An $\tilde{O}(n^2)$ Algorithm for Minimum Cuts

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

Clifford Stein†
Department of Math and CS
Dartmouth College
cliff@cs.dartmouth.edu

Abstract

A *minimum cut* is a set of edges of minimum weight whose removal disconnects a given graph. Minimum cut algorithms historically applied duality with maximum flows and thus had the same $\Omega(mn)$ running time as maximum flow algorithms. More recent algorithms which are not based on maximum flows also require $\Omega(mn)$ time.

In this paper, we present the first algorithm that breaks the $\Omega(mn)$ "max-flow barrier" for finding minimum cuts in weighted undirected graphs. We give a strongly polynomial randomized algorithm which finds a minimum cut with high probability in $O(n^2 \log^3 n)$ time. This suggests that the min-cut problem might be fundamentally easier to solve than the maximum flow problem. Our algorithm can be implemented in $\mathcal{RNC}$ using only n^2 processors—this is the first efficient $\mathcal{RNC}$ algorithm for the min-cut problem. Our algorithm is simple and uses no complicated data structures.

1 Introduction

1.1 The Problem

This paper studies the min-cut problem. Given a 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. 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, since otherwise the problem is $\mathcal{NP}$-complete by a trivial transformation from the maximum-cut problem. The problem actually has two variants: in the *s-t min-cut* problem, we require that the two specified vertices s and t be on opposite sides of the cut; in what we call the *min-cut* problem there is no such restriction. Observe that the value of the minimum cut in a graph is equal to the minimum, taken over all pairs of vertices s and t, of the s-t minimum cut.

Particularly on unweighted graphs, the min-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 min-cut problem has applications in many fields. The problem of determining the connectivity of a network arises frequently in issues of network design and network reliability (Karger [Kar93] demonstrates an extremely tight connection between the minimum cut and network reliability). For weighted graphs, Picard and Querayne [PQ82] survey numerous applications of minimum cuts, including graph partitioning problems, the study of project networks, and partitioning items in a database.

Minimum cuts also arise in other combinatorial optimization problems—for example in the traveling salesman problem. The currently best methods for finding exact solutions to large traveling salesman problems are all cutting-plane based algorithms. This means that they work by generating the space of feasible tours and then repeatedly generating inequalities that cut off part of the space. The inequalities that turn out to be most useful are *subtour elimina-*

*Supported by a National Science Foundation Graduate Fellowship, by NSF Grant CCR-9010517, and grants from Mitsubishi and OTL.

†Support provided by a Walter Burke Research Initiation Award and a Dartmouth College Research Initiation Award.

tion 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, these algorithms for the traveling salesman problem must solve a large number of min-cut problems (see [LLKS85] for a survey of the area). Padberg and Rinaldi [PR90] recently reported that the solution of min-cut problems was the computational bottleneck in their state-of-the-art cutting-plane based algorithm. They also reported that min-cut problems are the bottleneck in many other cutting-plane based algorithms for combinatorial problems whose solutions induce connected graphs. Applegate [App92] made similar observations and also noted that an algorithm to find *all* small cuts might be even more useful.

1.3 Background

The first known algorithm for the min-cut problem used the duality between s-t minimum cuts and s-t maximum flows [FF56, EFS56]. An s-t maximum flow algorithm can 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 showed that the minimum cut could be found by solving only $n-1$ maximum flow problems. The algorithmic result was accompanied by an insight into the structure of minimum cuts, as they showed that among the $\binom{n}{2}$ different s-t minimum cuts, there are only $n-1$ unique cut values. 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 influenced the development of min-cut algorithms for 25 years. The prevailing belief was that the best way to develop a faster min-cut algorithm was to develop a better maximum flow algorithm or a better method of performing a series of maximum flow computations.

Maximum flow algorithms have become progressively faster. Currently, the fastest deterministic algorithms are based on the work of Goldberg and Tarjan [GT88]. One developed by King, Rao, and Tarjan runs in $O(mn + n^{2+\epsilon})$ time [KRT92] (and has since been improved), and one developed by Phillips and Westbrook runs in $O(mn \log_{m/n} n)$ time provided $m > n \log^2 n$ [PW92]. The fastest randomized algorithm runs in $O(mn + n^2 \log^2 n)$ time (Cheriyan, Hagerup, and Maheshwari [CHM90]). 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 separately computing them. Podderyugin [Pod73], Karzanov and Timofeev [KT86], and Matula [Mat87] independently discovered several methods for determining edge connectivity in unweighted graphs in $O(mn)$ time. These algorithms all rely on the fact that the graph is unweighted. Hao and Orlin [HO92] extended this idea to 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 use the fastest such algorithm, that of Goldberg and Tarjan, to find a minimum cut in $O(mn \log(n^2/m))$ time.

Recently, two approaches that find minimum cuts *without* computing any maximum flows have been developed. One approach, developed by Gabow [Gab91], is based on a matroid characterization of the min-cut problem. He uses that approach to find the minimum cut of an unweighted graph in time $O(m + c^2 n \log(n^2/m))$ time, where c is the value of the minimum cut.

The second new approach is based on repeatedly identifying and *contracting* 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 which 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. Karger [Kar93] developed the Contraction Algorithm, which uses uniform random selection to choose an edge to contract. This algorithm can be used to find a minimum cut in $O(n^2 m \log n)$ time and is the basis of our new work.

Less work has been done on parallel solutions to the min-cut problem. For 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

min-cut bounds		unweighted		weighted	
		undirected	directed	undirected	directed
sequential time	previous	$c^2 n \log \frac{n^2}{m}$ [Gab91]		$mn + n^2 \log n$ [NI92]	$mn \log \frac{n^2}{m}$ [HO92]
	this paper	$n^2 \log^3 n$		$n^2 \log^3 n$	
processors used in $\mathcal{RNC}$	previous	cn^3 [Kar93]	$n^{4.37}$ [KUW86, GP85]	mn^2 [Kar93]	$\mathcal{P}$-complete [GSS82]
	this paper	n^2		n^2	

Figure 1: Bounds For the Min-Cut Problem

yield $\mathcal{RNC}$ algorithms for s-t minimum cuts. We can find a minimum cut by performing n of these computations in parallel. Unfortunately, the processor bounds are quite large—the best bound, using Galil and Pan's [GP85] adaptation of [KUW86], is $n^{4.37}$.

These unweighted graph algorithms can be extended to work for weighted graphs by representing an edge of weight w by w parallel edges. If W is the sum of the weights of all the edges 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. Even if we combine these algorithms with the scaling ideas of Edmonds and Karp [EK72], as suggested by Karp, Upfal and Wigderson [KUW86], the running times are proportional to $\log W$ and hence the algorithms are not in $\mathcal{RNC}$ unless $W = O(n^{\log^{O(1)} n})$.

This lack of an $\mathcal{RNC}$ algorithm is unsurprising. Goldschlager, Shaw, and Staples [GSS82] showed that the s-t min-cut problem on weighted directed graphs is $\mathcal{P}$-complete. Karger [Kar93] gives a simple reduction which proves that the directed min-cut problem is also $\mathcal{P}$-complete. On the other hand, he proves that on *undirected* graphs, the min-cut problem is in $\mathcal{RNC}$ even with arbitrarily large edge weights. His algorithm uses mn^2 processors.

While all of these algorithms are $\mathcal{RNC}$ algorithms, at least for unweighted graphs, they are not efficient. Of the above algorithms, that of Karger does the least work, $O(mn^2 \log^3 n)$, but this still exceeds by an $\Omega(n)$ factor the work done by the best sequential min-cut algorithms. These results, along with our new bounds, are summarized in the Figure 1 (c denotes the value of the minimum cut).

1.4 Our contribution

We present the most efficient known minimum cut algorithm. It is an improved implementation of Karger's Contraction Algorithm [Kar93]. Like that algorithm, it in fact finds *all* minimum cuts. The sequential version runs in $O(n^2 \log^3 n)$ time and thus improves on previous running times by a $\tilde{\Theta}(m/n)$ factor. The parallel version takes polylogarithmic time using n^2 processors on a PRAM. It is thus the first *efficient* $\mathcal{RNC}$ algorithm for the min-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). We give a slightly faster algorithm for a class of graphs which includes all planar graphs.

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

Our algorithm also improves on several other results of [Kar93]. The *minimum weight r-way cut problem* is the problem of identifying a minimum weight set of edges whose removal divides the graph into r nonempty pieces. We improve the algorithm of [Kar93] to use $\tilde{O}(n^{2(r-1)})$ work, yielding the fastest known sequential algorithm together with an efficient $\mathcal{RNC}$ implementation. We also improve the best known algorithms for finding all approximately minimum cuts and for finding the cactus representation of all minimum cuts.

Section 2 reviews the Contraction Algorithm. Sections 3, 4, and 5 contain our new results. Section 6 concludes with some discussion and open problems.

2 The Contraction Algorithm

We begin by restricting our attention to unweighted multigraphs (*i.e.*, graphs which may have multiple edges between vertices), and presenting an abstract version of the Contraction Algorithm. Although certain changes must be made to allow an efficient implementation, this version of the algorithm is particularly intuitive and easy to analyze. A more detailed exposition may be found in [Kar93].

Assume initially that we are given a multigraph $G(V, E)$ with n vertices and m edges. The Contraction Algorithm uses one fundamental operation, *contraction* of graph vertices. To contract two vertices v_1 and v_2, replace them by a new 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 which have the same other endpoint; instead, we create multiple instances of those edges. However, we remove edges which connect v_1 and v_2 to eliminate self loops. The Contraction Algorithm is described in Figure 2.

> **repeat** until two vertices remain
>
> **choose** an edge at random
>
> **contract** its endpoints

Figure 2: 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 in an obvious way: each side corresponds to the vertices contained in one of the metavertices.

Theorem 2.1 *A particular minimum cut in G is produced by the Contraction Algorithm with probability $\Omega(n^{-2})$.*

Proof: Details may be found in [Kar93]. The key idea is that a graph with r vertices and minimum cut c must have minimum degree c, and thus $rc/2$ edges. It follows that a randomly chosen edge is in the minimum cut with probability at most $2/r$. The probability that we never contract a min-cut edge through all $n-2$ contractions is thus at least

$$(1-\frac{2}{n})(1-\frac{2}{n-1})\cdots(1-\frac{2}{3}) = \binom{n}{2}^{-1} = \Omega(n^{-2}).$$

□

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 corollary of Theorem 2.1:

Corollary 2.2 *A particular minimum cut survives contraction to k vertices with probability at least $k(k-1)/n(n-1)$, which is $\Omega((k/n)^2)$.*

Direct implementation of the Contraction Algorithm as described above is undesirable. Such an implementation would require complex data structures to support contraction of vertices and would apply only to unweighted graphs. However, these problems can be circumvented. Graphs with integer edge weights can be viewed as compact representations of multigraphs, where an edge of weight w corresponds to w parallel multigraph edges. By working directly with this compact representation, Karger [Kar93] describes how to implement contraction to k vertices in $O(m)$ time on unweighted graphs, and in $O(m \log^2 n)$ time on weighted graphs. He also shows how the Contraction Algorithm can be parallelized to run in $\mathcal{RNC}$ using m processors.

The most direct application of the Contraction Algorithm is as follows: since a particular minimum cut has an $\Omega(n^{-2})$ probability of surviving contraction to two vertices, we perform $O(n^2 \log n)$ independent runs of the Contraction Algorithm to get a high probability of finding that minimum cut at the end of one of them. Each contraction takes $O(m \log^2 n)$ time, so the total running time of this approach is $(mn^2 \log^3 n)$. This is the algorithm of [Kar93].

3 Faster Implementation of the Contraction Algorithm

A first step towards our improved algorithm is an improvement in the running time of the Contraction Algorithm from $O(m \log^2 n)$ to $\Theta(n^2)$. Though this may appear to be a step backwards, it is useful for our future application of the Contraction Algorithm to graphs which may have $\Omega(n^2)$ edges.

An algorithm is said to be *strongly polynomial* if the number of operations it performs can be bounded independent of the size of the input numbers. The algorithm we present here does not have this property, since it generates random numbers of size equal to the sum of the edge weights. However, Karger [Kar93] shows how to round and scale a graph's edge weights so as to reduce to a problem in which all edge weights are polynomial in n. That reduction works equally well in this new algorithm (see Section 5), so for the remainder of this discussion we can assume that all edge weights are polynomially bounded.

We use a particularly simple data structure to implement the contraction algorithm: 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 (u, v) is not an edge, either because it was not in the initial graph or because it was contracted, then $W(u,v)=0$. We also maintain the total (weighted) degree $S(u)$ of each vertex u, thus $S(u)=\sum_v W(u,v)$. When parallel edges are created by a contraction, we immediately replace the two edges by one new edge whose weight is the sum of the weights of the two parallel edges.

We first consider the problem of finding an edge to contract. Our goal is to choose an edge (u,v) with probability proportional to $W(u,v)$. To do so, simply choose a first endpoint u with probability proportional to $S(u)$, and then choose a second endpoint v with probability proportional to $W(u,v)$. To choose a first enpoint, we compute $\sigma=\sum_v S(v)$, choose a random integer r uniformly at random between 0 and σ, and then compute the smallest index u such that $\sum_{v=1}^{u} \geq r$. We then choose an edge incident to u by applying an analogous procedure to the values $W(u,v)$. Thus an edge can be selected in $O(n)$ time.

The following lemma, similar to one used by Klein, Plotkin, Stein and Tardos [KPST91], proves the correctness of this procedure.

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

Proof: Recall $\sigma=\sum_v S(v)$.

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

□

We now show how to implement a contraction. Given W and S, which represent a graph G, we explain how to update W and S 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. This 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 sets row v and column v to zero. Intuitively, it moves all edges incident on v to u. W and S now 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 $S(u)$ and $S(v)$ are updated accordingly. Clearly, this procedure can be implemented in $O(n)$ time. Summarizing, in $O(n)$ time we can choose an edge and contract it. In order to contract to k vertices we only need to contract an edge $n-k\leq n$ times. This yields the following result:

Procedure to contract edge (u,v)

Let $S(u)\leftarrow S(u)+S(v)-2W(u,v)$

Let $S(v)\leftarrow 0$

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 3: Contracting an Edge

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 $W(u,v)$.

For the rest of this paper, we will use the Contraction Algorithm as a subroutine *Contract*(G,k), which 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 $k(k-1)/(n(n-1))$, 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.

4 Recursive Application of the Contraction Algorithm

We now make the more significant improvement, and show how to share work among the different trials of the Contraction Algorithm so as to reduce the running time of the Contraction Algorithm to $\tilde{O}(n^2)$.

Consider the contractions performed in one trial of the Contraction Algorithm. The first contraction has a reasonably low probability of contracting a cut edge, namely $2/n$. On the other hand, the last contraction has a much higher probability of contracting a cut edge, namely $2/3$. This suggests that the Contraction Algorithm works well initially, but has somewhat 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 min-cut algorithms, and this indeed yields some improvement—for example, if we contract to $m^{1/3}$ vertices and then run the Nagamochi-Ibaraki algorithm, then the running time (given the repetitions needed to ensure a high probability of success) improves to $O(n^2 m^{1/3})$. However, a better observation is that an algorithm which is more likely to succeed than the Contraction Algorithm is *two* trials of the Contraction Algorithm.

This suggests the Recursive Contraction Algorithm RC described in Figure 4. 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 $n/\sqrt{2}$ vertices, we ensure a roughly 50% probability of not contracting a min-cut edge, so that the expected number of successful contractions during the two trials is 1. We then recursively apply the Recursive Contraction Algorithm on the partially contracted graph.

Algorithm $RC(G, n)$

input A graph G of size n.

if $n = 2$

then examine the implied cut of the original graph

else repeat <u>**twice**</u>

$G' \leftarrow \text{Contract}(G, n/\sqrt{2})$
$RC(G', n/\sqrt{2})$.

Figure 4: The Recursive Contraction Algorithm

Lemma 4.1 *The Recursive Contraction Algorithm RC runs in $O(n^2 \log n)$ time.*

Proof: One level of recursion consists of two independent trials of contraction of G to $n/\sqrt{2}$ vertices followed by two recursive calls. This yields the following recurrence for the running time:

$$T(n) = 2\left(n^2 + T\left(\frac{n}{\sqrt{2}}\right)\right). \qquad (1)$$

This recurrence is solved by

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

and the depth of the recursion is $2\log_2 n$. □

Observe that since this recurrence is meant to apply to the contracted graphs as well, it is impossible to determine 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 running time of the Contraction Algorithm. This is the reason we modified the Contraction Algorithm to run in $O(n^2)$ time rather than $O(m \log n)$ time.

The space needed to support the recursion is also simple to determine: we have to store one graph at each level of the recursion, where the graph at the k^{th} level has $n_k = n/\sqrt{2^k}$ vertices. Since at each level the graph has no more than n_k^2 edges and can be stored using $O(n_k^2)$ space, the total storage needed is $\sum_k O(n_k^2) = O(n^2)$. We now analyze the probability that the algorithm finds the particular minimum cut we are looking for.

Lemma 4.2 *The Recursive Contraction Algorithm RC finds a particular minimum cut with probability $\Omega(1/\log n)$.*

Proof: Suppose that this minimum cut has survived up to some particular level in the recursion. It will survive to be output if two criteria are met: it must survive one of the graph contractions at this level, and it must be output by the recursive call following the contraction. Thus, each of the two trials 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 2.2, at least

$$\frac{(n/\sqrt{2})(n/\sqrt{2}-1)}{n(n-1)} = \frac{1}{2} - O(1/n).$$

This yields a recurrence $P(n)$ for a lower bound on the probability of success on a graph of size n:

$$P(n) = 1 - \left(1 - \frac{1}{2}P\left(\frac{n}{\sqrt{2}}\right)\right)^2 - O(1/n).$$

It can be shown that the $O(1/n)$ is negligible, so we ignore it in this sketch of the solution. We solve this recurrence through a change of variables. Letting $p_k = P(\sqrt{2^k})$, the recurrence above can be rewritten and simplified as

$$p_{k+1} = p_k - \frac{1}{4}p_k^2.$$

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

$$z_{k+1} = z_k + 1 + 1/z_k.$$

It follows by induction that

$$k < z_k < k + H_{k-1} + 3,$$

where H_k is the k^{th} harmonic number. Thus $z_k = k + O(\log k)$ and $p_k = \Theta(1/k)$. It follows that

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

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

Now observe that this analysis holds regardless of which minimum cut we fix our attention on initially. We have therefore proven the following:

Theorem 4.3 *All minimum cuts in an arbitrarily weighted undirected graph of n vertices can be found with high probability in $O(n^2 \log^3 n)$ time and $O(n^2)$ space.*

Proof: Repeating the Recursive Contraction Algorithm $O(\log^2 n)$ times gives a high success probability. □

It is noteworthy that unlike the best algorithms for maximum flow this algorithm uses no non-trivial data structures. We believe that the algorithm will be practical and easy to code.

We now take a more careful look at the computation performed by the algorithm. We can express the running of the 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 $\sqrt{2}$. 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 $2\log_2 n$, and it thus has n^2 leaves. This shows that the improvement over the algorithm of [Kar93] comes not from generating a smaller tree (the trials there also defined 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.

This algorithm can be parallelized to yield the first efficient $\mathcal{RNC}$ algorithm for the min-cut problem. Consider the computation tree just described. The sequential algorithm examines this computation tree using a depth-first traversal of the vertices. To solve the problem in parallel, we instead do a breadth-first traversal. Karger [Kar93] shows how to perform a contraction to k vertices in $\mathcal{RNC}$. 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 4.4 *The min-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).

5 Other results

The techniques used in this paper can be combined with ideas from [Kar93] to give the best known algorithms for other related problems. The proofs of the following theorems follow easily from various proofs in [Kar93].

Theorem 5.1 *The minimum r-way cut problem (that of minimizing the weight of edges cut when partitioning the graph into r groups) can be solved in $O(n^{2(r-1)} \log^3 n)$ time, or in $\mathcal{RNC}$ using $n^{2(r-1)}$ processors.*

Theorem 5.2 *All cuts with weight within a multiplicative factor α of the minimum cut can be found in $O(n^{2\alpha} \log^3 n)$ time.*

Proof: Change the reduction factor from $\sqrt{2}$ to $\sqrt[2\alpha]{2}$ in the Recursive Contraction Algorithm. □

Because of this theorem, we can apply the weight rounding technique of [Kar93] to make the Recursive Contraction Algorithm strongly polynomial. Essentially, we round the edge weights in such a way 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 which finds all approximate minimum cuts will find the original minimum cut. It is arranged that the parameter $\alpha = 1 + 1/n$, so that the running time is changed only by a constant factor.

Naor and Vazirani [NV91] give a parallel algorithm that computes the concise *cactus representation* of all min-cuts in a graph. The dominant step of their algorithm is a procedure to enumerate all the minimum cuts in a graph. They use an algorithm based on [MVV87], discussed in Section 1. Replacing that step with the Recursive Contraction Algorithm reduces the processor cost.

We also observe that our algorithm runs faster on graphs with excluded dense minors. Assume that we have a graph such that all r-vertex minors have $O(r^{2-\epsilon})$ edges for some some positive constant ϵ. Then we can be sure that at all times during the execution of the Recursive Contraction algorithm the contracted graphs of n_k vertices will never have more than $n_k^{2-\epsilon}$ edges. Therefore if we use the $O(m \log n)$ running time implementation of the Contraction Algorithm from [Kar93], we can be sure that the running time of the Contraction Algorithm will be $O(n_k^{2-\epsilon} \log n)$. We can make this substitution in the recurrence of Equation 1 and prove the following result:

Theorem 5.3 *Let G have the property that all n-vertex minors have $O(n^{2-\epsilon})$ edges for some some positive constant ϵ. Then with high probability the Recursive Contraction algorithm finds a minimum cut of G in $O(n^2 \log^2 n)$ time.*

Planar graphs fall into this class, as all r-vertex minors have $O(r)$ edges.

6 Conclusion

We have given efficient and simple algorithms for the min-cut problem, yet several interesting open questions remain. One desirable result would be to find a deterministic version of the algorithm with the same time bounds. Another would be to find a faster randomized algorithm. There are several probably unnecessary logarithmic factors in the running time of the Recursive Contraction Algorithm. However, it seems unlikely that the techniques presented here will yield an $o(n^2)$ algorithm, as our algorithm finds not just one minimum cut, but all of them. As there can be $\Omega(n^2)$ minimum cuts in a graph, any algorithm that finds a minimum cut in $o(n^2)$ time will either have to somehow break the symmetry of the problem and avoid finding all the minimum cuts, or will have to produce a concise representation (for instance the cactus representation) of all of them. The ideal, of course, would be an algorithm that did this in linear ($O(m)$) time.

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 the s-t minimum cut is not helpful in finding an s-t maximum flow. However, the question of whether a minimum cut may help one find an s-t maximum flow remains open.

Another obvious question is whether any of these results can be extended to directed graphs. It is 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 min-cut algorithm of Gomory and Hu [GH61] not only found the minimum cut, but found a *flow equivalent tree* which 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 intruiging open question is whether the methods in this paper can be extended to produce a Gomory-Hu tree.

Acknowledgements

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

[App92] D. Applegate, 1992. personal communication.

[CHM90] J. Cheriyan, T. Hagerup, and Kurt Mehlhorn Can a maximum flow be computed in $o(nm)$ time? *ICALP*, 1990.

[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.

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

[FF56] L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canad. J. Math.*, 8:399–404, 1956.

[Gab91] Harold N. Gabow. A matroid approach to finding edge connectivity and packing arborescences. *Proceedings of the 23^{rd} Annual ACM Symposium on Theory of Computing*, May 1991.

[GH61] R. E. Gomory and T. C. Hu. Multiterminal network flows. *J. SIAM*, 9:551–570, 1961.

[GP85] Z. Galil and V. Pan. Improved processor bounds for algebraic and combinatorial problems in $\mathcal{RNC}$. *Proceedings of the 26^{th} Annual Symposium on Foundations of Computer Science*, pages 490–495, 1985. To appear in *Journal of the ACM.*

[GSS82] L. Goldschlager, R. Shaw, and J. Staples. The maximum flow problem is log-space complete for $\mathcal{P}$. *Theoretical Computer Science*, 21:105–111, 1982.

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

[HO92] J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut in a graph. *Proceedings of the 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 165–174, January 1992.

[FF62] L. R. Ford Jr. and D. R. Fulkerson. *Flows in networks.* Princeton University Press, 1962.

[Kar93] D. Karger. Global min-cuts in $\mathcal{RNC}$, and other ramifications of a simple min-cut algorithm. *Proceedings of the 4^{th} ACM-SIAM Symposium on Discrete Algorithms*, January 1993.

[KPST91] P. Klein, S. A. Plotkin, C. Stein, and É. Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. Technical Report 961, School of Operations Research and Industrial Engineering, Cornell University, 1991. A preliminary version of this paper appeared in *Proceedings of the 22^{nd} Annual ACM Symposium on Theory of Computing*, pages 310–321, 1990. To appear in SIAM J. Computing.

[KRT92] V. King, S. Rao, and R. Tarjan. A faster deterministic maximum flow algorithm. *Proceedings of the 3^{rd} ACM-SIAM Symposium on Discrete Algorithms*, pages 157–164, January 1992.

[KT86] A. V. Karzanov and E. A. Timofeev. Efficient algorithm for finding all minimal edge cuts of a non-oriented graph. *Kibernetika*, 22:8–12, 1986. Translation in Cybernetics 22, 1986, pages 156–162.

[KUW86] R. Karp, E. Upfal, and A. Wigderson. Constructing a perfect matching is in random $\mathcal{NC}$. *Combinatorica*, 6:35–48, 1986.

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

[Mat87] D. W. Matula. Determining edge connectivity in $o(nm)$. *Proceedings of the 28^{th} Annual Symposium on Foundations of Computer Science*, pages 249–251, October 1987.

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

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

[NV91] D. Naor and V. Vazirani. Representing and enumerating edge connectivity cuts in $\mathcal{RNC}$. *Proceedings of the Second Workshop on Algorithms and Data Structures*, pages 273–285, 1991. Published as Lecture Notes in Computer Science 519, Springer-Verlag.

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

[PQ82] J.C. Picard and M. Querayne. Selected applications of minimum cuts in networks. *INFOR.*, 20:394–422, November 1982.

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

[PW92] S. Phillips and J. Westbrook. Online load balancing and network flow. To appear in Proceedings of the 25^{th} Annual ACM Symposium on the Theory of Computing, 1993.

[Ram87] V. Ramachandran. Flow value, minimum cuts and maxmium flows, 1987. Unpublished manuscript.