP97-25, Institute of Electronics, Information and Communication Engineers, 1997.
- [Lev00] Matthew S. Levine. Fast randomized algorithms for computing minimum  $\{3,4,5,6\}$ -way cuts. In *Proceedings of the 11<sup>th</sup> Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 735–742, January 2000.
- [NI92a] Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge connectivity in multigraphs and capacitated graphs. *SIAM Journal on Discrete Mathematics*, 5(1):54–66, 1992.
- [NI92b] Hiroshi Nagamochi and Toshihide Ibaraki. Linear time algorithms for finding  $k$ -edge connected and  $k$ -node connected spanning subgraphs. *Algorithmica*, 7:583–596, 1992.
- [NI99] Hiroshi Nagamochi and Toshihide Ibaraki. A fast algorithm for computing minimum 3-way and 4-way cuts. In *Proceedings of the 7<sup>th</sup> Conference on Integer Programming and Combinatorial Optimization*, pages 377–390, June 1999.
- [NKI99] Hiroshi Nagamochi, Shigeki Katayama and Toshihide Ibaraki. Faster algorithm for computing minimum 5-way and 6-way cuts. In *5th Annual International Conference on Computing and Combinatorics*, pages 164–173, July 1999.
- [NNI94] Hiroshi Nagamochi, Kazuhiro Nishimura and Toshihide Ibaraki. Computing all small cuts in an undirected network. Technical Report 94007, Kyoto University, 1994.
- [Que] Maurice Queyranne. Personal communication with David Karger.
- [ST83] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.
- [SV95] Huzur Saran and Vijay V. Vazirani. Finding  $k$  cuts within twice the optimal. *SIAM Journal on Computing*, 24(1):101–108, 1995.
- [VY92] Vijay V. Vazirani and Mihalis Yannakakis. Suboptimal cuts: Their enumeration, weight, and number. In *Proceedings of the International Colloquium on Automata, Languages and Programming*, pages 366–377, July 1992.

- [ZNI01] Liang Zhao, Hiroshi Nagamochi and Toshihide Ibaraki. Approximating the minimum  $k$ -way cut in a graph via minimum 3-way cuts. *Journal of Combinatorial Optimization*, 5:397–410, 2001.