A New Approach to the Minimum Cut Problem

David R. Karger[*]
Department of Computer Science
Stanford University
karger@cs.stanford.edu

Clifford Stein[†]
Department of Mathematics and Computer Science
Dartmouth College
cliff@cs.dartmouth.edu

August 6, 1996

Abstract This paper presents a new approach to finding minimum cuts in undirected graphs. The fundamental principle is simple: the edges in a graph's minimum cut form an extremely small fraction of the graph's edges. Using this idea, we give a randomized, strongly polynomial algorithm that finds the minimum cut in an arbitrarily weighted undirected graph with high probability. The algorithm runs in $O(n^2 \log^3 n)$ time, a significant improvement over the previous $\tilde{O}(mn)$ time bounds based on maximum flows. It is simple and intuitive and uses no complex data structures. Our algorithm can be parallelized to run in $\mathcal{RNC}$ with n^2 processors; this gives the first proof that the minimum cut problem can be solved in $\mathcal{RNC}$. The algorithm does more than find a single minimum cut; it finds all of them.

With minor modifications, our algorithm solves two other problems of interest. Our algorithm finds all cuts with value within a multiplicative factor of α of the minimum cut's in expected $\tilde{O}(n^{2\alpha})$ time, or in $\mathcal{RNC}$ with $n^{2\alpha}$ processors. The problem of finding a minimum multiway cut of a graph into r pieces is solved in expected $\tilde{O}(n^{2(r-1)})$ time, or in $\mathcal{RNC}$ with $n^{2(r-1)}$ processors. The "trace" of the algorithm's execution on these two problems forms a new compact data structure for representing all small cuts and all multiway cuts in a graph. This data structure can be efficiently transformed into the more standard *cactus* representation for minimum cuts.

1 Introduction

1.1 The Problem

This paper studies the minimum cut problem. Given an undirected graph with n vertices and m (possibly weighted) edges, we wish to partition the vertices into two non-empty sets so as to minimize the number (or total weight) of edges crossing between them. More formally, a *cut* (A, B) of a graph G is a partition of the vertices of G into two nonempty sets A and B. An edge (v, w) *crosses* cut (A, B) if one of v and w is in A and the other in

[*]Supported by a National Science Foundation Graduate Fellowship, by NSF Grant CCR-9010517, NSF Young Investigator Award CCR-9357849, and grants from Mitsubishi Corporation and OTL.

[†]Support provided by a Walter Burke Research Initiation Award, a Dartmouth College Research Initiation Award and NSF Research Initiation Award CCR-9308701.

B. The *value* of a cut is the number of edges that cross the cut or, in a weighted graph, the sum of the weights of the edges that cross the cut. The minimum cut problem is to find a cut of minimum value.

Throughout this paper, the graph is assumed to be connected, since otherwise the problem is trivial. We also require that all edge weights be non-negative because otherwise the problem is $\mathcal{NP}$-complete by a trivial transformation from the maximum-cut problem [GJ79, page 210]. We distinguish the minimum cut problem from the *s-t minimum cut problem* in which we require that two specified vertices s and t be on opposite sides of the cut; in the *minimum cut problem* there is no such restriction.

Particularly on unweighted graphs, solving the minimum cut problem is sometimes referred to as finding the *connectivity* of a graph, that is, determining the minimum number of edges (or minimum total edge weight) that must be removed to disconnect the graph.

1.2 Applications

The minimum cut problem has many applications, some of which are surveyed by Picard and Queyranne [PQ82]. We discuss others here.

The problem of determining the connectivity of a network arises frequently in issues of network design and network reliability [Col87]: in a network with random edge failures, the network is most likely to be partitioned at the minimum cuts. For example, consider an undirected graph in which each edge fails with some probability p, and suppose we wish to determine the probability that the graph becomes disconnected. Let f_k denote the number of edge sets of size k whose removal disconnects the graph. Then the graph disconnection probability is $\sum_k f_k p^k (1-p)^{m-k}$. If p is very small, then the value can be accurately approximated by considering f_k only for small values of k. It therefore becomes important to to enumerate all minimum cuts and, if possible, all nearly minimum cuts [RC87]. In a more recent application [Kar95], this enumeration is used in a fully polynomial time approximation scheme for the all terminal network reliability problem.

In information retrieval, minimum cuts have been used to identify clusters of topically related documents in hypertext systems [Bot93]. If the links in a hypertext collection are treated as edges in a graph, then small cuts correspond to groups of documents that have few links between them and are thus likely to be unrelated.

Minimum cut problems arise in the design of compilers for parallel languages [Cha94]. Consider a parallel program which we are trying to execute on a distributed memory machine. In the *alignment distribution graph* for this program, vertices correspond to program operations and edges corresponds to flows of data between program operations. When the program operations are distributed among the processors, the edges connecting nodes on different processors are "cut." These cut edges are bad because they indicate a need for interprocessor communication. Finding an optimum layout of the program operations requires repeated solution of minimum cut problems in the alignment distribution graph.

Minimum cut problems also play an important role in large-scale combinatorial optimization. Currently the best methods for finding exact solutions to large traveling salesman problems are based on the technique of *cutting planes.* The set of feasible traveling salesman tours in a given graph induces a convex polytope in a high-dimensional vector space. Cutting plane algorithms find the optimum tour by repeatedly generating linear inequalities that cut off undesirable parts of the polytope until only the optimum tour remains. The inequalities that have been most useful are *subtour elimination constraints*, first introduced by Dantzig, Fulkerson and Johnson [DFJ54]. The problem of identifying a subtour elimination constraint can be rephrased as the problem of finding a minimum cut in a graph with real-valued edge weights. Thus, cutting plane algorithms for the traveling salesman problem must solve a large number of minimum cut problems (see [LLKS85] for a survey of the area). Padberg

minimum cut bounds		unweighted		weighted	
		undirected	directed	undirected	directed
sequential time	previous	$c^2 n \log \frac{n}{c}$	$cm \log \frac{n^2}{m}$	$mn + n^2 \log n$	$mn \log \frac{n^2}{m}$
		[Gab95]		[NI92]	[HO94]
	this paper	$n^2 \log^3 n$		$n^2 \log^3 n$	
processors used in $\mathcal{RNC}$	previous	$n^{4.37}$ [KUW86, GP88]		Unknown	$\mathcal{P}$-complete [GSS82]
	this paper	n^2		n^2	

Figure 1: Bounds For the Minimum Cut Problem

and Rinaldi [PR90] recently reported that the solution of minimum cut problems was the computational bottleneck in their state-of-the-art cutting-plane based algorithm. They also reported that minimum cut problems are the bottleneck in many other cutting-plane based algorithms for combinatorial problems whose solutions induce connected graphs. Applegate et al. [App92, ABCC95] made similar observations and also noted that an algorithm to find *all nearly minimum* cuts might be even more useful.

1.3 History

Several different approaches to finding minimum cuts have been investigated. Until recently, the most efficient algorithms used maximum flow computations. As the fastest known algorithms for maximum flow take $\Omega(mn)$ time, the best minimum cut algorithms inherited this bound. Recently, new and slightly faster approaches to computing minimum cuts without computing maximum flows have appeared. Parallel algorithms for the problem have also been investigated, but until now processor bounds have been quite large for unweighted graphs, and no good algorithms for weighted graphs were known.

Previously best results, together with our new bounds, are summarized in Figure 1, where c denotes the value of the minimum cut.

1.3.1 Flow based approaches

The first algorithm for finding minimum cuts used the duality between s-t minimum cuts and s-t maximum flows [FF56, EFS56]. Since an s-t maximum flow saturates every s-t minimum cut, it is straightforward to find an s-t minimum cut given a maximum flow—for example, the set of all vertices reachable from the source in the residual graph of a maximum flow forms one side of such an s-t minimum cut. An s-t maximum flow algorithm can thus be used to find an s-t minimum cut, and minimizing over all $\binom{n}{2}$ possible choices of s and t yields a minimum cut. In 1961, Gomory and Hu [GH61] introduced the concept of a *flow equivalent tree* and observed that the minimum cut could be found by solving only $n-1$ maximum flow problems. In their classic book *Flows in Networks* [FF62], Ford and Fulkerson comment on the method of Gomory and Hu:

> Their procedure involved the successive solution of precisely $n-1$ maximal flow problems. Moreover, many of these problems involve smaller networks than the original one. Thus one could hardly ask for anything better.

This attitude was prevalent in the following 25 years of work on minimum cuts. The focus in minimum cut algorithms was on developing better maximum flow algorithms and better methods of performing series of maximum flow computations.

Maximum flow algorithms have become progressively faster over the years. Currently, the fastest algorithms are based on the push-relabel method of Goldberg and Tarjan [GT88]. Their early implementation of this method runs in $O(nm\log(n^2/m))$ time. Incrementally faster implementations appeared subsequently. Currently, the fastest deterministic algorithms, independently developed by King, Rao and Tarjan [KRT94] and by Phillips and Westbrook [PW92]) run in $O(nm(\log_{\frac{m}{n\log n}} n))$ time. Randomization has not helped significantly. The fastest randomized maximum flow algorithm, developed by Cheriyan, Hagerup and Mehlhorn [CH95] runs in expected $O(mn + n^2\log^2 n)$ time. Finding a minimum cut by directly applying any of these algorithms in the Gomory-Hu approach requires $\Omega(mn^2)$ time.

There have also been successful efforts to speed up the series of maximum flow computations that arise in computing a minimum cut. The basic technique is to pass information among the various flow computations, so that computing all n maximum flows together takes less time than computing each one separately. Applying this idea, Podderyugin [Pod73], Karzanov and Timofeev [KT86], and Matula [Mat87] independently discovered several algorithms which determine edge connectivity in unweighted graphs in $O(mn)$ time. Hao and Orlin [HO94] obtained similar types of results for weighted graphs. They showed that the series of $n-1$ related maximum flow computations needed to find a minimum cut can all be performed in roughly the same amount of time that it takes to perform one maximum flow computation, provided the maximum flow algorithm used is a non-scaling push-relabel algorithm. They used the fastest such algorithm, that of Goldberg and Tarjan, to find a minimum cut in $O(mn\log(n^2/m))$ time.

1.3.2 Cuts without flows

Recently, two approaches to finding minimum cuts *without* computing any maximum flows have appeared. One approach, developed by Gabow [Gab95], is based on a matroid characterization of the minimum cut problem. According to this characterization, the minimum cut in a graph is equal to the maximum number of disjoint directed spanning trees that can be found in it. Gabow's algorithm finds the minimum cut by finding such a maximum packing of trees. This approach finds the minimum cut of an unweighted graph in $O(cm\log(n^2/m))$ time, where c is the value of the minimum cut. Although flows are not used, the trees are constructed through a sequence of augmenting path computations. Rather than computing the minimum cut directly, Gabow's algorithm computes a flow-like structure that saturates the minimum cut of the graph. In [Kar94a], randomization is used to speed up Gabow's algorithm to run in $\tilde{O}(m\sqrt{c})$ time with high probability, where $\tilde{O}(f)$ denote $O(f \operatorname{polylog} f)$.

The second new approach bears some similarity to our own work, as it uses no flow-based techniques at all. The central idea is to repeatedly identify and *contract* edges that are not in the minimum cut until the minimum cut becomes apparent. It applies only to undirected graphs, but they may be weighted. Nagamochi and Ibaraki [NI92] give a procedure called *scan-first search* that identifies and contracts an edge that is not in the minimum cut in $O(m + n\log n)$ time. This yields an algorithm that computes the minimum cut in $O(mn + n^2\log n)$ time. Stoer and Wagner [SW94] subsequently gave a simplified version of the Nagamochi and Ibaraki algorithm with the same running time (this simplification was subsequently discovered independently by Frank [Fra94]). Scan-first search is also used by Gabow [Gab95] to improve the running time of his matroid algorithm to $O(m+c^2 n\log(n/c))$ on undirected graphs. Matula [Mat93] uses scan-first search in an algorithm that approximates the minimum cut to within a multiplicative factor of $(2+\epsilon)$ in $O(m)$ time.

1.3.3 Parallel algorithms

Parallel algorithms for the minimum cut problem have also been explored, though with much less satisfactory results. For undirected and unweighted graphs, Khuller and Schieber [KS91] gave an algorithm that uses cn^2 processors to find a minimum cut of value c in $\tilde{O}(c)$ time; this algorithm is therefore in $\mathcal{RNC}$ when c is polylogarithmic in n ($\mathcal{RNC}$ is the class of problems that can be solved by a randomized algorithm in polylogarithmic time using a PRAM with a polynomial number of processors). For directed unweighted graphs, the $\mathcal{RNC}$ matching algorithms of Karp, Upfal, and Wigderson [KUW86] or Mulmuley, Vazirani, and Vazirani [MVV87] can be combined with a reduction of s-t maximum flow to matching [KUW86] to yield $\mathcal{RNC}$ algorithms for s-t minimum cuts. We can find a minimum cut by performing n of these s-t cut computations in parallel (number the vertices, and find a minimum $v_i, v_{(i+1) \bmod n}$-cut for each i). Unfortunately, the processor bounds are quite large—the best bound, using Galil and Pan's [GP88] adaptation of [KUW86], is $n^{4.37}$.

These unweighted directed graph algorithms can be extended to work for weighted graphs by treating an edge of weight w as a set of w parallel edges. If W is the sum of all the edge weights then the number of processors needed is proportional to W; hence the problem is not in $\mathcal{RNC}$ unless the edge weights are given in unary. If we combine these algorithms with the scaling techniques of Edmonds and Karp [EK72], as suggested in [KUW86], the processor count is $mn^{4.37}$ and the running times are proportional to $\log W$. Hence, the algorithms are not in $\mathcal{RNC}$ unless $W = n^{\log^{O(1)} n}$.

The lack of an $\mathcal{RNC}$ algorithm is not surprising. Goldschlager, Shaw, and Staples [GSS82] showed that the s-t minimum cut problem on weighted directed graphs is $\mathcal{P}$-complete. In Section 6.5 we note a simple reduction to their result that proves that the weighted directed minimum cut problem is also $\mathcal{P}$-complete. Therefore, a (randomized) parallel algorithm for the directed minimum cut problem would imply that $\mathcal{P} \subseteq \mathcal{NC}$ ($\mathcal{RNC}$), which is believed to be unlikely.

1.4 Our Contribution

We present a new approach to the minimum cut problem that is entirely independent of maximum flows. Our randomized *Recursive Contraction Algorithm* runs in $O(n^2 \log^3 n)$ time—a significant improvement on the previous $\tilde{O}(mn)$ bounds. It is slightly faster for a class of graphs that includes all planar graphs. The algorithm is *strongly polynomial*—that is, number of operations it performs can be bounded by a polynomial independent of the size of the input numbers. With high probability (that is, with probability exceeding $1 - 1/n$ on problems of size n) it finds the minimum cut—in fact, it finds *all* minimum cuts. This suggests that the minimum cut problem may be fundamentally easier to solve than the maximum flow problem. The parallel version of our algorithm runs in polylogarithmic time using n^2 processors on a PRAM. It thus provides the first proof that the minimum cut problem with arbitrary edge weights can be solved in $\mathcal{RNC}$. It is also an *efficient* $\mathcal{RNC}$ algorithm for the minimum cut problem in that the total work it performs is within a polylogarithmic factor of that performed by the best sequential algorithm (namely, the one presented here). In a contrasting result, we show that the directed minimum cut problem is $\mathcal{P}$-complete and thus appears unlikely to have an $\mathcal{RNC}$ solution.

Our algorithm is extremely simple and, unlike the best flow-based approaches, does not rely on any complicated data structures such as dynamic trees [ST83]. The most time consuming steps of the sequential version are simple computations on arrays, while the most time consuming steps in the parallel version are sorting and computing connected components. All of these computations can be performed practically and efficiently.

With slight modifications, the Recursive Contraction Algorithm can be used to compute minimum multi-way cuts. The *minimum r-way cut problem* is to find a minimum weight set of edges whose removal partitions a given graph into r separate components. Previously, the best known sequential bound, due to Goldschmidt and Hochbaum [GH88], was $O(n^{r^2/2-r+11/2})$, and no parallel algorithm was known. Our algorithm runs in expected $\tilde{O}(n^{2(r-1)})$ time, and in $\mathcal{RNC}$ using $n^{2(r-1)}$ processors. This shows that the minimum r-way cut problem is in $\mathcal{RNC}$ for any constant r. In contrast, it is shown in [DJP+94] that the multiway cut problem in which r specified vertices are required to be separated (*i.e.*, a generalization of the s-t minimum cut problem) is $\mathcal{NP}$-complete for any $r > 2$.

An *approximately minimum cut* is one whose value is within a constant factor of the value of the minimum cut. Our algorithm can be used to compute all approximately minimum cuts in polynomial time and in $\mathcal{RNC}$. There was no previously known algorithm for doing so. One application is to the identification of violated *comb inequalities* [ABCC95]—another important class of cutting planes for the Traveling Salesman Problem.

Our analysis can also be used to give new theorems regarding the structure and enumeration of approximately minimal and multiway cuts. In particular, we give tight bound on the number of approximately minimum and multiway cuts in a graph. These results have important applications in the study of network reliability [RC87]. They have also been used in the development of fast algorithms for approximate solutions to minimum cut, maximum flow, and other graph problems [Kar94c, Kar94a].

A minor modification of our algorithm lets us use it to construct the *cactus representation* of minimum cuts introduced in [DKL76]. We improve the sequential time bound of this construction to $\tilde{O}(n^2)$. We give the first $\mathcal{RNC}$ algorithm for weighted graphs, improving the previous (unweighted graph) processor bound from $mn^{4.5}$ to n^4.

A drawback of our algorithms is that they are *Monte Carlo.* Monte Carlo algorithms give the right answer with high probability but not with certainty. For many problems, such a flaw can be rectified because it is possible to verify a "certificate" of the correctness of the output and rerun the algorithm if the output is wrong. This turns the Monte Carlo algorithms into *Las Vegas* algorithms that are guaranteed to produce the right answer but have a small probability of taking a long time to do so. Unfortunately, all presently known minimum cut certificates (such as maximum flows, or the complete intersections of [Gab95]) take just as long to construct when the minimum cut is known as when it is unknown. Thus we can provide no speedup if a guarantee of the minimum cut value is desired.

The original Contraction Algorithm with an $\tilde{O}(mn^2)$ running time and processor bound, as well as the connections to multiway and approximately minimum cuts and analyses of network reliability, originally appeared in [Kar93]. The improved algorithm with faster running times and processor bounds originally appeared in [KS93]. This paper combines results from those two conference papers. Lomonosov [Lom94] independently developed some of the basic intuitions leading to the Contraction Algorithm, using them to investigate questions of network reliability.

1.5 Related Work

Subsequent to the initial presentation of this work [Kar93, KS93], several related papers based upon it have appeared. Karger [Kar94a] used the Contraction Algorithm to prove theorems about the structure and enumeration of near-minimum cuts. These have led to a random-sampling approach to cut problems. [Kar94c] shows how to approximate the minimum cut to within any constant factor in $O(m + n\log^2 n)$ time sequentially, and to within a factor of 2 in parallel using a linear number of processors. Algorithms for dynamically maintaining approximate minimum cuts during edge insertions and deletions are also presented. [Kar94a] gives an $\tilde{O}(m\sqrt{c})$-time Las Vegas algorithm for finding minimum cuts in unweighted undirected graphs. It also gives techniques for solving to other cut-related problems such as network design. Most recently, [Kar96] has given sampling-based minimum cut algorithms with running times of $O(n^2 \log n)$ and $O(m \log^3 n)$. Karger and Benczur [BK96] have given fast sampling-based algorithms for approximating s-t minimum cuts. In [Kar95], the structure of minimum cuts is used to obtain bounds on the reliability of a network with random edge failures.

Karger and Motwani [KM96] have shown that in fact the minimum cut problem for weighted graphs is in $\mathcal{NC}$. Rather than derandomizing the algorithms presented here, they develop a new algorithm based on the combinatorial aspects of minimum cuts that follow from this work. Benczur [Ben94] has used the Contraction Algorithm to get improved sequential and parallel algorithms for augmenting the connectivity of a graph to a given value.

1.6 Presentation Overview

The starting point of our work is an abstract formulation of the *Contraction Algorithm* in Section 2. This extremely simple algorithm has an $\Omega(1/n^2)$ probability of outputting a minimum cut. It is based on the observation that the edges of a graph's minimum cut form a very small fraction of the graph's edges, so that a randomly selected edge is unlikely to be in the minimum cut. Therefore, if we choose an edge at random and *contract* its endpoints into a single vertex, the probability is high that the minimum cut will be unaffected. We therefore find the minimum cut by repeatedly choosing and contracting random edges until the minimum cut is apparent.

Moving from the abstract formulation to a more concrete algorithm divides naturally into two stages. In the first stage, we show how to efficiently implement the repeated selection and contraction of edges which forms a single trial of the Contraction Algorithm. Section 3 uses a simple adjacency matrix scheme to implement the algorithm in $O(n^2)$ time.

The second stage deals with the need for multiple trials of the Contraction Algorithm. Given the $\Omega(1/n^2)$ success probability of the Contraction Algorithm, repeating it $O(n^2 \log n)$ times gives a high probability of finding the minimum cut in some trial. However, this yields undesirably high sequential time and parallel processor bounds of $\tilde{O}(n^4)$. Thus in Section 4 we show how the necessary $O(n^2 \log n)$ trials can share their work so that the total work performed by any one trial is $\tilde{O}(1)$. This gives our $\tilde{O}(n^2)$ sequential time bounds. In Section 5, we describe certain modifications needed to make the algorithm strongly polynomial, in particular with respect to its use of random bits.

We then give parallel implementations of the Contraction Algorithm. To achieve parallelism, we "batch together" numerous selections and contractions, so that only a few contraction phases are necessary. We present a simple but slightly inefficient (by logarithmic factors) parallel implementation in Section 6. This suffices to show that minimum cuts in undirected graphs can be found in $\mathcal{RNC}$. In contrast, in Section 6.5 we show that the corresponding directed graph problem is $\mathcal{P}$-complete.

In section 7 we give an improved implementation of the Contraction Algorithm that runs in linear time sequentially and is more efficient in parallel than our previous implementation. This gives us improved sequential time bounds on certain classes of graphs and a more efficient parallel algorithm.

In Sections 8 and 9 we show how to find minimum multiway cuts and approximate minimum cuts. In Section 10 we discuss the cactus representation for minimum cuts [DKL76], and show how the Contraction Algorithm leads to more efficient algorithms for constructing it. In Section 11 we discuss how to trade time for space, showing that we can still match the $\tilde{O}(mn)$ time bounds of previous minimum cut algorithms, even if our computational space is restricted to $O(n)$.

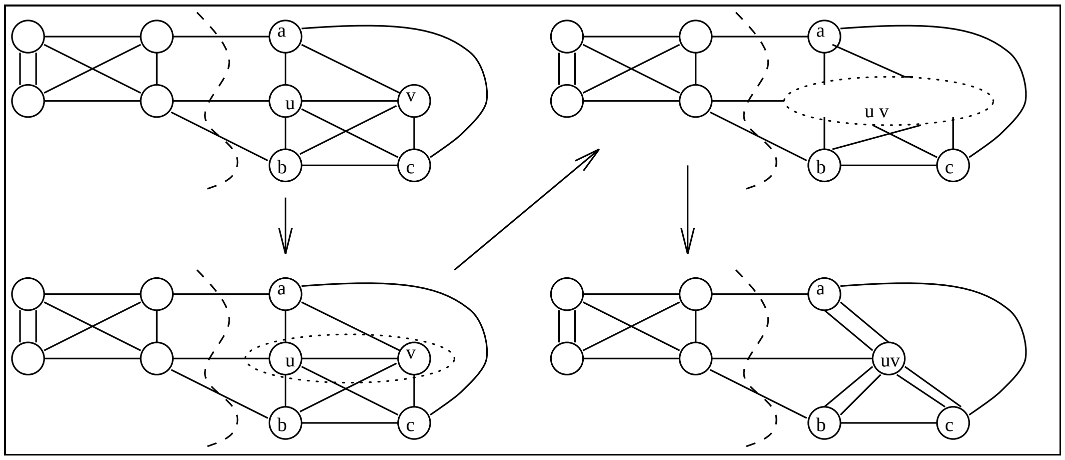

Figure 2: contraction

2 The Contraction Algorithm

In this section we restrict our attention to unweighted multigraphs (*i.e.*, graphs that may have multiple edges between one pair of vertices), and present an abstract version of the Contraction Algorithm. This version of the algorithm is particularly intuitive and easy to analyze. In later sections, we will describe how to implement it efficiently.

The Contraction Algorithm uses one fundamental operation, *contraction* of graph vertices. To contract two vertices v_1 and v_2 we replace them by a vertex v and let the set of edges incident on v be the union of the sets of edges incident on v_1 and v_2. We do not merge edges from v_1 and v_2 that have the same other endpoint; instead, we create multiple instances of those edges. However, we remove self loops formed by edges originally connecting v_1 to v_2. Formally, we delete all edges (v_1, v_2), and replace each edge (v_1, w) or (v_2, w) with an edge (v, w). The rest of the graph remains unchanged. We will use $G/(v, w)$ to denote graph G with edge (v, w) contracted (by *contracting an edge*, we will mean contracting the two endpoint of the edge). Extending this definition, for an edge set F we will let G/F denote the graph produced by contracting all edges in F (the order of contractions is irrelevant up to isomorphism). An example of an edge contraction is given in Figure 2.

Assume initially that we are given a multigraph $G(V, E)$ with n vertices and m edges. The Contraction Algorithm is based on the idea that since the minimum cut is small, a randomly chosen edge is unlikely to be in the minimum cut. The Contraction Algorithm, which is described in Figure 3, repeatedly chooses an edge at random and contracts it.

```
Procedure Contract($G$)

repeat until $G$ has 2 vertices

    choose an edge $(v, w)$ uniformly at random from $G$
    let $G \leftarrow G/(v, w)$

return $G$
```

Figure 3: The Contraction Algorithm

When the Contraction Algorithm terminates, each original vertex has been contracted into one of the two remaining "metavertices." This defines a cut of the original graph: each side corresponds to the vertices contained in one of the metavertices. More formally, at any point in the algorithm, we can define $s(a)$ to be the set of original vertices contracted to a current metavertex a. Initially $s(v) = v$ for each $v \in V$, and whenever we contract (v, w) to create vertex x we set $s(x) = s(v) \cup s(w)$. We say a cut (A, B) in the contracted graph *corresponds to* a cut (A', B') in G, where $A' = \cup_{a \in A} s(a)$ and $B' = \cup_{b \in B} s(b)$. Note that a cut and its corresponding cut will have the same value, where we define the value of a cut to be the sum of the weights of the edges crossing the cut.

When the Contraction Algorithm terminates, yielding a graph with two metavertices a and b, we have a corresponding cut (A, B) in the original graph, where $A = s(a)$ and $B = s(b)$.

Lemma 2.1 *A cut (A, B) is output by the Contraction Algorithm if and only if no edge crossing (A, B) is contracted by the algorithm.*

Proof: The only if direction is obvious. For the other direction, consider two vertices on opposite sides of the cut (A, B). If they end up in the same metavertex, then there must be a path between them consisting of edges that were contracted. However, any path between them crosses (A, B), so an edge crossing cut (A, B) would have had to be contracted. This contradicts our hypothesis. □

Lemma 2.1 is also the basis of Nagamochi and Ibaraki's min-cut algorithm [NI92]. They give a linear-time deterministic algorithm for identifying and contracting a non-minimum-cut edge. Doing this n times (for a total running time of $O(mn)$) yields two vertices which by Lemma 2.1 define the minimum cut of the graph.

Theorem 2.2 *A particular minimum cut in G is returned by the Contraction Algorithm with probability at least $\binom{n}{2}^{-1} = \Omega(n^{-2})$.*

Proof: Fix attention on some specific minimum cut (A, B) with c crossing edges. We will use the term *minimum cut edge* to refer only to edges crossing (A, B). From Lemma 2.1, we know that if we never select a minimum cut edge during the Contraction Algorithm, then the two vertices we end up with must define the minimum cut.

Observe that after each contraction, the minimum cut of the new graph must still be at least c. This is because every cut in the contracted graph corresponds to a cut of the same value in the original graph, and thus has value at least c. Furthermore, if we contract an edge (v, w) that does not cross (A, B), then the cut (A, B) corresponds to a cut of value c in $G/(v, w)$; this corresponding cut is a minimum cut (of value c) in the contracted graph.

Each time we contract an edge, we reduce the number of vertices in the graph by one. Consider the stage in which the graph has r vertices. Since the contracted graph has a minimum cut value of at least c, it must have minimum degree at least c, and thus at least $rc/2$ edges. However, only c of these edges are in the minimum cut. Thus, a randomly chosen edge is in the minimum cut with probability at most $2/r$. The probability that we never contract a minimum cut edge through all $n - 2$ contractions is thus at least

$$\begin{aligned}\left(1 - \frac{2}{n}\right)\left(1 - \frac{2}{n-1}\right)\cdots\left(1 - \frac{2}{3}\right) &= \left(\frac{n-2}{n}\right)\left(\frac{n-3}{n-1}\right)\cdots\left(\frac{2}{4}\right)\left(\frac{1}{3}\right) \\ &= \binom{n}{2}^{-1} \\ &= \Omega(n^{-2}).\end{aligned}$$

□

This bound is tight. In the graph consisting of a cycle on n vertices, there are $\binom{n}{2}$ minimum cuts, one for each pair of edges in the graph. Each of these minimum cuts is produced by the Contraction Algorithm with equal probability, namely $\binom{n}{2}^{-1}$.

An alternative interpretation of the Contraction Algorithm is that we are randomly ranking the edges and then constructing a minimum spanning tree of the graph based on these ranks (using Kruskal's minimum spanning tree algorithm [Kru56]). If we remove the heaviest edge in the minimum spanning tree, the two components that result have an $\Omega(n^{-2})$ chance of defining a particular minimum cut. This intuition forms the basis of the implementation in Section 7.

The Contraction Algorithm can be halted when k vertices remain. We refer to this as *contraction to k vertices*. The following result is an easy extension of Theorem 2.2:

Corollary 2.3 *A particular minimum cut (A, B) survives contraction to k vertices with probability at least $\binom{k}{2}/\binom{n}{2} = \Omega((k/n)^2)$.*

2.1 Weighted Graphs

Extending the Contraction Algorithm to weighted graphs is simple. For a given integer-weighted graph G, we consider a corresponding unweighted multigraph G' on the same set of vertices. An edge of weight w in G is mapped to a collection of w parallel unweighted edges in G'. The minimum cuts in G and G' are the same, so it suffices to run the Contraction Algorithm on G'. We choose a pair of vertices to contract in G' by selecting an edge of G' uniformly at random. Therefore, the probability that we contract u and v is proportional to the number of edges connecting u and v in G', which is just the weight of the edge (u, v) in G. This leads to the weighted version of the Contraction Algorithm given in Figure 4.

```
Procedure Contract($G$)

repeat until $G$ has 2 vertices
    choose an edge $(v, w)$ with probability proportional to the weight of $(v, w)$
    let $G \leftarrow G/(v, w)$
return $G$
```

Figure 4: The Weighted Contraction Algorithm

The analysis of this algorithm follows immediately from the unweighted case. The analysis also applies to graphs with non-integral edge weights.

Corollary 2.4 *The Weighted Contraction Algorithm outputs a particular minimum cut of G with probability $\Omega(1/n^2)$.*

3 Implementing the Contraction Algorithm

We now turn to implementing the algorithm described abstractly in the previous section. First, we give a version that runs in $O(n^2)$ time and space. Later, we shall present a version that runs in $O(m)$ time and space with high probability, and is also parallelizable. This first method, though, is easier to analyze, and its running time does not turn out to be the dominant factor in our analysis of the running time of our overall algorithm.

To implement the Contraction Algorithm we use an $n \times n$ weighted adjacency matrix, which we denote by W. The entry $W(u, v)$ contains the weight of edge (u, v), which can equivalently be viewed as the number of multigraph edges connecting u and v. If there is no edge connecting u and v then $W(u, v) = 0$. We also maintain the total (weighted) degree $D(u)$ of each vertex u; thus $D(u) = \sum_v W(u, v)$.

We now show how to implement two steps: randomly selecting an edge and performing a contraction.

3.1 Choosing an Edge

A fundamental operation that we need to implement is the selection of an edge with probability proportional to its weight. A natural method is the following. First, from edges $e_1, \ldots, e_m$ with weights $w_1, \ldots, w_m$, construct *cumulative weights* $W_k = \sum_{i=1}^{k} w_i$. Then choose an integer r uniformly at random from $0, \ldots, W_m$ and use binary search to identify the edge e_i such that $W_{i-1} \leq r < W_i$. This can easily be done in $O(\log W)$ time. While this is not a strongly polynomial bound since it depends on the edge weights being small, we will temporarily ignore this issue. For the time being, we assume that we have a black-box subroutine called `Random-Select`. The input to `Random-Select` is a cumulative weight array of length m. `Random-Select` runs in $O(\log m)$ time and returns an integer between 1 and m, with the probability that i is returned being proportional to w_i. In practice the lack of strong polynomiality is irrelevant since implementors typically pretend that their system-provided random number generator can be made to return numbers in an arbitrarily large range by scaling. We will provide theoretical justification for using `Random-Select` by giving a strongly polynomial implementation of it in Section 5. Note that the input weights to `Random-Select` need not be edge weights, but are simply arbitrary measures of proportionality.

We now use `Random-Select` to find an edge to contract. Our goal is to choose an edge (u, v) with probability proportional to $W(u, v)$. To do so, choose a first endpoint u with probability proportional to $D(u)$, and then once u is fixed choose a second endpoint v with probability proportional to $W(u, v)$. Each of these two choices requires $O(n)$ time to construct a cumulative weight array plus one $O(\log n)$-time call to `Random-Select`, for a total time bound of $O(n)$.

The following lemma, similar to one used by Klein, Plotkin, Stein and Tardos [KPST94], proves the correctness of this procedure.

Lemma 3.1 *If an edge is chosen as described above, then* $\Pr[(u,v)$ *is chosen*$]$ *is proportional to* $W(u, v)$.

Proof: Let $\sigma = \sum_v D(v)$. Then

$$\begin{aligned}
\Pr[\text{choose}(u,v)] &= \Pr[\text{choose } u] \cdot \Pr[\text{choose } (u,v) \mid \text{chose } u] \\
&\quad + \Pr[\text{choose } v] \cdot \Pr[\text{choose } (u,v) \mid \text{chose } v] \\
&= \frac{D(u)}{\sigma} \cdot \frac{W(u,v)}{D(u)} + \frac{D(v)}{\sigma} \cdot \frac{W(u,v)}{D(v)} \\
&= \frac{2W(u,v)}{\sigma} \\
&\propto W(u,v).
\end{aligned}$$

□

3.2 Contracting an Edge

Having shown how to choose an edge, we now show how to implement a contraction. Given W and D, which represent a graph G, we explain how to update W and D to reflect the contraction of a particular edge (u, v). Call the new graph G' and compute its representation via the algorithm of Figure 3.2. Intuitively, this algorithm moves all edges incident on v to u. The algorithm replaces row u with the sum of row u and row v, and replaces column u with the sum of column u and column v. It then clears row v and column v. W and D now

```
Procedure to contract edge $(u,v)$
Let $D(u) \leftarrow D(u) + D(v) - 2W(u,v)$
Let $D(v) \leftarrow 0$
Let $W(u,v) \leftarrow W(v,u) \leftarrow 0$
For each vertex $w$ except $u$ and $v$
        Let $W(u,w) \leftarrow W(u,w) + W(v,w)$
        Let $W(w,u) \leftarrow W(w,u) + W(w,v)$
        Let $W(v,w) \leftarrow W(w,v) \leftarrow 0$
```

Figure 5: Contracting an Edge

represent G', since any edge that was incident to u or v is now incident to u and any two edges of the form (u,w) and (v,w) for some w have had their weights added. Furthermore, the only vertices whose total weighted degrees have changed are u and v, and $D(u)$ and $D(v)$ are updated accordingly. Clearly, this procedure can be implemented in $O(n)$ time. Summarizing this and the previous section, we have shown that in $O(n)$ time we can choose an edge and contract it. This yields the following result:

Corollary 3.2 *The Contraction Algorithm can be implemented to run in $O(n^2)$ time.*

Observe that if the Contraction Algorithm has run to completion, leaving just two vertices u and v, then we can determine the weight of the implied cut by inspecting $W(u,v)$. In order to contract to k vertices we only need to perform $n-k \leq n$ edge-contractions.

For the rest of this paper, we will use the Contraction Algorithm as a subroutine, `Contract`(G,k), that accepts a weighted graph G and a parameter k and, in $O(n^2)$ time, returns a contraction of G to k vertices. With probability at least $\binom{k}{2}/\binom{n}{2}$ (Corollary 2.3), a particular minimum cut of the original graph will be preserved in the contracted graph. In other words, no vertices on opposite sides of this minimum cut will have been merged, so there will be a minimum cut in the contracted graph corresponding to the particular minimum cut of the original graph.

We can in fact implement the Contraction Algorithm using only $O(m)$ space (in the worst case $m = \Theta(n^2)$, but for sparse graphs using this approach will save space). We do so by maintaining an adjacency list representation. All the edges incident to vertex v are in a linked list. In addition, we have pointers between the two copies of the same edge (v,w) and (w,v). When v and w are merged, we traverse the adjacency list of v, and for each edge (v,u) find the corresponding edge (u,v) and rename it to (u,w). Note that as a result of this renaming the adjacency lists will not be sorted. However, this is easy to deal with. Whenever we choose to merge two vertices, we can merge their adjacency lists by using a bucket sort into n buckets based on the edges' other endpoints; the time for this merge thus remains $O(n)$ and the total time for the algorithm remains $O(n^2)$.

4 The Recursive Contraction Algorithm

The Contraction Algorithm can be used by itself as an algorithm for finding minimum cuts. Recall that the Contraction Algorithm has an $\Omega(n^{-2})$ probability of success. We can therefore repeat the Contraction Algorithm $\Theta(n^2 \log n)$ times, and output the smallest cut produced by any of runs of the Contraction Algorithm. The only way this procedure can fail to find the minimum cut is if all $cn^2 \ln n$ runs of the Contraction Algorithm fail to find the minimum cut, but we can upper bound the probability that this occurs by

$$\left(1 - \frac{1}{n^2}\right)^{cn^2 \ln n} \leq e^{c \ln n} \leq n^c.$$

Thus we will find the minimum cut with high probability. However, the resulting sequential running time of $\tilde{O}(n^4)$ is excessive. We therefore show how to wrap the Contraction Algorithm within the *Recursive Contraction Algorithm.* The idea of this new algorithm is to share the bulk of the work among the $O(n^2 \log n)$ Contraction Algorithm trials so as to reduce the total work done.

We begin with some intuition as to how to speed up the Contraction Algorithm. Consider the contractions performed in one trial of the Contraction Algorithm. The first contraction has a reasonably low probability of contracting an edge in the minimum cut, namely $2/n$. On the other hand, the last contraction has a much higher probability of contracting an edge in the minimum cut, namely $2/3$. This suggests that the Contraction Algorithm works well initially, but has poorer performance later on. We might improve our chances of success if, after partially contracting the graph, we switched to a (possibly slower) algorithm with a better chance of success on what remains.

One possibility is to use one of the deterministic minimum cut algorithms, such as that of [NI92], and this indeed yields some improvement. However, a better observation is that an algorithm that is more likely to succeed than the Contraction Algorithm is *two* trials of the Contraction Algorithm.

This suggests the Recursive Contraction Algorithm described in Figure 6. As can be seen, we perform two independent trials. In each, we first partially contract the graph, but not so much that the likelihood of the cut surviving is too small. By contracting the graph until it has $\lceil n/\sqrt{2}+1\rceil$ vertices, we ensure a greater than 50% probability of not contracting a minimum cut edge, so we expect that on the average one of the two attempts will avoid contracting a minimum cut edge. We then recursively apply the algorithm to each of the two partially contracted graphs. As described, the algorithms returns only a cut value; it can easily be modified to return a cut of the given value. Alternatively, we might want to output every cut encountered, hoping to enumerate all the minimum cuts.

```
Algorithm Recursive-Contract($G, n$)

input A graph $G$ of size $n$.

if $G$ has fewer than 6 vertices

then
        $G' \leftarrow$ Contract($G, 2$)
      return the weight of cut ($A = s(a), B = s(b)$) in $G'$

else repeat twice
            $G' \leftarrow$ Contract($G, \lceil n/\sqrt{2} + 1\rceil$)
            Recursive-Contract($G', \lceil n/\sqrt{2} + 1\rceil$).
        return the smaller of the two resulting values.
```

Figure 6: The Recursive Contraction Algorithm

We now analyze the running time of this algorithm.

Lemma 4.1 *Algorithm* `Recursive-Contract` *runs in $O(n^2 \log n)$ time.*

Proof: One level of recursion consists of two independent trials of contraction of G to $\lceil n/\sqrt{2}+1\rceil$ vertices followed by a recursive call. Performing a contraction to $\lceil n/\sqrt{2}+1\rceil$ vertices can be implemented by Algorithm `Contract` from Section 3 in $O(n^2)$ time. We thus have the following recurrence for the running time:

$$T(n) = 2\left(n^2 + T\left(\lceil n/\sqrt{2}+1\rceil\right)\right). \tag{1}$$

This recurrence is solved by

$$T(n) = O(n^2 \log n).$$

□

Lemma 4.2 *Algorithm* `Recursive-Contract` *uses $O(n^2)$ or $O(m\log(n^2/m))$ space (depending on the implementation).*

Proof: We have to store one graph at each level of the recursion. The size of the graph at the k^{th} level is described by the recurrence $n_1 = n;\ \ n_{k+1} = \lceil n_k/\sqrt{2}+1\rceil$. If we use the original $O(n^2)$-space formulation of the Contraction Algorithm, then the space required is $O(\sum_k n_k^2) = O(n^2)$. To improve the space bound, we can use the linear-space variant of procedure `Contract`. Since at each level the graph has no more than $\min(m, n_k^2)$ edges and can be stored using $O(\min(m, n_k^2))$ space, the total storage needed is $\sum_k O(\min(m, n_k^2)) = O(m\log(n^2/m))$. □

This analysis shows why the running time of the Contraction Algorithm is not the bottleneck in the Recursive Contraction Algorithm. We shall later present a linear time (in the number of edges) implementation of the Contraction Algorithm. However, since the recurrence we formulate must apply to the contracted graphs as well, there is no *a priori* bound on the number of edges in the graph we are working with. Therefore n^2 is the only bound we can put on the number of edges in the graph, and thus on the time needed to perform a contraction to $\lceil n/\sqrt{2}+1\rceil$ vertices. Furthermore, the existence of n^2 leaves in the recursion tree gives a lower bound of $\Omega(n^2)$ on the running time of `Recursive-Contract`, regardless of the speed of `Contract`. This is why the linear-time implementation of `Contract` that we shall give in Section 6 provides no speedup in general.

We now analyze the probability that the algorithm finds the particular minimum cut we are looking for. We will say that the Recursive Contraction Algorithm *finds* a certain minimum cut if that minimum cut corresponds to one of the leaves in the computation tree of the Recursive Contraction Algorithm. Note that if the algorithm finds any minimum cut then it will certainly output some minimum cut.

Lemma 4.3 *The Recursive Contraction Algorithm finds a particular minimum cut with probability $\Omega(1/\log n)$.*

Proof: Suppose that a particular minimum cut has survived up to some particular node in the recursion tree. It will survive to a leaf below that node if two criteria are met: it must survive one of the graph contractions at this node, and it must be found by the recursive call following that contraction. Each of the two branches thus has a success probability equal to the product of the probability that the cut survives the contraction and the probability that the recursive call finds the cut. The probability that the cut survives the contraction is, by Corollary 2.3, at least

$$\frac{(\lceil n/\sqrt{2}+1\rceil)(\lceil n/\sqrt{2}+1\rceil - 1)}{n(n-1)} \geq 1/2.$$

In the base case the probability that a cut survives is at least $\frac{1}{15}$. This yields a recurrence $P(n)$ for a lower bound on the probability of success on a graph of size n:

$$P(n) \geq \begin{cases} 1 - \left(1 - \frac{1}{2}P\left(\lceil n/\sqrt{2} + 1\rceil\right)\right)^2 & \text{if } n \geq 7 \\ \frac{1}{15} & \text{otherwise} \end{cases} \tag{2}$$

We solve this recurrence through a change of variables. Let p_k be the probability of success of a problem on the k^{th} level of recursion, where a leaf has level 0. Then the recurrence above can be rewritten and simplified as

$$\begin{aligned} p_0 &= 1/15 \\ p_{k+1} &\geq 1 - \left(1 - \tfrac{1}{2}p_k\right)^2 \\ &= p_k - \tfrac{1}{4}p_k^2. \end{aligned}$$

Let $z_k = 4/p_k - 1$, so $p_k = 4/(z_k + 1)$. Substituting this in the above recurrence and solving for z_{k+1} yields

$$\begin{aligned} z_0 &= 59 \\ z_{k+1} &= z_k + 1 + 1/z_k. \end{aligned}$$

Since clearly $z_k \geq 1$, it follows by induction that

$$k < z_k < 59 + 2k$$

Thus $z_k = \Theta(k)$ and $p_k = 4/(z_k + 1) = \Theta(1/k)$. The depth of recursion is $2\log_2 n + O(1)$. Hence it follows that

$$P(n) \geq p_{2\log_2 n + O(1)} = \Theta(1/\log n).$$

In other words, one trial of the Recursive Contraction Algorithm finds any particular minimum cut with probability $\Omega(1/\log n)$.

We note that the cutoff at problems of size 7 is just for ease of analysis. The same bounds hold, up to constant factors, if the algorithm is run until a 2-vertex graph remains, or if a deterministic algorithm is used on the problems of size 7. □

Those familiar with branching processes might see that we are evaluating the probability that the extinction of contracted graphs containing the minimum cut does not occur before depth $2\log n$.

Theorem 4.4 *All minimum cuts in an arbitrarily weighted undirected graph with n vertices and m edges can be found with high probability in $O(n^2 \log^3 n)$ time and $O(m \log(n^2/m))$ space.*

Proof: It is known ([DKL76], see also Lemma 8.4) that there are at most $\binom{n}{2}$ minimum cuts in a graph. Repeating `Recursive-Contract` $O(\log^2 n)$ times gives an $O(1/n^4)$ chance of missing any particular minimum cut. Thus our chance of missing any one of the at most $\binom{n}{2}$ minimum cuts is upper bounded by $O(\binom{n}{2}/n^4) = O(1/n^2)$. □

It is noteworthy that unlike the best algorithms for maximum flow this algorithm uses straightforward data structures. The algorithm has proven to be practical and easy to code.

We can view the running of the Recursive Contraction Algorithm as a binary computation tree, where each vertex represents a graph with some of its edges contracted and each edge represents a contraction by a factor of approximately $\sqrt{2}$. A leaf in the tree is a contracted graph with 2 metavertices and defines a cut, potentially a minimum cut. The depth of this tree is $2\log_2 n + O(1)$, and it thus has $O(n^2)$ leaves. This shows that the improvement over the direct use of n^2 trials of the Contraction Algorithm comes not from generating a narrower tree (those trials also define a "tree" of depth 1 with n^2 leaves), but from being able to amortize the cost of the contractions used to produce a particular leaf.

If it suffices to output only one minimum cut, then we can keep track of the smallest cut encountered as the algorithm is run and output it afterwards in $O(n)$ time by unraveling the sequence of contractions that led to it. If we want to output all the minimum cuts, then this might in fact become the dominant factor in the running time: there could be n^2 such cuts, each requiring $O(n)$ time to output as a list of vertices on each side of the cut. This is made even worse by the fact that some minimum cuts may be produced many times by the algorithm. Applegate [App92, ABCC95] observed that there is a simple hashing technique that can be used to avoid outputting a cut more than once. At the beginning, assign to each vertex a random $O(\log n)$-bit key. Whenever two vertices are merged by contractions, combine their keys with an exclusive-or. At a computation leaf in which there are only two vertices, the two keys of those vertices form an identifier for the particular cut that has been found. With high probability, no two distinct cuts we find will have the same identifiers. Thus by checking whether an identifier has already been encountered we can avoid outputting any cut that has already been output.

An alternative approach to outputting all minimum cuts is to output a concise representation of them; this issue is taken up in Section 10.

In [Kar93], several simple implementation of the Contraction Algorithm for unweighted multigraphs were given. However, in the context of the Recursive Contraction Algorithm the unweighted graph algorithms are no longer useful. This is because our time bound depends on the many subproblems deep in the recursion tree being small. The contractions reduce the number of vertices in the subproblems, but need not reduce the number of edges. If we worked with multigraphs, it is entirely possible that each of the $O(n^2)$ subproblems on 4 vertices would have n^2 edges, causing the algorithm to be slow. Using a weighted graph algorithm, it becomes possible to merge parallel edges, thus ensuring that every k vertex graph we encounter has at most k^2 edges.

5 Strongly Polynomial Random Selection

In this section, we finish showing how the Recursive Contraction Algorithm can be implemented in the claimed time bound by implementing the procedure `Random-Select`. The input to `Random-Select` is an array W of length n. This *cumulative weight array* is constructed from n weights w_i by setting $W_k = \sum_{i \leq k} w_i$. Procedure `Random-Select` implements the goal of choosing an index i at random with probability proportional to weight w_i. While the analysis of this section is necessary to prove the desired time bound of `Recursive-Contract`, it is unlikely that it would be necessary to actually implement the procedure `Random-Select` in practice. The system supplied random number generator and rounding will probably suffice.

This problem of *nonuniform selection* is not new. It has been known for some time [KY76] that the fastest possible algorithm for nonuniform random selection has expected running time proportional to the *entropy* of the distribution being sampled; this section essentially uses similar techniques to get high probability amortized bounds.

Let $M = W_n$ be the sum of all weights. If the edge weights w_i (and thus the total weight M) are polynomial in n, then we use standard methods to implement Procedure `Random-Select` in $O(\log n)$ time: generate a uniformly distributed $(\log M)$-bit number k in the range $[0, M]$ (all logs are base 2), and return the value i such that $W_{i-1} \leq k < W_i$. This can be done even in the model where only a single random bit, rather than an $O(\log n)$-bit random number, can be generated in unit time.

When the weights are arbitrary integers that sum to M, the time needed for an exact implementation is $\Omega(\log M)$. However, we can modify the algorithm to introduce a negligible error and run in $O(\log n)$ time. Suppose we know that only t calls to `random-select` will be made during the running of our algorithm. To select an edge from the cumulative distribution, even if the sum of the edge weights is superpolynomial in n, we let $N = tn^4$, generate s uniformly at random from $[0, N]$, and choose the edge i such that $W_{i-1} < W_n s/N < W_i$. The edge that we choose differs from the one that we would have chosen using exact arithmetic only if $W_n s/N$ and $W_n(s+1)/N$ specify different indices. But there can only be at most n such values in the "boundaries" of different indices, so there are at most n values that we could chose for s that would cause an error. Thus the probability that we make an error with one selection is less than $n/N = O(1/tn^3)$ and the probability that we make any errors is $O(1/n^3)$. This approach reflects what is typically done in practice—we simply use the random number generator available in a system call, perform rounding, and ignore the possible loss of precision that results.

A drawback of this approach in theory is that even if a particular input to `Random-Select` has only two choices, we still need to use $\Omega(\log t)$ bits to generate a selection. Using this approach adds an extra $\log n$ factor to the running time of `Random-Select` on constant size inputs (which arise at the leaves of the recursion tree of our algorithm) and thus increases the running time of `Recursive-Contract`.

A better approach is the following. Intuitively, we generate the $\log M$ random bits needed to select uniformly from the range $[0, M]$, but stop generating bits when all possible outcomes of the remaining bits yield the same selection. Given the length n input, partition the range $[0, M]$ into $2n$ equal sized intervals of length $M/2n$. Use $1 + \log n$ random bits to select one of the intervals uniformly at random—this requires $O(\log n)$ time spent in binary search among the cumulative weights. If this interval does not contain any of the cumulative weight values W_i (which happens with probability $1/2$, since at most n of the $2n$ intervals can contain one of the cumulative weight values), then we have unambiguously selected a particular index because the values of the remaining bits in the $(\log M)$-bit random number are irrelevant. If the interval contains one or more of the cumulative values, then divide this one interval into $2n$ equal sized subintervals and again use $1 + \log n$ bits to select one subinterval. If the subinterval contains a cumulative weight value, then we subdivide again. Repeat this process until an index is unambiguously selected. Each subdivision requires $O(\log n)$ time and $O(\log n)$ random bits, and successfully identifies an index with probability $1/2$.

Lemma 5.1 *On an input of size n, the expected time taken by* `Random-Select` *is $O(\log n)$. The probability the* `Random-Select` *takes more than $t \log n$ time to finish is $O(2^{-t})$.*

Proof: Each binary search to select a subinterval requires $O(\log n)$ time. Call an interval search a *success* if it selects a unique index, and a *failure* if it must further subdivide an interval. The probability of a success is then at least $1/2$. The total number of interval searches is therefore determined by how many failures occur before a success. Since each search fails with probability at most $1/2$, the probability that t failures occur before a success is $O(2^{-t})$ and the expected number of failures preceding the first success is at most 2. □

Remark: Inspection of the above lemma shows that `Random-Select` can also be implemented by generating one random bit at a time (rather than $\log M$) and stopping when the selected interval is unambiguous. □

Lemma 5.2 *Suppose that t calls are made to* `Random-Select` *on inputs of size n. Then with probability $1 - e^{-\Omega(t)}$, the amortized time for each call is $O(\log n)$.*

Proof: Each interval search in a call requires $O(\log n)$ time. It therefore suffices to prove that the amortized number of interval searches used is $O(1)$, *i.e.* that the total number is $O(t)$. We use the definitions of success and failure from the previous lemma. We know the number of successes over the t calls to `Random-Select` is t, since each success results in the termination of one call. The total number of searches is therefore determined by how many trials occur before the t^{th} success. This number is simply the *negative binomial distribution* for the t^{th} success with probability 1/2. Since the chances of success and failure are equal, we expect to see roughly the same number of successes as failures, namely t, for a total of $2t$ trials. The Chernoff bound (cf. [Mul94, page 427]) proves the probability that the number of trials exceeds $3t$ is exponentially small in t. $\square$

Theorem 5.3 *If n calls are made to* `Random-Select` *and each input is of size $n^{O(1)}$, then with high probability in n the amortized time for* `Random-Select` *on an input of size s is $O(\log s)$.*

Proof: Let the i^{th} input have size n_i and let $t_i = \lceil \log n_i \rceil$. From above, we know that the expected time to run `Random-Select` on input i is $O(t_i)$. We need to show that the total time to run `Random-Select` on all the problems is $O(\sum t_i)$ with high probability. Note that the largest value of t_i is $O(\log n)$.

Call the i^{th} call to `Random-Select` *typical* if there are more than $5 \log n$ calls with the same value t_i and *atypical* otherwise. Since the largest value of t_i is $O(\log n)$, there can be only $O(\log^2 n)$ atypical calls. For the i^{th} atypical call, by Lemma 5.1 and the fact that $t_i = O(\log n)$, we know that the time for call i is $O(\log^2 n)$ with high probability. Thus the time spent in all the atypical calls is $O(\log^4 n)$ with high probability. By Lemma 5.2, if i is a typical call then its amortized cost is $O(t_i)$ with high probability in n. Therefore, the total time spent on all calls is $O(\log^4 n + \sum t_i)$, which is $O(n + \sum t_i)$. Since there are n calls made, the amortized cost for call i is $O(1 + t_i) = O(\log n_i)$. $\square$

We have therefore shown how to implement `Random-Select` in $O(\log n)$ amortized time on size n inputs, assuming a simple condition on the inputs. To see that this condition is met in the Recursive Contraction Algorithm, note that we perform $\Omega(n)$ calls to `Random-Select` (for example, the ones in the two calls to `Contract` at the top level of the recursion), while the largest input is of size n (since no graph we contract has more vertices). This concludes the proof of the time bound of the Recursive Contraction Algorithm.

6 A Parallel Implementation

We now show how the Recursive Contraction Algorithm can be implemented in $\mathcal{RNC}$. To do so, we give an m processor $\mathcal{RNC}$ implementation of the `Contract` by eliminating the apparently sequential nature of the selection and contraction of edges. Parallelizing `Recursive-Contract` is then easy.

As a first step, we will show how a series of selections and contractions needed for the Contraction Algorithm can be implemented in $\tilde{O}(m)$ time. The previous $O(n^2)$ time bound arose from a need to update the graph after each contraction. We circumvent this problem by grouping series of contractions together and performing them all simultaneously. As before, we focus initially on unweighted multigraphs. We start by giving our algorithms as sequential ones, and then show how they can be parallelized.

6.1 Using A Permutation of the Edges

We reformulate the Contraction Algorithm as follows. Instead of choosing edges one at a time, we begin by generating a random permutation L of the edges according to the uniform distribution. Imagine contracting edges in the order in which they appear in the permutation, until only two vertices remain. This is clearly equivalent to the abstract formulation of the Contraction Algorithm. We can immediately deduce that with probability $\Omega(n^{-2})$, a random permutation will yield a contraction to two vertices which determine a particular minimum cut.

Given a random permutation L of the edges, contracting the edges in the order specified by the permutation until two vertices remain corresponds to identifying a prefix L' of L such that contracting the edges in L' yields a graph with exactly two vertices. Equivalently, we are looking for a prefix L' of edges such that the graph $H = (V, L')$ has exactly two connected components. Binary search over L can identify this prefix, because any prefix that is too short will yield more than two connected components while any prefix that is too long will yield only one. The correct prefix can therefore be determined using $O(\log m)$ connected component computations, each requiring $O(m + n)$ time. The total running time of this algorithm (given the permutation) is therefore $O(m \log m)$.

We can improve this running time by reusing information between the different connected component computations. Given the initial permutation L, we first use $O(m + n)$ time to identify the connected components induced by the first $m/2$ edges. If exactly two connected components are induced, we are done. If only one connected component is induced, then we can discard the last $m/2$ edges because the desired prefix ends before the middle edge, and recurse on the first half of L. If more than two connected components are induced, then we can contract the first $m/2$ edges all at once in $O(m)$ time by finding the connected components they induce and relabeling the last $m/2$ edges according to the connected components, producing a new, $m/2$ edge graph on which we can continue the search. Either way we have reduced the number of edges to $m/2$ in $O(m + n)$ time. Since the graph is assumed to be connected, we know that $n \leq m$ as m decreases. Therefore, if we let $T(m)$ be the time to execute this procedure on a graph with m edges, then $T(m) \leq T(m/2) + O(m)$, which has solution $T(m) = O(m)$. In Figure 7 we formally define this `Compact` subroutine. `Compact` takes a parameter k describing the goal number of vertices. Our running time analysis assumes that k is two. Clearly, running times do not increase when k is larger. Recall the notation G/F that denotes the result of contracting graph G by edge set F. We extend this definition as follows. If E is a set of edges in G, then E/F denotes a corresponding set of edges in G/F: an edge $\{v, w\} \in E$ is transformed in E/F to an edge connecting the vertices containing v and w in G/F. Constructing E/F requires merging edges with identical endpoints. Since each endpoint is an integer between 1 and n, we can use a linear-time sorting algorithm, such as bucket sort, to merge edges. Thus `Compact` runs in $O(m)$ time.

```
Compact($G, L, k$)

input: A graph $G$, list of edges $L$, and parameter $k$

if $G$ has $k$ vertices or $L = \emptyset$
then
    return $G$
else
    Let $L_1$ and $L_2$ be the first and second halves of $L$
    Find the connected components in graph $H = (V, L_1)$
    if $H$ has fewer than $k$ components
    then
        return Compact($G, L_1, k$)
    else
        return Compact($G/L_1, L_2/L_1, k$).
```

Figure 7: Procedure Compact

6.2 Generating Permutations using Exponential Variates

The only remaining issue is how to generate the permutation of edges that is used as the list L in `Compact`. To show how the permutation generation can be accomplished in $\mathcal{RNC}$, we give in this section an approach to the problem that is easy to explain but gives somewhat worse than optimal bounds in both theory and practice. In Section 7, we describe a more efficient (and practical) but harder to analyze approach.

For unweighted graphs, a simple method is to assign each edge a score chosen uniformly at random from the unit interval, and then to sort the edges according to score. To extend this approach to weighted graphs, we use the equivalence between an edge of weight w in a weighted graph and a set of w parallel edges in the natural corresponding unweighted multigraph. We use the term *multiedge* to mean an edge of the multigraph corresponding to the weighted graph, and simulate the process of generating a random permutation of the multiedges. The entire multiedge permutation is not necessary in the computation, since as soon as a multiedge is contracted, all the other multiedges with the same endpoints vanish. In fact, all that matters is the earliest place in the permutation that a multiedge with particular endpoints appears. This information suffices to tell us in which order vertices of the graph are merged: we merge u and v before x and y precisely when the first (u, v) multiedge in the permutation precedes the first (x, y) multiedge in the permutation. Thus our goal is to generate an edge permutation whose distribution reflects the order of first appearance of endpoints in a uniform permutation of the corresponding multigraph edges.

As in the unweighted case, we can consider giving each multiedge a score chosen uniformly at random from a large ordered set and then sorting according to score. In this case, the first appearance in the permutation of a multiedge with w copies is determined by the minimum of w randomly chosen scores. We can therefore generate an appropriately distributed permutation of the weighted edges if we give an edge of weight w the minimum of w randomly chosen scores and sort accordingly.

Consider multiplying each edge weight by some value k, so that an edge of weight w corresponds to wk multiedges. This scales the value of the minimum cut without changing its structure. Suppose we give each multiedge a score chosen uniformly at random from the continuous interval $[0, k]$. The probability distribution for the minimum score X among wk edges is then

$$\Pr[X > t] = (1 - t/k)^{wk}.$$

If we now let k become arbitrarily large, the distribution converges to one in which an edge of weight w receives a score chosen from the exponential distribution

$$\Pr[X > t] = e^{-wt}.$$

Thus, assuming we can generate an exponential random variable in $O(1)$ time, then we can generate a permutation in $O(m)$ time. As in the unweighted case, we do not actually have to sort based on the scores: once scores are assigned we can use median finding to split the edge list as needed by `Compact` in $O(m)$ time. If all we have is coin flips, it is possible to use them to sample from an approximately exponential distribution in polylogarithmic time and introduce a negligible error in the computation. As we shall be describing a better method later, we only sketch the details of this approach. Perhaps the simplest way to generate a variable X with probability density function e^{-wt} is to generate a variable U uniformly distributed in the $[0, 1]$ interval, and then to set $X = -(\ln U)/w$. Two obstacles arise in practice. One is that we cannot sample uniformly from $[0, 1]$. Instead, we choose an integer $M = n^{O(1)}$, select uniformly from the integers in $[1, M]$ using $O(\log n)$ random bits, and then divide by M. This gives us an approximation U' to the uniform distribution. Another obstacle is that we cannot exactly compute logarithms. Instead, given U', we use the first $O(\log n)$ terms of the Taylor expansion of the natural logarithm to compute an approximation to $\ln U'$. This gives us an approximation X' to the desired exponential distribution. It is now straightforward to show that with high probability, the permutation that results from these approximate values is exactly the same as it would be if we were using exact arithmetic and continuous distributions. We summarize this in the following lemma:

Lemma 6.1 *In $\log^{O(1)} m$ time per edge, it is possible to assign to each edge an approximately exponentially distributed score that, with high probability, yields the same results in* `Compact` *as if we had used exact exponential distributions.*

An alternative scheme due to Von Neumann [Neu51] generates a random variate with the exact exponential distribution in constant expected time given a uniform random number generator. Details can be found in [Kar94b].

6.3 Parallelizing the Contraction Algorithm

Parallelizing the previous algorithms is simple. To generate the permutation, given a list of edges, we simply assign one processor to each edge and have it generate the (approximately) exponentially distributed score for that edge in polylogarithmic time. We then use a parallel sorting algorithm on the resulting scores. Given the permutation, it is easy to run `Compact` in parallel. $\mathcal{RNC}$ algorithms for connected components exist that use $m/\log n$ processors and run in $O(\log n)$ time on a CRCW PRAM [SV82] or even on the EREW PRAM [HZ94]. Procedure `Compact`, which terminates after $O(\log n)$ iterations, is thus easily seen to be parallelizable to run in $O(\log^2 n)$ time using m processors. As a result, we have the following:

Theorem 6.2 *The Contraction Algorithm can be implemented to run in $\mathcal{RNC}$ using m processors on an m edge graph.*

Using the linear-processor $\mathcal{RNC}$ implementation of `Contract`, we can give an efficient $\mathcal{RNC}$ algorithm for the minimum cut problem. Consider the computation tree generated by Recursive-Contract. The sequential algorithm examines this computation tree using a depth-first traversal of the tree nodes. To solve the problem in parallel, we instead use a breadth-first traversal. The subroutine `Contract` has already been parallelized. We can therefore evaluate our computation tree in a breadth-first fashion, taking only polylogarithmic time to advance one level. Since the depth of the tree is logarithmic, and since the total size of all subproblems at a particular level of the tree is $O(n^2)$, we deduce:

Theorem 6.3 *The minimum cut problem can be solved in $\mathcal{RNC}$ using n^2 processors.*

The space required is now the space needed to store the entire tree. The sequential running-time recurrence $T(n)$ also provides a recursive upper bound on the space needed to store the tree. Thus the space required is $O(n^2 \log^3 n)$ (on the assumption that we perform all $O(\log^2 n)$ trials of the Recursive Contraction Algorithm in parallel).

6.4 Maximizing PRAM Speed

If speed is of the utmost importance, we can decrease the parallel running time to $O(\log n)$ on unweighted graphs, even on an EREW PRAM. We modify the original implementation of a single trial of the Contraction Algorithm. Recall that in the case of an unweighted graph, a permutation of the edges can be generated in $O(\log n)$ time by assigning a random score to each edge and sorting. After generating the permutation, instead of using `Compact` to identify the correct permutation prefix, we examine all prefixes in parallel. Each prefix requires a single connected components computation, which can be performed in $O(\log n)$ time, even on an EREW PRAM, using $m/\log n$ processors [HZ94]. We can therefore perform a single trial of the Contraction Algorithm in $O(\log n)$ time using m^2 processors. As was mentioned in the overview, running this algorithm $n^2 \log n$ times in parallel yields the minimum cut with high probability. All of this takes $O(\log n)$ time. This is in fact the best possible asymptotic running time, since even distinguishing whether a graph is connected (positive connectivity) or unconnected (0 connectivity) takes $\Omega(\log n)$ time ([HZ94], based on a reduction from [CDR86]—a similar lower bound of $\Omega(\log n/\log\log n)$ for the CRCW model follows from [Has89]). However, the processor bounds are quite large.

6.5 Related Problem are $\mathcal{P}$-complete

The previous section indicates a distinction between minimum cut problems on directed and undirected graphs. In a directed graph, the s-t minimum cut problem is the problem of finding a partition of the vertices into two sets S and T, with $s \in S$ and $t \in T$, such that the weight of edges going from S to T is minimized. Note that the weights of edges going from T to S is not counted in the value of the cut. The s-t minimum cut problem on directed graphs was shown to be $\mathcal{P}$-complete [GSS82]. A similar result holds for the global minimum cut problem:

Lemma 6.4 *The global minimum cut problem is $\mathcal{P}$-complete for directed graphs.*

Proof: Given an algorithm the finds global minimum cuts, we find a minimum s-t cut as follows. We add, for each vertex v, directed edges of infinite weight from t to v and from v to s. The global minimum cut in this modified graph must now have $s \in S$ and $t \in T$, for otherwise some of the edges of infinite weight will appear in the cut. Hence the global minimum cut must be a minimum s-t cut of the original graph. □

A different reduction [PR75] transforms a directed minimum s-t cut problem into an undirected minimum s-t cut problem. It follows that the undirected s-t minimum cut problem is $\mathcal{P}$-complete as well.

7 A Better Implementation

We now discuss a conceptually more complicated (but still easy to implement) version of the Contraction Algorithm based on permutations. It has several advantages, both theoretical and practical, over the exponential variates approach. First, it does not need to generate exponential variates. Although we have argued that such a computation can be done in theory, in practice both approaches we described are more expensive than generating uniform random numbers. Second, the sequential implementation runs in linear time. As we have discussed, this will not produce any improvement in the worst-case running time of the Recursive Contraction Algorithm on arbitrary graphs, since such graphs might have $\Omega(n^2)$ edges. However, it does give a slightly improved time bounds for finding minimum cuts in certain classes of sparse graphs. Yet another advantage is that it uses $O(m)$ space without using the pointers and linked lists needed in the $O(m)$-space adjacency list version of the sequential implementation in Section 3. Finally, the parallel version of this algorithm performs less work (by several polylogarithmic factors) than the exponential variates implementation.

As in the exponential variates algorithm of Section 6.2, we generate a permutation by treating each weighted edge as a collection of parallel unweighted edges. Rather than generating scores, we repeatedly simulate the uniform selection of a multigraph edge by choosing from the graph edges with probabilities proportional to the edge weights; the order of selection then determines the order of first appearance of multigraph edges.

Suppose we construct an array of m cumulative edge weights as we did in the sequential algorithm. We can use the procedure `Random-Select` to select one edge at random in $O(\log m)$ amortized time. Since it takes $O(m)$ time to recompute the cumulative distribution, it is undesirable to do so each time we wish to sample an edge. An alternative approach is to keep sampling from the original cumulative distribution and to ignore edges if we sample them more than once. Unfortunately, to make it likely that all edges have been sampled once, we may need a number of samples equal to the sum of the edge weights. For example, if one edge contains almost all the weight in the graph, we will continually select this edge. We solve this problem by combining the two approaches and recomputing the cumulative distribution only occasionally. For the time being, we shall assume that the total weight of edges in the graph is polynomial in n.

```
Procedure Iterated-Sampling($G, k$)

input A graph $G$

let $s = n^{1+\epsilon}$, for some constant $0 < \epsilon < 1$.

repeat
        Compute cumulative edge weights in $G$
        Let $M$ be a list of $s$ edge selections using Random-Select on the cumulative edge
            weights
        $G \leftarrow$ Compact($G, M, k$)
until $G$ has $k$ vertices
```

Figure 8: Iterated-Sampling Implementation

An implementation of the Contraction Algorithm called `Iterated-Sampling` is presented in Figure 8. Take ϵ to be any constant (say 1/2). We choose $s = n^{1+\epsilon}$ edges from the same cumulative distribution, contract all theses edges at once, recompute the cumulative distribution and repeat.

We now analyze the running time of `Iterated-Sampling`. We must be somewhat careful with this analysis because, as in `Random-Select`, we call `Iterated-Sampling` on very small problems that arise in the recursive computation of `Recursive-Contract`. Therefore, events that are "low probability" may actually happen with some frequency in the context of the original call to `Recursive-Contract`. We will therefore have to amortize these "low probability" events over the entire recursion, as we did for `Random-Select`. To analyze the running time of `Iterated-Sampling`, we use the following lemmas:

Lemma 7.1 *The worst case running time of* `Iterated-Sampling` *is* $O(n^3)$.

Proof: Each iteration requires $O(m + s \log n) = O(n^2)$ time. The first edge chosen in each iteration will identify a pair of vertices to be contracted; thus the number of iterations is at most n. □

Lemma 7.2 *Call an iteration of* `Iterated-Sampling` successful *if it finishes contracting the graph or if it reduces the total weight in the graph by a factor of* $2n/s = O(n^{-\epsilon})$ *for the next iteration. Then the probability that an iteration is not successful is* $e^{-\Omega(n)}$.

Proof: We assume that the weight reduction condition does not hold and show that the iteration must then be likely to satisfy the other success condition. Consider contracting the edges as they are chosen. At any time, call an edge *good* if its endpoints have not yet been merged by contractions Since `Iterated-Sampling` is not aware of the contractions, it may choose non-good edges. The total weight of edges in the next iteration is simply the total weight of good edges at the end of this iteration. Suppose that at the start of the iteration the total (all good) weight is W. By assumption, at the end the total good weight exceeds $2nW/s$. Since the total good weight can only decrease as contractions occur, we know that the total good weight at any time during this iteration exceeds $2nW/s$.

It follows that each time an edge is selected, the probability that it will be a good edge exceeds $2n/s$. Given that we perform s selections, the expected number of good selections exceeds $2n$. Then by the Chernoff bound [Che52, Mul94], the probability that fewer than n good edges are selected is exponentially small in n.

The number of contractions performed in an iteration is simply the number of good edges selected. Thus, by performing more than n good selections, the iteration will necessarily finish contracting the graph. □

Corollary 7.3 *On an n vertex graph, the number of* `Iterated-Sampling` *iterations before completion is at most t with probability $1 - e^{-\Omega(nt)}$.*

Proof: Recall our assumption that $W = n^{O(1)}$. Thus, in the language of Lemma 7.2, after a constant number of successful iterations `Iterated-Sampling` will terminate. Thus the only way for it to take more than t iterations is for there to be roughly t failures, each with probability $e^{-\Omega(n)}$ according to Lemma 7.2. □

Corollary 7.4 *On an n vertex graph, the running time of* `Iterated-Sampling` *is $O(t(m + n^{1+\epsilon}))$ with probability $1 - e^{-\Omega(nt)}$.*

Note that we set $s = n^{1+\epsilon}$ to make the analysis easiest for our purposes. A more natural setting is $s = m/\log m$ since this balances the time spent sampling and the time spent recomputing cumulative edge weights. Setting $s = m/\log m$ yields the same time bounds, but the analysis is more complicated.

7.1 A Strongly Polynomial Version

We now show how to let `Iterated-Sampling` remove the assumption that W is polynomial in n, while maintaining the same running times. The obstacle we must overcome is that the analysis of the number of iterations of `Iterated-Sampling` deals with the time to reduce W to zero. If W is arbitrarily large, this can take an arbitrarily large number of iterations.

To solve the problem, we use a very rough approximation to the minimum cut to ensure that Corollary 7.3 applies even when the edge weights are large. Let w be the largest edge weight such that the set of edges of weight greater than or equal to w connects all of G. This is just the minimum weight of an edge in a maximum spanning tree of G, and can thus be identified in $O(m \log n)$ time using any standard minimum spanning tree algorithm [CLR90]. Even better, it can be identified in $O(m)$ time by the `Compact` subroutine if we use the inverses of the actual edge weights as edge scores to determine the order of edge contraction. It follows that any cut of the graph must cut an edge of weight at least w, so the minimum cut has weight at least w. It also follows from the definition of w that there is a cut that does not cut any edge of weight exceeding w. This means that the graph has a cut of weight at most mw and hence the minimum cut has weight at most $mw \leq n^2 w$. This guarantees that no edge of weight exceeding $n^2 w$ can possibly be in the minimum cut. We can therefore contract all such edges, without eliminating any minimum cut in the graph. Afterwards the total weight of edges in the graph is at most $n^4 w$. Since we merge some edges, we may create new edges of weight exceeding $n^2 w$; these could be contracted as well but it is easier to leave them.

Consider running `Iterated-Sampling` on this reduced graph. Lemma 7.2 holds unchanged. Since the total weight is no longer polynomial, Corollary 7.3 no longer holds as a bound on the time to reduce the graph graph weight to 0. However, it does hold as bounds on the number of iterations needed to reduce the total remaining weight by a factor of n^4, so that it is less than w. Since the minimum cut exceeds w, the compacted graph at this point can have no cuts, since any such cut would involve only uncontracted edges and would thus have weight less than w. In other words, the graph edges that have been sampled up to this point must suffice to contract the graph to a single vertex. This proves that Corollaries 7.3 and 7.4 also hold in the case of arbitrary weights.

7.2 Sparse Graphs

Using the new, $O(m + n^{1+\epsilon})$-time algorithm allows us to speed up `Recursive Contract` on graphs with excluded minors. A minor of graph G is a graph that can be derived from G by deleting edges and vertices and contracting edges. A minor-excluded graph is one that does not contain some particular graph as a minor. Mader ([Mad68], see also [Bol86]) proved that in any minor-excluded graph, all r-vertex minors have $O(r)$ edges (thanks to Uri Zwick for pointing this out). Planar graphs fall into the class just discussed, as they exclude K_5.

Assume that we have a minor-excluded graph. Then we can be sure that at all times during the execution of the Recursive Contraction Algorithm the contracted graphs of r vertices will never have more than $O(r)$ edges. We use the $O(m + n^{1+\epsilon})$-time bound of Corollary 7.4 to get an improved running time for `Recursive-Contract`.

Theorem 7.5 *Let G be a minor-excluded graph. Then with high probability the Recursive Contraction algorithm finds all minimum cuts of G in $O(n^2 \log^2 n)$ time.*

Proof: We need to bound the time spent in all calls to `Iterated-Sampling` over all the various calls made to `Contract` in the computation tree of `Recursive-Contract`. An expected time analysis is quite easy. By Corollary 7.4, the expected time of `Iterated-Sampling` on a problem with m edges is $O(m + n^{1+\delta})$. By the assumption about graph minors, this means that the expected running time of `Contract` on an r-vertex subproblem will be $O(r)$. This gives us an improved recurrence for the expected running time:

$$T(n) = 2(n + T(n/\sqrt{2})).$$

This recurrence solve to $T(n) = O(n^2)$.

To improve the analysis to a high probability result, we consider two cases. At depths less than $\log n$ in the computation tree, where the smallest graph has at least $\sqrt{n}$ vertices, Corollary 7.4 says that the expected time bound for `Iterated-Sampling` is in fact a high probability time bound, so the recurrence holds with high probability at each node high in the computation tree. Below depth $\log n$, some of the problems are extremely small. However, Corollary 7.3 proves that each such problem has a running time that is geometrically distributed around its expectation. Since there are so many problems (more than n), the Chernoff bound can be applied to prove that the amortized time per problem is proportional to its expected value with high probability, much as was done to amortize the time for `Random-Select`. Thus at lower depths the recurrence holds in an amortized sense with high probability. $\square$

Observe that regardless of the structure of the graph minors, any attempt to reduce the running time below n^2 is frustrated by the need to generate n^2 computation leaves in order to ensure a high probability of finding the minimum cut.

7.3 Parallel Implementation

The iterated sampling procedure as also easy to parallelize. To perform one iteration of `Iterated-Sampling` in parallel, we use $m/\log n + n^{1+\epsilon}$ processors to first construct the cumulative edge weights and then perform $n^{1+\epsilon}$ random selections. We call the selection by processor 1 the "first" selection, that by processor 2 the "second" selection, imposing a selection order even though all the selections take place simultaneously. We use these selections in the parallel implementation of the procedure `Compact`. Corollary 7.4 proves that until the problem sizes in the `Recursive-Contract` computation tree are smaller than

$\Omega(\log n)$, each application of `Iterated-Sampling` runs in $O(\log^2 n)$ time with high probability. At levels below $\log n$, we can use the worst case time bound for `Iterated-Sampling` to show that the running time remains polylogarithmic.

8 Approximately Minimum Cuts

The Contraction Algorithm can also be used to find cuts that are not minimum but are relatively small. The problem of finding all nearly minimum cuts has been shown to have important ramifications in the study of network reliability, since such enumerations allow one to drastically improve estimates of the reliability of a network. This was shown in [RC87], where an $O(n^{k+2}m^k)$ bound was given for the number of cuts of value $c+k$ in a graph with minimum cut c, and an algorithm with running time $O(n^{k+2}m^k)$ was given for finding them. Karger [Kar95] uses small-cut enumeration is the basis of a fully polynomial time approximation scheme for the all-terminal reliability problem. We begin with some cut-counting results from [Kar94a].

Definition 8.1 *An* α-minimal cut *is a cut of value within a multiplicative factor of* α *of the minimum.*

Theorem 8.2 *For k a half-integer, the probability that a particular k-minimal cut survives contraction to $2k$ vertices is* $\Omega(\binom{n}{2k}^{-1})$.

Proof: We consider the unweighted case; the extension to the weighted case goes as before. The goal is to again apply Lemma 2.1. Let k be a half-integer, and c the minimum cut, and consider some cut of weight at most kc. Suppose we run the Contraction Algorithm. If with r vertices remaining we choose a random edge, then since the number of edges is at least $cr/2$, we take an edge from a cut of weight kc with probability at most $2k/r$. If we do this until $r = 2k$, then the probability that the cut survives is

$$(1-\frac{2k}{n})(1-\frac{2k}{(n-1)})\cdots(1-\frac{2k}{(2k+1)}) \quad = \quad \binom{n}{2k}^{-1}$$

□

Remark: As with the minimum cut theorem, a ring on n vertices shows this theorem is tight. □

We can use this theorem to find k-minimal cuts. Since we stop before the number of vertices reaches 2, we still have to finish selecting a cut. Do so by randomly partitioning the remaining vertices into two groups. Since there are less than 2^{2k} partitions, it follows that the probability of a particular cut being chosen is at least $2^{-2k}\binom{n}{2k}^{-1}$.

Corollary 8.3 *For arbitrary real values of k, the probability that a particular k-minimal cut is found by the Contraction Algorithm is* $\Omega((2n)^{-2k})$.

Proof: Let $r = \lceil 2k \rceil$. Suppose we contract the graph until there are only r vertices remaining, and then pick one of the 2^r cuts of the resulting graph uniformly at random. The probability that a particular k-minimal cut survives the contraction to r vertices is

$$\begin{aligned}(1-\frac{2k}{n})(1-\frac{2k}{(n-1)})\cdots(1-\frac{2k}{r+1}) &= \frac{\binom{n-2k}{n-r}}{\binom{n}{n-r}} \\ &= \frac{\binom{r}{2k}}{\binom{n}{2k}},\end{aligned}$$

where in the above equations we use *generalized binomial coefficients* for non-integral arguments (see Knuth [Knu73, Sections 1.2.5–6] for details). From [Knu73, Exercise 1.2.6.45], we know that $\binom{n}{2k} = \Theta(n^{2k})$ for fixed k. Since $\binom{r}{2k}$ is a constant independent of n, the overall probability is $\Theta(n^{-2k})$. Multiplying this by the probability that we pick the correct one of the $2^r \approx 2^{2k}$ remaining cuts yields the desired result. $\square$

It follows from the above proof that we can in fact find *all* approximately minimal cuts in polynomial time. The first step towards proving this is a corollary regarding the number of such cuts that can exist in a graph. This corollary has other important applications which are investigated in [Kar94c, KM96]. Further exploration of this theorem can be found in those papers.

Theorem 8.4 *In any graph, the number of α-minimal cuts is $O((2n)^{2\alpha})$.*

Proof: Since the above algorithm outputs only one cut, the survivals of the different cuts are disjoint events. Thus, the probability that one of the α-minimal cuts is output is the sum of the probabilities that each is output. This sum must be less than 1. By Corollary 8.3, every such cut has an $\Omega((2n)^{-2\alpha})$ probability of being produced by the Contraction Algorithm. The bound on the possible number of cuts follows. $\square$

A previous bound of $O(n^2)$ for the number of minimum cuts was proved by other means in [DKL76]. No previous bound on the number of cuts of by value was known. Karger [Kar96] has sine improved the bound to $O(\binom{n}{\lfloor 2\alpha \rfloor})$.

Our efficient implementation of the contraction algorithm can be applied to approximately minimum cuts:

Theorem 8.5 *All cuts with weight within a multiplicative factor α of the minimum cut can be found in $O(n^{2\alpha}\log^2 n)$ time.*

Proof: Change the reduction factor from $\sqrt{2}$ to $\sqrt[2\alpha]{2}$ in the Recursive Contraction Algorithm. Stop when the number of vertices remaining is $2\lceil\alpha\rceil$, and check all remaining cuts. The recurrence for success probability yields the same result, while the running time recurrence becomes

$$T(n) = n^2 + 2T(n/\sqrt[2\alpha]{2})$$

and solve to the claimed time bound. The probability that any one cut is missed can be made polynomially small, and thus, since there are only polynomially many approximately minimal cuts, we will find all of them with high probability. $\square$

Remark: The disappearance of an $O(\log n)$ factor that was present in the 2-way cut case was brought to our attention by Jan Hvid Sorensen. $\square$

Vazirani and Yannakakis [VY92] give algorithms for enumerating cuts by *rank*, finding the k^{th} smallest cut in $O(n^{3k})$ time, while we derive bounds based on the *value* of a cut relative to the others. They also give a bound of $O(n^{3k-1})$ on the number of cuts with the k^{th} smallest weight. Note that their bounds are incomparable with ours.

This theorem gives another way to make the Recursive Contraction Algorithm strongly polynomial. Essentially, we scale and round the edge weights in such a way that all edges become polynomial sized integers. At the same time, we arrange that no cut changes in value by more than a small amount; it follows that the minimum cut in the original graph must be a nearly minimum cut in the new graph. Thus an algorithm that finds all approximately minimum cuts will find the original minimum cut. We arrange that the relative change in any cut value is $1/n$, so that the running time is changed only by a constant factor. This method is necessary in the derandomization of [KM96].

9 Multiway Cuts

The Contraction Algorithm can also be used to find a *minimum weight r-way cut* that partitions the graph into r pieces rather than 2. As before, the key to the analysis is to apply Lemma 2.1 by bounding the probability p that a randomly selected graph edge is from a particular minimum r-cut. Throughout, to simplify our asymptotic notation, we assume r is as constant.

Lemma 9.1 *The number of edges in the minimum r-way cut of a graph with m edges and n vertices is at most*

$$[1-(1-\frac{r-1}{n})(1-\frac{r-1}{n-1})]m$$

Proof: We use the probabilistic method. Suppose we choose $r-1$ vertices uniformly at random, and consider the r-way cut defined by taking each of the chosen vertices alone as of the $r-1$ vertex sets of the cut and all the other vertices as the last set. An edge is in an r-way cut if its endpoints are in different partitions. The probability that a particular edge is in the cut is thus the probability that either of its endpoints is one of the $r-1$ single-vertex components of the cut, which is just

$$1-(1-\frac{r-1}{n})(1-\frac{r-1}{n-1}).$$

Let f be the number of edges cut by this random partition, and m the number of graph edges. The number of edges we expect to cut is m times the probability that any one edge is cut, *i.e.*

$$E[f]=[1-(1-\frac{r-1}{n})(1-\frac{r-1}{n-1})]m.$$

Since f can be no less than the value of the minimum r-way cut, $E[f]$ must also be no less than the minimum r-way cut. □

The quantity in brackets is thus an upper bound on the probability that a randomly selected edge is an r-way minimum cut edge.

Theorem 9.2 *Stopping the Contraction Algorithm when r vertices remain yields a particular minimum r-way cut with probability at least*

$$r\binom{n}{r-1}^{-1}\binom{n-1}{r-1}^{-1}=\Omega(n^{-2(r-1)}).$$

Proof: By the previous lemma, arguing as in Lemma 2.2, the probability that a particular minimum r-cut survives the reduction process until there are r vertices remaining is at least

$$\begin{aligned}&\prod_{u=r+1}^{n}(1-\frac{r-1}{u})(1-\frac{r-1}{u-1})\\ &=\prod_{u=r+1}^{n}(1-\frac{r-1}{u})\prod_{u=r+1}^{n}(1-\frac{r-1}{u-1})\\ &=r\binom{n}{r-1}^{-1}\binom{n-1}{r-1}^{-1}.\end{aligned}$$

□

Corollary 9.3 *The probability that a particular minimum r-way cut survives contraction to $k\geq r$ vertices is $\Omega((k/n)^{2(r-1)})$.*

Corollary 9.4 *There are $O(n^{2(r-1)})$ minimum multiway cuts in a graph.*

Proof: Use the same argument as for counting approximately minimum cuts. □

Theorem 9.5 *For any integral $r>2$, all minimum r-way cuts in a graph can be found with high probability in $O(n^{2(r-1)}\log^2 n)$ time, or in $\mathcal{RNC}$ using $n^{2(r-1)}$ processors.*

Proof: Apply the Recursive Contraction Algorithm, but contract at each level by a factor of $\sqrt[2(r-1)]{2}$ and stop when r vertices remain. The recurrence for the probability of success is unchanged. The running time recurrence becomes

$$T(n) = n^2 + 2T(n/2^{1/2(r-1)})$$

and solves to $T(n) = O(n^{2(r-1)})$. The fact that all cuts are found follows as in the approximately minimal cuts case. □

This is a significant improvement over the previously best known sequential time bound of $O(n^{r^2-r+11/2})$ reported in [GH88]. This also provides the first proof that the multiway cut problem is in $\mathcal{RNC}$ for constant r. The extension of these techniques to approximately minimum multiway cuts is an easy exercise that we omit due to rather complicated notation needed.

10 Cut Data Structures

Researchers have investigated several representations of the minimum cuts of a graph. Desirable properties of such representations include small space requirements and, perhaps more importantly, the ability to quickly answer queries about the minimum cuts in the graph. Several representations are known [DKL76, Gab91]. We concentrate on the *cactus representation* [DKL76]. This data structure represents all $\binom{n}{2}$ minimum cuts via an n-node, $O(n)$-edge graph. It can be used to quickly identify, for example, all minimum cuts separating a particular pair of vertices. Karzanov and Timofeev [KT86] give an algorithm for constructing the cactus sequentially; their algorithm is parallelized by Naor and Vazirani [NV91]. We describe the general framework of both algorithms below. The reader is referred to [NV91] for a much more detailed description.

1. Number the vertices so that for each vertex (except vertex 1) is connected to at least one lower numbered vertex.
2. For each $i \geq 2$, compute the set S_i of minimum cuts that separate vertices $\{1, \ldots, i-1\}$ from vertex i.
3. Form a cactus out of $\cup_i S_i$.

Step 2 turns out to be the crux of the algorithm. The sets S_i form what we call the *chain representation* of minimum cuts, for reasons we now explain. For our explanation, it is convenient to slightly change our definition of cuts. Given a cut (A, B), we can identify the cut with either set A or set B since one is a complement of the other. To make the identification unique we take the set containing vertex 1. Thus a cut is simply a set A of vertices containing vertex 1, and its value is weight of edges with exactly one endpoint in A. We will say that the vertices in A are *inside* the cut, and those in $\overline{A}$ are *outside* the cut. We let the *size* of a cut be the number of vertices in its representative set.

Given the numbering of Step 1 and our redefinition of cuts, each S_i has a particularly nice structure. Namely, given any two cuts A and A' in S_i, either $A \subset A'$ or $A' \subset A$. This property is typically referred to the *non-crossing cut property*. It follows that the cuts in S_i form a *chain*, *i.e.* the cuts can be numbered as A_i such that $A_1 \subset A_2 \subset \cdots \subset A_k$. Therefore, it is easy to represent each set S_i in $O(n)$ space, meaning that the S_i form an $O(n^2)$-size *chain representation* of the minimum cuts of G.

We now consider the implementation of the cactus construction. Step 1 of the algorithm can be implemented easily: find a spanning tree of G and then number the vertices according to a preorder traversal. This can be done in $O(m)$ time sequentially and also in $O(\log n)$

time using $m/\log n$ processors in parallel [KR90]. Step 3 can also be implemented relatively efficiently. Karzanov and Timofeev [KT86] describe a sequential implementation that, given the set of chains for each S_i, takes $O(n^2)$ time. Naor and Vazirani [NV91] do not explicitly bound their implementation of Step 3, but it can be shown to run in $O(\log^2 n)$ time using n^4 processors. For both the sequential and parallel algorithms, the bottleneck in performance turned out to be Step 2, constructing the chain representation.

10.1 Constructing the Chain Representation

In Step 2, each S_i can be found via a maximum flow computation and a strongly connected components computation and thus Step 2 can be done by n such computations. This led to a sequential algorithm that took $\tilde{O}(n^2m)$ time [KT86] and an $O(\log^2 n)$ time randomized algorithm that used $n^{4.5}m$ processors on unweighted graphs [NV91]. We will explain how to implement Step 2 to run using the same amount of resources as the Recursive Contraction Algorithm (up to constant factors), thus leading to improved sequential time and parallel processor bounds.

Suppose that for each vertex number j, we know the size of the smallest cut in S_i containing j (that is, with j on the same side as vertex 1). Then it is straightforward to construct S_i in $O(n)$ time. Bucket-sort the vertices according to the smallest S_i-cut containing them. Those inside the smallest cut form A_1; those inside the next smallest form $A_2 - A_1$, and so on. Therefore, we have reduced the problem of constructing the S_i to the following: for each i and j, identify the smallest S_i-cut containing j. We now show how to modify the Recursive Contraction Algorithm to recursively compute this information. For simplicity, assume that we have already run the Recursive Contraction Algorithm once so that the value of the minimum cut is known.

We begin by adding two information fields to each metavertex v that arises during the Recursive Contraction Algorithm's execution. Let $size(v)$ be the number of vertices contained in v, and let $min(v)$ be the smallest label of a vertex in v. Note that these two quantities are easy to update as the algorithm executes; when we merge two metavertices, the updated values are determined in constant time by a sum and a minimum operation. Now consider a leaf in the computation tree of the Recursive Contraction Algorithm. One metavertex v in this leaf will have $min(v) = 1$ while the other metavertex w will have $min(w) = i$ for some i. If this leaf corresponds to a minimum cut of G, then we call it an *i-leaf.* Each i-leaf must correspond to a cut in S_i, since by the labeling, vertices $1, \ldots, i-1$ must be in v while vertex i must be in w. Furthermore, $size(v)$, which we also call the size of the i-leaf, is just the number of vertices inside the corresponding minimum cut. We have therefore reduced our chain construction problem to the following: for each pair of labels i and j, find the minimum size i-leaf containing j (where we identify an i-leaf with the cut (set of vertices) it represents).

We solve this problem by generalizing it while running the Recursive Contraction Algorithm. Consider some graph G which arises at some point in the computation tree. We solve the following problem: for each pair of labels i and j of vertices in G, consider all i-leaves that are descendants of G and find $\mu^i_G(j)$, the smallest i-leaf descendant of G containing j. Recalling that in the computation tree G has two contracted graphs G' and G'' as children, we show that it is easy to compute μ^i_G from $\mu^i_{G'}$ and $\mu^i_{G''}$. Note that each i-leaf descended from G is descended from either G' or G''. Consider graph G'. The metavertices with labels i and j in G are merged into metavertices with labels i' and j' in G'. Suppose $i \neq i'$. Then there is no vertex labeled i in G', and it follows by induction that there is no i-leaf descended from G'. If $i = i'$, then the smallest i-leaf descendent of G' containing j is just the smallest i'-leaf descendant of G' containing j', namely $\mu^{i'}_{G'}(j')$. Applying the same argument to G'', it follows that

$$\mu^i_G(j) = \min(\mu^i_{G'}(j'), \mu^i_{G''}(j')),$$

where $\mu^i_G()$ is defined to be infinite if there is no vertex labeled i in G.

We have therefore shown that, after the recursive calls to G' and G'' which return $\mu_{G'}$ and $\mu_{G''}$, the new $\mu^i_G(j)$ can be computed in constant time for each pair of labels i and j in G. Therefore, if G has n vertices and thus n labels, the time to compute all $\mu^i_G(j)$ is $O(n^2)$. Since the original contraction algorithm already performs $O(n^2)$ work at each size n graph in the computation, the additional $O(n^2)$ work does not affect the running time bound. This procedure is easy to parallelize, as computing $\mu^i_G(j)$ for all pairs i and j can be done simultaneously, and the sorting can also be done efficiently in $\mathcal{NC}$.

Finally, recall that we run the Recursive Contraction Algorithm $\Theta(\log^2 n)$ times in order to get a high probability of finding every minimum cut. It is trivial to combine the resulting μ values from these $\Theta(\log^2 n)$ computations in $O(n^2 \log^2 n)$ time or with the same number of processors in $O(\log n)$ time. We have therefore shown:

Theorem 10.1 *The chain representation of minimum cuts in a weighted labeled graph can be computed with high probability in $O(n^2 \log^3 n)$ time, or in $\mathcal{RNC}$ using n^2 processors.*

Corollary 10.2 *The cactus representation of minimum cuts in a graph can be computed in $O(n^2 \log^3 n)$ time or in $\mathcal{RNC}$ using n^4 processors.*

11 Optimizing Space

In this section, we show how the Contraction Algorithm can be implemented to run in $O(n)$ space, though with an increase in running time. The Union-Find data structure of [Tar83, page 23] provides for an implementation of the Contraction Algorithm. We use the Union-Find data structure to identify sets of vertices that have been merged by the contractions. Initially, each vertex is in its own set. We repeatedly choose an edge at random, and apply a union operation to its endpoints' sets if they do not already belong to the same set. We continue until only two sets remain. Each choice of an edge requires one find operation, and we will also perform a total of $n-2$ union operations. Furthermore, after $O(m \log m)$ random selections, the probability is high that we will have selected each edge at least once. Thus, if the graph is connected, we will have contracted to two vertices by this time. Therefore the total running time of the Contraction Algorithm will be $O(m \log m)$ with high probability. The use of path compression in the union-find data structure provides no improvement in this running time, which is dominated by the requirement that every edge be sampled at least once.

The results of this section can be summarized as follows:

Theorem 11.1 *On unweighted graphs, the Contraction Algorithm can be implemented to run in $O(m \log m)$ time and $O(n)$ space with high probability.*

We can find a minimum cut by running this algorithm $O(n^2 \log n)$ times and taking the best result. An improved approach is the following.

Corollary 11.2 *Using $s \geq n$ space, it is possible to find the minimum cut in an unweighted graph in $\tilde{O}(n^2 + mn^2/s)$ time with high probability.*

Proof: Use the modified contraction algorithm above to contract the graph to $\sqrt{s}$ vertices in $\tilde{O}(m)$ time. At this point, the entire contracted graph has $O(s)$ edges and can therefore be represented in s space. Therefore, we can run the Recursive Contraction Algorithm in time $\tilde{O}(s)$ to find the minimum cut. The probability that the minimum cut survives the contraction to $\sqrt{s}$ vertices is $\Omega(s/n^2)$, so we need to repeat this whole procedure $\tilde{O}(n^2/s)$ times. This gives an overall running time of $\tilde{O}((m+s)n^2/s) = \tilde{O}(n^2 + mn^2/s)$. □

We can extend this unweighted-graph approach to weighted graphs, although the time bound becomes worse. As before, we use the union-find data structure of [Tar83] to contract edges as we select them. Instead of maintaining a list of all unsampled edges, we maintain a threshold $X(t)$ such that any edge of weight exceeding $X(t)$ has a high probability of being sampled within t trials. After time t we sample only from among those edges that have weight less than this threshold. This gives a running time of $O(m \log W)$.

12 Conclusions

We have given efficient and simple algorithms for the minimum cut problem, yet several interesting open questions remain. Karger [Kar96] has given faster minimum cut algorithms: one with running time $O(m \log^3 n)$ and a simpler one with running time $O(n^2 \log n)$. An obvious open question is therefore: how close to linear-time can we get in solving the minimum cut problem in theory and in practice?

Another question is the extent to which randomization is needed. Karger and Motwani [KM96] have used the Contraction Algorithm to prove that the minimum cut can be found in $\mathcal{NC}$; however, the resulting processor bounds are prohibitively large for practical purposes.

An important first step towards derandomization would be a so-called *Las Vegas* algorithm for minimum cuts. The Recursive Contraction Algorithm has a very high probability of finding a minimum cut, but there is no fast way to prove that it has done so, as all known certificates for a minimum cut, such as a maximum flow, or Gabow's the complete intersections [Gab95], take too long to compute. The Contraction Algorithm is thus *Monte Carlo.* The same applies to the faster algorithms of [Kar96]. A fast Las Vegas Algorithm for unweighted graphs is given in [Kar94a], but the running time does not match the Monte Carlo algorithms'.

Since we are now able to find a minimum cut faster than a maximum flow, it is natural to ask whether it is any easier to compute a maximum flow given a minimum cut. Ramachandran [Ram87] has shown that knowing an s-t minimum cut is not helpful in finding an s-t maximum flow. However, the question of whether knowing any or all minimum cuts may help to find an s-t maximum flow remains open.

Another obvious question is whether any of these results can be extended to directed graphs. It seems unlikely that the Contraction Algorithm, with its inherent parallelism, could be applied to the $\mathcal{P}$-complete directed minimum cut problem. However, the question of whether it is easier to find a minimum cut than a maximum flow in directed graphs remains open.

The minimum cut algorithm of Gomory and Hu [GH61] not only found the minimum cut, but found a *flow equivalent tree* that succinctly represented the values of the $\binom{n}{2}$ minimum cuts. No algorithm is known that computes a flow equivalent tree or the slightly stronger Gomory-Hu tree in time that is less than the time for n maximum flows. An intriguing open question is whether the methods in this paper can be extended to produce a Gomory-Hu tree.

Acknowledgments

We thank David Applegate for providing us with timely information about cutting plane based algorithms for the traveling salesman problems. We also thank Tom Cormen and Philip Klein for helpful conversations, and Daphne Koller for her comments.

References

[ABCC95] David Appelgate, Robert Bixby, Vašek Chváatal, and William Cook. Finding cuts in the tsp. Technical Report 95-05, DIMACS, Rutgers University, New Brunswick, NJ, 1995.

[ACM92] ACM. *Proceedings of the* 24^{th} *ACM Symposium on Theory of Computing.* ACM Press, May 1992.

[ACM93] ACM-SIAM. *Proceedings of the* 4^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1993.

[ACM94] ACM. *Proceedings of the* 26^{th} *ACM Symposium on Theory of Computing.* ACM Press, May 1994.

[ACM96] ACM. *Proceedings of the* 28^{th} *ACM Symposium on Theory of Computing.* ACM Press, May 1996.

[App92] David Applegate. AT&T Bell Labs, 1992. Personal Communication.

[Ben94] András A. Benczúr. Augmenting undirected connectivity in $\mathcal{RNC}$ and in randomized $\tilde{O}(n^3)$ time. In *Proceedings of the* 26^{th} *ACM Symposium on Theory of Computing* [ACM94], pages 658–667.

[BK96] András A. Benczúr and David R. Karger. Approximate s–t min-cuts in $\tilde{O}(n^2)$ time. In *Proceedings of the* 28^{th} *ACM Symposium on Theory of Computing* [ACM96], pages 47–55.

[Bol86] Béla Bollobás. *Extremal Graph Theory with Emphasis on Probabilistic Methods.* Number 62 in Regional Conference Series in Mathematics. American Mathematical Society, Providence, RI, 1986.

[Bot93] Rodrigo A. Botafogo. Cluster analysis for hypertext systems. In *Proceedings of the* 16^{th} *Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 116–125, June 1993.

[CDR86] Stephen Cook, Cynthia Dwork, and Rudiger Reischuk. Upper and lower bounds for parallel random access machines without simultaneous writes. *SIAM Journal on Computing*, 15(1):87–97, February 1986.

[CH95] Joseph Cheriyan and Torben Hagerup. A randomized maximum-flow algorithm. *SIAM Journal on Computing*, 24(2):203–226, April 1995. A preliminary version appeared in FOCS 1989.

[Cha94] S. Chaterjee, April 1994. Personal communication.

[Che52] H. Chernoff. A measure of the asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–509, 1952.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[Col87] Charles J. Colbourn. *The Combinatorics of Network Reliability*, volume 4 of *The International Series of Monographs on Computer Science.* Oxford University Press, 1987.

[DFJ54] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a large-scale traveling salesman problem. *Operations Research*, 2:393–410, 1954.

[DJP+94] E. Dahlhaus, David S. Johnson, Christos H. Papadimitriou, P. D. Seymour, and Mihalis Yannakakis. The complexity of multiway cuts. *SIAM Journal on Computing*, 23(4):864–894, 1994. A preliminary version appeared in STOC 1992.

[DKL76] Efim A. Dinitz, A. V. Karzanov, and Micael V. Lomonosov. On the structure of a family of minimum weighted cuts in a graph. In A. A. Fridman, editor, *Studies in Discrete Optimization*, pages 290–306. Nauka Publishers, 1976.

[EFS56] P. Elias, A. Feinstein, and C. E. Shannon. Note on maximum flow through a network. *IRE Transactions on Information Theory IT-2*, pages 117–199, 1956.

[EK72] Jack Edmonds and Richard M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19:248–264, 1972.

[FF56] Lester R. Ford, Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[FF62] Lester R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks.* Princeton University Press, Princeton, New Jersey, 1962.

[Fra94] Andras Frank. On the edge-connectivity algorithm of Nagamochi and Ibaraki. Labarotoire Artemis, IMAG, Université J. Fourier, Grenoble, March 1994.

[Gab91] Harold N. Gabow. Applications of a poset representation to edge connectivity and graph rigidity. In *Proceedings of the* 32^{nd} *Annual Symposium on the Foundations of Computer Science*, pages 812–821. IEEE, IEEE Computer Society Press, October 1991.

[Gab95] Harold N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *Journal of Computer and System Sciences*, 50(2):259–273, April 1995. A preliminary version appeared in STOC 1991.

[GH61] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society of Industrial and Applied Mathematics*, 9(4):551–570, December 1961.

[GH88] Oliver Goldschmidt and Dorit Hochbaum. Polynomial algorithm for the k-cut problem. In *Proceedings of the* 29^{th} *Annual Symposium on the Foundations of Computer Science*, pages 444–451. IEEE Computer Society Press, 1988.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, San Francisco, 1979.

[GP88] Zvi Galil and Victor Pan. Improved processor bounds for combinatorial problems in $\mathcal{RNC}$. *Combinatorica*, 8:189–200, 1988.

[GSS82] L. M. Goldschlager, R. A. Shaw, and J. Staples. The maximum flow problem is logspace complete for P. *Theoretical Computer Science*, 21:105–111, 1982.

[GT88] Andrew V. Goldberg and Robert E. Tarjan. A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988.

[Has89] Hastad. Almost optimal lower bounds for small depth circuits. *Advances in Computing Research*, 5, 1989. A preliminary version appeared in STOC 1986.

[HO94] Hao and Orlin. A faster algorithm for finding the minimum cut in a directed graph. *Journal of Algorithms*, 17(3):424–446, 1994. A preliminary version appeared in SODA 1992.

[HZ94] Shay Halperin and Uri Zwick. An optimal randomized logarithmic time connectivity algorithm for the EREW PRAM. In *Proceedings of the* 6^{th} *Annual ACM-SIAM Symposium on Parallel Algorithms and Architectures*, pages 1–10. ACM, 1994.

[Kar93] David R. Karger. Global min-cuts in $\mathcal{RNC}$ and other ramifications of a simple mincut algorithm. In *Proceedings of the* 4^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms* [ACM93], pages 21–30.

[Kar94a] David R. Karger. Random sampling in cut, flow, and network design problems. In *Proceedings of the* 26^{th} *ACM Symposium on Theory of Computing* [ACM94], pages 648–657. Submitted for publication..

[Kar94b] David R. Karger. *Random Sampling in Graph Optimization Problems.* PhD thesis, Stanford University, Stanford, CA 94305, 1994. Contact at `karger@lcs.mit.edu`. Available by ftp from `theory.lcs.mit.edu`, directory `pub/karger`.

[Kar94c] David R. Karger. Using randomized sparsification to approximate minimum cuts. In *Proceedings of the* 5^{th} *Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 424–432. ACM-SIAM, January 1994.

[Kar95] David R. Karger. A randomized fully polynomial approximation scheme for the all terminal network reliability problem. In *Proceedings of the* 27^{th} *ACM Symposium on Theory of Computing*, pages 11–17. ACM, ACM Press, May 1995.

[Kar96] David R. Karger. Minimum cuts in near-linear time. In *Proceedings of the* 28^{th} *ACM Symposium on Theory of Computing* [ACM96], pages 56–63.

[KM96] David R. Karger and Rajeev Motwani. Derandomization through approximation: An $\mathcal{NC}$ algorithm for minimum cuts. *SIAM Journal on Computing*, 1996. To appear.A preliminary version appeared in STOC 1993, p. 497.

[Knu73] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley Publishing Company, 2nd edition, 1973.

[KPST94] Philip Klein, Serge A. Plotkin, Clifford Stein, and Éva Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. *SIAM Journal on Computing*, 23(3):466–487, 1994. A preliminary version appeared in STOC 90.

[KR90] Richard M. Karp and Vijaya Ramachandran. Parallel algorithms for shared memory machines. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume A, pages 869–932. MIT Press, Cambridge, MA, 1990.

[KRT94] Valerie King, Satish Rao, and Robert E. Tarjan. A faster deterministic maximum flow algorithm. *Journal of Algorithms*, 17(3):447–474, November 1994. A preliminary version appeared in SODA 1992.

[Kru56] J. B. Kruskal, Jr. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7(1):48–50, 1956.

[KS91] Samir Khuller and Baruch Schieber. Efficient parallel algorithms for testing connectivity and finding disjoint s-t paths in graphs. *SIAM Journal on Computing*, 20(2):352–375, April 1991.

[KS93] David R. Karger and Clifford Stein. An $\tilde{O}(n^2)$ algorithm for minimum cuts. In *Proceedings of the* 25th *ACM Symposium on Theory of Computing*, pages 757–765. ACM, ACM Press, May 1993.

[KT86] A. V. Karzanov and E. A. Timofeev. Efficient algorithm for finding all minimal edge cuts of a non-oriented graph. *Cybernetics*, 22:156–162, 1986.

[KUW86] Richard M. Karp, Eli Upfal, and Avi Wigderson. Constructing a perfect matching is in random $\mathcal{NC}$. *Combinatorica*, 6(1):35–48, 1986.

[KY76] Donald E. Knuth and Andrew C. Yao. The complexity of nonuniform random number generation. In Joseph F. Traub, editor, *Algorithms and Complexity: New Directions and Recent Results*, pages 357–428. Academic Press, 1976.

[LLKS85] Eugene L. Lawler, J. K. Lenstra, A. H. G. Rinooy Kan, and David B. Shmoys, editors. *The Traveling Salesman Problem.* John Wiley & Sons, 1985.

[Lom94] Micael V. Lomonosov. On Monte Carlo estimates in network reliability. *Probability in the Engineering and Informational Sciences*, 8:245–264, 1994.

[Mad68] W. Mader. Homomorphiesätze für graphen. *Math. Ann.*, 178:154–168, 1968.

[Mat87] D. W. Matula. Determining edge connectivity in $O(nm)$. In *Proceedings of the* 28th *Annual Symposium on the Foundations of Computer Science*, pages 249–251. IEEE, IEEE Computer Society Press, 1987.

[Mat93] D. W. Matula. A linear time $2+\epsilon$ approximation algorithm for edge connectivity. In *Proceedings of the* 4th *Annual ACM-SIAM Symposium on Discrete Algorithms* [ACM93], pages 500–504.

[Mul94] Ketan Mulmuley. *Computational Geometry.* Prentice Hall, 1994.

[MVV87] Ketan Mulmuley, Umesh V. Vazirani, and Vijay V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

[Neu51] J. Von Neumann. Various techniques used in connection with random digits. *National Bureau of Standards, Applied Math Series*, 12:36–38, 1951.

[NI92] Hiroshi Nagamochi and Toshihide Ibaraki. Computing edge connectivity in multigraphs and capacitated graphs. *SIAM Journal of Discrete Mathematics*, 5(1):54–66, February 1992.

[NV91] Dalit Naor and Vijay V. Vazirani. Representing and enumerating edge connectivity cuts in $\mathcal{RNC}$. In F. Dehne, J. R. Sack, and N. Santoro, editors, *Proceedings of the* 2nd *Workshop on Algorithms and Data Structures*, volume 519 of *Lecture Notes in Computer Science*, pages 273–285. Springer-Verlag, August 1991.

[Pod73] V. D. Podderyugin. An algorithm for finding the edge connectivity of graphs. *Vopr. Kibern.*, 2:136, 1973.

[PQ82] J. C. Picard and M. Queyranne. Selected applications of minimum cuts in networks. *I.N.F.O.R: Canadian Journal of Operations Research and Information Processing*, 20:394–422, November 1982.

[PR75] J.C. Picard and H.D. Ratliff. Minimum cuts and related problems. *Networks*, 5:357–370, 1975.

[PR90] M. Padberg and G. Rinaldi. An efficient algorithm for the minimum capacity cut problem. *Mathematical Programming*, 47:19–39, 1990.

[PW92] Steven Phillips and Jeffrey Westbrook. Online load balancing and network flow. In *Proceedings of the* 24^{th} *ACM Symposium on Theory of Computing* [ACM92], pages 402–411.

[Ram87] Vijaya Ramachandran. Flow value, minimum cuts and maximum flows. Manuscript., 1987.

[RC87] A. Ramanathan and Charles Colbourn. Counting almost minimum cutsets with reliability applications. *Mathematical Programming*, 39(3):253–61, December 1987.

[ST83] Daniel D. Sleator and Robert E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, June 1983.

[SV82] Y. Shiloach and Uzi Vishkin. An $O(\log n)$ parallel connectivity algorithm. *Journal of Algorithms*, 3:57–67, 1982.

[SW94] Methchild Stoer and F. Wagner. A simple min cut algorithm. In Jan van Leeuwen, editor, *Proceedings of the 1994 European Symposium on Algorithms*, pages 141–147. Springer-Verlag, September 1994.

[Tar83] Robert E. Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NSF Regional Conference Series in Applied Mathematics*. SIAM, 1983.

[VY92] Vijay V. Vazirani and Mihalis Yannakakis. Suboptimal cuts: Their enumeration, weight, and number. In *Automata, Languages and Programming.* 19^{th} *International Colloquium Proceedings*, volume 623 of *Lecture Notes in Computer Science*, pages 366–377. Springer-Verlag, July 1992.