Chapter 3
Global Min-cuts in $\mathcal{RNC}$, and Other Ramifications of a Simple Min-Cut Algorithm*

David R. Karger†

Abstract This paper presents a new algorithm for finding global min-cuts in weighted, undirected graphs. One of the strengths of the algorithm is its extreme simplicity. This randomized algorithm can be implemented as a strongly polynomial sequential algorithm with running time $\tilde{O}(mn^2)$, even if space is restricted to $O(n)$, or can be parallelized as an $\mathcal{RNC}$ algorithm which runs in time $O(\log^2 n)$ on a CRCW PRAM with $mn^2 \log n$ processors. In addition to yielding the best known processor bounds on unweighted graphs, this algorithm provides the first proof that the min-cut problem for weighted undirected graphs is in $\mathcal{RNC}$. The algorithm does more than find a single min-cut; it finds all of them. The algorithm also yields numerous results on network reliability, enumeration of cuts, multi-way cuts, and approximate min-cuts.

1 Introduction

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 S and T so as to minimize the number of edges crossing from S to T (if the graph is weighted, we wish to minimize the total weight of crossing edges). Throughout this paper, the graph is assumed to be connected, since otherwise the problem is trivial. The problem actually comes in two flavors: in the *s-t min-cut* problem, we require that the two specific vertices s and t be on opposite sides of the cut; in what will be called the *min-cut* problem, or for emphasis the *global min-cut* problem, there is no such restriction.

1.1 Previous Work. The oldest known way to compute min-cuts is to use their well known duality with max-flows [FF56, FF62]. Computation of an s-t max-flow allows the immediate determination of an s-t min-cut. The best presently known sequential time bound for max-flow is $O(mn \log(n^2/m))$, found by Goldberg and Tarjan [GT88]. Global min-cuts can be computed by minimizing over s-t max-flows; Hao and Orlin [HO92] show how the max-flow computations can be pipelined so that together they take no more time than a single max-flow computation; thus the global min-cut problem can be solved in the same $\tilde{O}(mn)$ running time.[1]

Recently, progress has been made in special cases of the min-cut problem. On unweighted graphs, the min-cut problem is often known as the edge-connectivity problem. Gabow [Gab91] shows how to find the edge-connectivity c of a graph in time $O(cn \log(n^2/m))$. On weighted, undirected graphs, the algorithm of Nagamochi and Ibaraki [NI92] computes the min-cut in time $O(mn+n^2 \log n)$. These algorithms make no use of max-flow computations.

Work has also been done on parallel solutions to the min-cut problem. Goldschlager, Shaw, and Staples [GSS82] showed that the s-t min-cut problem on weighted directed graphs is P-complete. This is also true for the global min-cut problem (see section 4.2). In the special case of unweighted directed or undirected graphs, the matching algorithm of Karp, Upfal and Wigderson [KUW86], together with a reduction described by Mulmeley, Vazirani and Vazirani [MVV87], can be used to find s-t max-flows and min-cuts in $O(\log^2 n)$ time using $mn^{3.5}$ processors. An alternative approach of Galil and Pan [GP88] uses $n^2M(n)$ processors, where $M(n)$ is the processor cost for multiplying two matrices (presently about $n^{2.37}$). In undirected graphs, fixing a vertex s and finding s-t min-cuts for all vertices t identifies a min-cut; this requires performing n min-cut computations in parallel at a total cost of $mn^{4.5}$ or $n^2M(n)$ processors. Either algorithm can be extended to weighted graphs by treating an edge of weight w as a collection of w unweighted edges. How-

*Supported by a National Science Foundation Graduate Fellowship

†Department of Computer Science, Stanford University. karger@cs.stanford.edu

[1] The notation $\tilde{O}(f)$ denotes $O(f \text{ polylog } f)$

min-cut bounds		unweighted		weighted	
		undirected	directed	undirected	directed
sequential time		$cn\log\frac{n^2}{m}$ [Gab91]		$mn+n^2\log n$ [NI92]	$mn\log\frac{n^2}{m}$ [GT88, HO92]
processors used in $\mathcal{RNC}$	previous work	$mn^{4.5}$ or $n^2M(n)$ [KUW86, MVV87, GP88]		?	$\mathcal{P}$-complete [GSS82]
	new	cn^3		mn^2	

Figure 1: Bounds For the Min-Cut Problem

ever, this makes the processor cost polynomial in the total weight of the edges, and therefore the algorithm is only in $\mathcal{RNC}$ when edge weights are represented in unary. Until now there has been no known $\mathcal{RNC}$ algorithm for the general weighted case. These results are summarized in the Figure 1 (c denotes the value of the min-cut).

1.2 New Results. This paper presents a new framework for the computation of global min-cuts. The surprisingly simple Contraction Algorithm described in Section 2 finds global min-cuts in weighted undirected graphs without any use of max-flow or s-t min-cut computations. In Section 3 we describe how to implement the algorithm with a sequential running time $\tilde{O}(mn^2)$. No complex data structures are used. The algorithm is parallelized in section 4 to yield the first $\mathcal{RNC}$ algorithm for global min-cuts of weighted graphs. Extensions to the algorithm, including a practical time-processor tradeoff, an approximation algorithm, and an improved construction of the cactus representation of min-cuts described by Naor and Vazirani in [NV91], are described in Section 5

Section 6 describes combinatorial ramifications of the Contraction Algorithm. The algorithm yields theorems which bound the number of distinct minimal or small cuts which a graph may have. These results are in interesting counterpoint to work of Vazirani and Yannakakis [VY92] on enumeration of small cuts. In section 7, we show how this counting allows an accurate estimation of how likely a graph is to become disconnected if its edges fail with certain probabilities. This is relevant to the practical problem of network reliability, which is studied in, for example, Colbourn's book [Col87]. It is also closely related to a result of Margulis [Mar75], discussed in [Bol85], which proves the existence of a threshold function for connectedness in graphs.

With slight modifications described in Section 8, the Contraction Algorithm can be used to compute minimal multi-way cuts. The sequential time bound improves on the multi-way cut algorithm of [GH88], and the parallel version shows that the minimal r-way cut problem is in $\mathcal{RNC}$ for any constant r. In contrast, it is shown in [DJP+92] that the multiway cut problem in which k specified vertices are required to be separated (*i.e.*, a generalization of the s-t min-cut problem) is $\mathcal{NP}$-complete for any $k > 2$.

2 The Contraction Algorithm

We now present an abstract version of the Contraction Algorithm. Although certain changes must be made for efficient implementation, this version of the algorithm is particularly intuitive and easy to analyze.

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 give v 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.

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

Proof. Fix attention on some specific min-cut of c edges (from now on, the term "min-cut edge" refers only to edges in this particular min-cut). First, observe that if we never select a min-cut edge during the Contraction

Algorithm, then the two metavertices we end up with must define the min-cut. To see this, consider two vertices on opposite sides of the min-cut. If they end up in the same metavertex, then there must be a path between them consisting of edges which were contracted. However, any path between them crosses the min-cut, so a min-cut edge would have had to be contracted. This contradicts our assumption.

Next observe that after each contraction, the min-cut of the new graph must still be at least c. This is because contracting vertices u and v simply restricts attention to cuts of the original graph in which u and v are on the same side.

Each contraction in the above loop reduces the number of vertices in the graph by one. Consider the contraction during which the graph has r vertices. Since the contracted graph has a min-cut of at least c, it must have minimum degree c, and thus at least $rc/2$ edges. However, only c of these edges are in the min-cut. Thus, a randomly chosen edge is in the min-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

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

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

COROLLARY 2.1. *If we perform $O(n^2 \log n)$ independent contractions to two vertices, we find a min-cut with high probability. In fact, with high probability we find* every *min-cut.*

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 (we are in fact emulating Kruskal's minimum spanning tree algorithm [Kru56]). If we then remove the heaviest edge in the minimum spanning tree, the two components which result have an $\Omega(n^{-2})$ chance of defining a particular min-cut.

3 Sequential Implementation

Our most interesting new results are in the parallel version of this algorithm; however, it is easier to explain certain concepts by describing a sequential implementation and then showing how to parallelize it. To show how to implement the Contraction Algorithm, we need only show how to implement a single trial, since it is simple to remember the best result which occurs during the $O(n^2 \log n)$ trials. We begin by presenting a simple method for unweighted graphs, and then show how to improve the running time by working a little harder. We then extend to the case of graphs with polynomially bounded edge weights, and finally to arbitrarily weighted graphs.

3.1 Unweighted Graphs. A minor reformulation of the Contraction Algorithm is convenient. If we performed contractions one at a time, we would need to use complex data structures to update the adjacency lists of the graph. Instead, we simulate the performance of many contractions at once. We begin by generating a random permutation of the edges. Imagine contracting edges in the order in which they appear in the permutation, until only two vertices remain. This is clearly equivalent to the first formulation of the Contraction Algorithm. We can immediately say that with probability $\Omega(n^{-2})$, a random permutation will yield a contraction to two vertices which determine a particular min-cut.

Consider any such permutation. It has a prefix such that the set of edges in this prefix induces two connected components which are the two sides of the min-cut. All we need to do is determine how long the prefix is. Binary search solves this problem, because any prefix which is too short will yield more than two connected components, and any prefix which is too long will yield only one. The correct prefix can therefore be determined using $\log m$ connected component computations, each requiring $O(m)$ time. The total running time of the trial is therefore $O(m \log m)$.

We can improve this running time by taking better advantage of the connected component computations. Given the permutation, use $O(m)$ time to identify the connected components induced by the first $m/2$ edges. 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. If not, then we can contract the first $m/2$ edges all at once in $O(m)$ time by finding connected components, and search for the correct prefix in the remaining $m/2$ edges. Either way, in $O(m)$ time, we have reduced the problem size to $m/2$. Thus we finish in $O(m)+O(m/2)+O(m/4)+\cdots = O(m)$ time.

The two methods described above both require $O(m)$ space. We can improve this bound to $O(n)$ space if we are willing to sacrifice some time. The only part of the algorithm which requires $O(m)$ space is the generation of an edge permutation. If, instead, the edges are stored in read-only memory, we can avoid generating the permutation. We use the union-find data structure of [AHU83] to identify sets of vertices which have been contracted together. We 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. We have a high probability of choosing every edge at least once after making $O(m \log m)$ choices, and we will necessarily contract the graph to two vertices some time before this. Each choice requires one find operation, and we will also perform a total of $n-2$ union operations. Therefore the total running time of a trial will be $O(m \log m)$. The use of path compression in the union-find data structure provides no improvement in the running time, which is dominated by the requirement that every edge be sampled at least once.

COROLLARY 3.1. *With high probability, the Contraction Algorithm finds all min-cuts of an multigraph with m edges and n vertices in time $O(mn^2 \log n)$ and space $O(m)$, or in time $O(mn^2 \log^2 n)$ if space is restricted to be $O(n)$.*

3.2 Weighted Graphs. It is easy to apply the Contraction Algorithm to integer weighted graphs: just treat an edge of weight w as a collection of w parallel edges. This might appear to cause the running time to become dependent on the sum of the edge weights, but we show how to avoid this. We begin by assuming that edge weights are integers with maximum value polynomial in the problem size, and then clear up a few details to make the algorithm strongly polynomial.

Observe that the entire edge permutation is not necessary in the computation, since as soon as a multigraph edge is contracted, all the other edges with the same endpoints vanish. In fact, all that matters is the earliest place in the permutation that an edge with particular endpoints appears. This information suffices to tell us in which order vertices of the graph are contracted: we contract u and v before w and x precisely when the first (u, v) edge in the permutation precedes the first (w, x) edge 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. We can then use the permutation to contract the graph in the same fashion as was described for unweighted graphs.

We present two separate methods for generating a permutation. While they may not be the best possible sequential algorithms, they have the advantage of being easy to parallelize.

3.2.1 Exponential Variates. The first method is to directly model the weighted graph as a multigraph. One way we can generate a permutation of the multigraph edges is by assigning a uniform random score to each edge and sorting according to score. In this case, the first appearance of a multigraph edge with w copies is determined by the minimum of w randomly chosen scores. Consider multiplying each edge by a large constant weight k, so that an edge of weight w corresponds to wk multigraph edges. This scales the value of the min-cut without changing its structure. Suppose we gave each multigraph edge 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

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

If we can generate an exponential random variable in $O(1)$ time, then we can simulate a permutation in $O(m)$ time (note that we do not actually have to sort based on the scores: we can use median finding to do a binary search of the edges in $O(m)$ time, as was described in Section 3.1). If all we have is coin flips, it is possible to use them to generate approximately exponential distributions in polylogarithmic time and introduce a negligible error in the computation. This technique will be described in the full paper.

3.2.2 Iterated Sampling. Our second method avoids the mathematical computations needed to generate exponential variates if one has access only to coin flips or uniform integer distributions. 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. The following procedure can be used to choose one edge. First, from edge 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 $W_{i-1} < r < W_i$.

Once the cumulative weights are known, choosing an edge takes $O(\log n)$ time (based on the present assumption that W_m is polynomial in n). Since it takes linear time to recompute the cumulative distribution, it is undesirable to do this each time we wish to sample an edge. An alternative approach is to keep sampling from the original cumulative distribution, and ignore edges if we sample them more than once. Unfortunately, to ensure that all edges have been sampled once, we expect to need a number of samples equal to the sum of the edge weights. We solve this problem by combining the two approaches and recomputing the cumulative distribution only occasionally. We use the following lemma:

LEMMA 3.1. *With high probability, (weighted) sampling m times from a set of at most m edges yields a set of edges whose total weight is more than $1/3$ of the total weight of the entire set of edges.*

Proof. If the outcome of the lemma does not occur, there must be some set of edges which contains 2/3 of the total weight, such that no edge in this set is sampled. The probability of this happening is $1/3^m$. Since there are only 2^m different sets of edges, the probability that this happens with some set of edges is at most $2^m/3^m$, which is negligible.

We can therefore apply the procedure of Figure 3. A single iteration of this loop takes $O(m \log m)$ time.

```
repeat until no edges remain
    • compute the cumulative weight measures.
    • extend the permutation with $m$ samples
      from the remaining edges.
    • remove edges which were sampled at least
      once
```

Figure 3: Generating a Permutation

If the total weight of edges polynomial in n, then Lemma 3.1 shows that $O(\log n)$ iterations of the loop ensure that the total remaining weight of unsampled edges is less than 1, *i.e.* no edges remain and we have finished constructing a permutation.

We remark that the $O(n)$ space bound discussed for unweighted graphs can be achieved here as well. As before, we use the union-find data structure of [AHU83] 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 which have weight less than this threshold.

3.3 Strong Polynomiality. The exponential variable technique for generating permutations can be made strongly polynomial by approximating the exponential distribution appropriately; however, we will focus on the second technique.

Construction of the cumulative edge weights is easily strongly polynomial. To quickly select an edge from the cumulative distribution, even if the edge weights are large, let $M = n^{O(1)}$, generate s uniformly at random from $0, \ldots, M$, and choose the edge i such that $W_{i-1} < W_m s/M < W_i$. We have only a polynomially small probability of having a different result than we would if we used exact arithmetic, since such an error is introduced only if $W_m s/M$ and $W_m(s+1)/M$ specify different edges.

We also need to ensure that not too many iterations of the permutation generating loop of Figure 3 are needed. We use a very rough approximation to the min-cut to ensure that $O(\log n)$ iterations suffice 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 $O(m + n \log n)$ time [FT86]. It follows that any cut of the graph must cut an edge of weight at least W, so the min-cut has weight at least W. It also follows from the definition of W that there is a cut which does not cut any edge of weight exceeding W. This means the min-cut has weight less than $n^2 W$, since fewer than n^2 edges are in the graph, and at worst all edges of weight at most W are cut. This guarantees that no edge of weight exceeding $n^2 W$ can possibly be in the min-cut. We can therefore contract all such edges, without eliminating any min-cut in the graph. Afterwards the total weight of edges in the graph is at most $n^4 W$.

Since initially the total weight of edges was at most $n^4 W$, Lemma 3.1 proves that the amount of weight remaining unsampled after $O(\log n)$ iterations of Figure 3 is less than W. It follows that the portion of the permutation which we have constructed at this point must suffice to contract the graph to a single vertex, since otherwise we would have a cut of weight less than W (it could cut only the unsampled edges), which is less than the min-cut. We can therefore ignore the remaining unsampled edges and use the permutation prefix which we have constructed so far.

COROLLARY 3.2. *A single Contraction Algorithm trial on weighted graphs can be run in strongly polynomial $O(m \log^2 n)$ time, so the Contraction Algorithm can be run in $O(mn^2 \log^3 n)$ time.*

4 Parallel Complexity of Min-cut

This section demonstrates a significant difference in the complexity of the min-cut problem on directed and undirected graphs. Our parallelization of the Contraction Algorithm proves the the undirected min-cut problem is in $\mathcal{RNC}$. On the other hand, we show that the global min-cut problem on directed graphs is $\mathcal{P}$-complete.

4.1 Parallel Implementation. We now show how to parallelise the Contraction Algorithm to give an

$O(\log^2 n)$ time parallel algorithm which uses $mn^2 \log n$ processors. As before, the only real question is how to run a single trial of the Contraction Algorithm, since it is simple to run $O(n^2 \log n)$ trials in parallel and combine their results. We implement a trial as in the sequential case, by generating a permutation of the edges and contracting based on that permutation. $\mathcal{RNC}$ algorithms for connected components exist which run in $O(\log n)$ time on a CRCW PRAM [Gaz86] or in $O(\log n \log\log n)$ time on an EREW PRAM [KPN92], and use $O(m)$ processors. There is therefore no difficulty in performing the binary search on connected components which was described in the sequential algorithm. Thus we need only show how a linear number of processors can be used to generate an appropriately distributed permutation.

In the case of an unweighted graph, generating a permutation is trivial. Each processor takes one edge and assigns it a score chosen uniformly at random from the integers $1, \ldots, n^7$ (this large range guarantees that with high probability no two edges get the same score). We then sort the edges according to score in $O(\log n)$ time (using, *e.g.*, Cole's algorithm [Col88]). All of this requires only m processors per trial. This yields the result for unweighted graphs:

THEOREM 4.1. *All min-cuts in an unweighted multigraph can be found in $O(\log^2 n)$ time using $mn^2 \log n$ CRCW processors.*

The bottleneck in the runtime is caused by the binary search for connected components. If we increase the number of processors to $m^2n^2 \log n$, we can examine all prefixes of each permutation in parallel and achieve a running time of $O(\log n)$, even on an EREW PRAM. This matches the $\Omega(\log n)$ EREW lower bound of [CDR86], and closely approaches the $\Omega(\log n/\log\log n)$ CRCW lower bound of [Has86].

It remains to generalize the algorithm to the case of weighted graphs. We do this by parallelizing the sequential methods described in Section 3. The reduction to small edge weights can be parallelized using, for example, the parallel maximum spanning tree algorithm of [AS87] and the connected components algorithms described above. Once edge weights are small, permutation by assignment of exponentially distributed scores is simple to parallelize using a parallel sorting algorithm. It is also straightforward to parallelize a single iteration of the weighted sampling loop used in our second method, by assigning one processor to perform each of the m selections described there.

THEOREM 4.2. *The min-cut problem on arbitrarily weighted graphs can be solved in $\mathcal{RNC}$ in $O(\log^2 n)$ time using $mn^2 \log n$ CRCW processors.*

4.2 Comparison to Directed Graphs. The previous result shows a fundamental distinction between the min-cut problems on directed and undirected graphs. The s-t min-cut problem on directed graphs was shown to be $\mathcal{P}$-complete [GSS82]. A simple reduction shows that the global min-cut problem is also $\mathcal{P}$-complete for directed graphs. To find a minimum s-t cut using a global min-cut algorithm, simply add, for each vertex v, directed edges of infinite weight from t to v and from v to s. The global min-cut in this modified graph must have s on the inside and t on the outside and thus corresponds to the minimum s-t cut in the original graph.

The min-cut problem is therefore in the family of problems, such as reachability [NSW92], which presently have dramatically different difficulties on directed and undirected graphs.

5 Extensions of the Algorithm

5.1 Approximating the Min-cut. If we are looking only for a "small" cut, then it is possible to significantly reduce the amount of work required in the algorithm.

THEOREM 5.1. *With probability $n^{-2/k}$, a single trial of the Contraction Algorithm will yield a cut of weight kc.*

Proof. We return to the unweighted multigraph discussion. We again fix our attention on a particular min-cut. Suppose that at some point we have contracted to r vertices and have not yet seen a vertex of degree less than kc (if we have, than we have a corresponding cut of the desired size). Then the total number of edges in the graph is at least $kc/2$. It follows that we pick a min-cut edge with probability $2/kr$. Arguing as before, it follows that our probability of success over $n-2$ iterations is at least

$$\begin{aligned}
\prod_{u=3}^{n}(1-\frac{2}{ku}) &= \exp(\sum_{i=3}^{n}\ln(1-\frac{2}{ki})) \\
&\approx \exp(-\sum_{i=3}^{n}\frac{2}{ki}) \\
&\approx e^{(-2\ln n/k)} \\
&= \Omega(n^{-2/k}).
\end{aligned}$$

COROLLARY 5.1. *A cut within a factor of k of the min-cut can be found with high probability in $O(mn^{2/k}\log n)$ time.*

Proof. Because of the above theorem, we need only show that we can identify the smallest degree metavertex which arises during the contraction process.

Recall that the Contraction Algorithm can be simulated by assigning random ranks and running a minimum spanning tree algorithm. Given the minimum spanning tree, it is relatively simple to identify the smallest vertex which arose from a contraction. Details are left for the full paper.

5.2 A Time-Processor Tradeoff. The Contraction Algorithm may be effective in practice as a way to parallelize sequential min-cut algorithms. The key observation is that if we only contract the graph until it has been reduced to s vertices, then a particular min-cut survives with probability $\Omega((s/n)^2)$ (this is a simple extension of the original proof of correctness). This contracted graph will have at most $\min(m, s^2)$ edges. Assuming the min-cut survives, we can find it by running a sequential min-cut algorithm for a graph of size s. It follows that the Contraction Algorithm can be used by p processors to accelerate any sequential weighted graph algorithm by a factor of $\sqrt{p}$.

5.3 Fewer Processors for Unweighted Graphs. In the case of unweighted graphs, we can reduce the processor cost from mn^2 to n^3c. This provides no improvement in the worst case, since a graph with min-cut c may have as few as $nc/2$ edges, but it does improve performance on dense graphs with small min-cuts. This improvement is achieved by transforming the graph into one with $\tilde{O}(nc)$ edges, and running the original algorithm. We use the following lemma:

LEMMA 5.1. *If each edge of a graph is marked independently with probability p, and connected components induced by the marked edges are contracted, then with high probability the number of edges of the contracted graph is $O(n \ln n/p)$.*

Proof. The number of edges in the contracted graph is just the number of edges crossing between two different connected components induced by the marked edges. The number of different arrangements of connected components is certainly no more than the number of ways to partition the set of n vertices into at most n groups, namely n^n. For any given partition which cuts k edges, the probability that no crossing edge is chosen is $(1-p)^k \approx e^{-kp}$. The probability that k edges are cut in the partition resulting from the connected component construction is just the probability that for some partition with at least k crossing edges, no one of these k edges is chosen. This is at most $n^n e^{-kp} = e^{n \ln n - kp}$, which is negligible when $kp = \Omega(n \ln n)$.

We apply this lemma to our problem by letting the probability p in the lemma be $1/c$. If we mark edges and contract components which are connected by marked edges, then any particular min-cut has a constant probability of having none of its edges chosen. If this happens, then this min-cut will still be a min-cut in the contracted graph. It will happen with high probability after only $O(\log n)$ trials. In each trial, the lemma proves that the contracted graph will contain $\tilde{O}(nc)$ edges. We then apply the Contraction algorithm, using $\tilde{O}(n^2)$ trials on a graph of $\tilde{O}(nc)$ edges, yielding a total processor cost of $\tilde{O}(n^3c)$.

5.4 Cactus Representation For Min-Cuts. The set of all min-cuts in a graph has a simple and compact representation known as the cactus representation. The best presently known sequential algorithm for constructing the cactus ([NK92]) runs in time $O(mn)$. Naor and Vazirani [NV91] have shown how to construct this cactus representation in $\mathcal{RNC}$ when edge weights are represented in unary. The processor cost for their algorithm is $mn^{4.5}$. Both the processor cost and the restriction to unary edge weights stem from the same source, namely the need for an algorithm to compute individual min-cuts in $\mathcal{RNC}$. They use the algorithm of [KUW86]. If we instead use the Contraction Algorithm, both of these problems are eliminated. We therefore deduce:

THEOREM 5.2. *The cactus representation of an arbitrarily weighted graph can be computed in $\mathcal{RNC}$ using $mn^2 \log n$ processors.*

6 Combinatorial Ramifications

We now use the Contraction Algorithm to prove several interesting facts about the combinatorial structure of cuts in a graph. In particular, we show bounds on the number of small cuts in a graph. Vazirani and Yannakakis [VY92] perform a similar investigation with different results.

THEOREM 6.1. *The number of min-cuts in an arbitrarily weighted graph is at most $\binom{n}{2}$.*

Proof. The Contraction Algorithm can be viewed as a procedure for randomly generating cuts. We proved that any particular min-cut is generated with probability at least $p = \binom{n}{2}^{-1}$. It follows that there can be at most $1/p$ min-cuts.

We can perform a similar analysis of larger cuts:

THEOREM 6.2. *For k half an integer, the number of cuts of weight at most k times the graph min-cut is at most $2^{2k-1}\binom{n}{2k}$, which is less than n^{2k}.*

Proof. We consider the unweighted case; the extension to weights goes as before. Let k be half an integer, and c the min-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 the min-cut with probability at most $2k/r$.

If we do this until $r = 2k$, then the probability that the cut survives is

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

We can again use the algorithm to generate a random cut, although we must now add an extra step. 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-1} partitions, it follows that the probability of a particular cut being chosen is at least $2^{1-2k}\binom{n}{2k}^{-1}$.

When $k > n/2$ we can apply the obvious upper bound of 2^{n-1} to the number of cuts of this size.

COROLLARY 6.1. *For arbitrary real values of k, the number of cuts of size less than k times the min-cut is $O(n^{2k})$.*

Proof. Full Paper.

Vazirani and Yannakakis [VY92] derive bounds based on the *rank* of a cut relative to the others; we instead derive bounds based on the *value* of a cut relative to the others. Thus neither bound dominates the other.

COROLLARY 6.2. *The problem of enumerating all cuts within any constant factor of the min-cut is in $\mathcal{RNC}$.*

Consider the complete n-vertex graph in the context of these results. The min-cut there has value $n-1$. Any set of k vertices defines a cut of about kn edges. Thus the number of cuts of size about kn is $\binom{n}{k}$, a result which is strikingly close to the one we have derived. The cycle graph shows an even closer match for the case $c = 2$ and k an integer. Such a graph has $\binom{n}{2k}$ cuts of size $2k$, since every choice of $2k$ edges defines such a cut.

7 Network Reliability

From the cut counting theorem we can deduce a useful fact about the ability of graphs with large min-cuts to resist being separated. In [Col87], the relationship between the min-cut and graph reliability is investigated in great detail; however, this result is of a different flavor:

THEOREM 7.1. *If each edge of a graph with min-cut c is removed with probability $p = n^{-a/c}$, then the probability that the graph becomes disconnected is $O(n^2p^c/(a-2))$.*

Proof. In order for the graph to be disconnected, some cut must have all its edges eliminated. We therefore bound the probability that all the edges in any cut are eliminated. A cut of weight αc has probability at most $p^{\alpha c}$ of having all of its edges eliminated. Let $f(\alpha)$ be the number of cuts of weight αc. Recall that for some constant λ,

$$F(\beta) = \sum_{\alpha<\beta} f(\alpha) < \lambda n^{2\beta}.$$

Let C denote the set of all cuts. The probability of disconnection is at most

$$\sum_{C\in\mathcal{C}} \Pr[\text{all edges of } C \text{ are eliminated}] = \sum_{\alpha} f(\alpha)p^{\alpha c}.$$

Since $p^{\alpha c}$ is decreasing with α, a perturbation argument shows that to maximize the sum, it is desirable to have as much of the mass of f as possible at small values of α. In other words, we want as many small cuts as possible, so $F(\beta)$ should be maximized at every value of β (subject to the constraints). If we remove the restriction that f be discrete and integer valued, then we can take $f(1) = \lambda n^2$ and $F(\beta) = \lambda n^{2\beta}$ for $\beta > 1$. Then the sum is bounded by

$$\lambda n^2p^c + \int_1^\infty (\frac{\partial}{\partial\alpha}\lambda n^{2\alpha})p^{\alpha c}\,d\alpha = \lambda n^2p^c + \lambda n^2p^c/(a-2).$$

Note that for large c, $n^{-a/c} \approx 1 - a\ln n/c$. We have thus shown that if we kill edges with probability $1 - 3\ln n/c$, the graph is disconnected with probability $O(1/n)$. On the other hand, if we kill edges with probability $1 - 1/c$, then the graph is disconnected with constant probability.

Consider the complete graph. Choosing to kill edges with probability $1 - \Theta(\ln n/n)$ corresponds to choosing a random graph from $G_{n,p}$ with $p = \Theta(\ln n/n)$. It is well known [Bol85] that $p = \Theta(\ln n/n)$ is precisely the threshold at which the complete graph becomes connected with high probability. This result is extended by Margulis [Mar75], who shows that every graph has a connectivity threshold; however, this paper appears to be the first to explicitly describe the threshold function.

COROLLARY 7.1. *In a weighted graph, if each edge of weight w fails with probability $n^{-2(1+\delta)w/c}$, then the graph remains connected with probability $1 - O(n^{-\delta}/\delta)$.*

Proof. Apply the above theorem to the multigraph corresponding to the weighted graph.

This gives a method for analyzing the reliability of a given network.

COROLLARY 7.2. *Suppose in an n-vertex network each edge e has failure probability p_e. Assign to edge e a weight $-\log_n p_e$. The network failure probability is $O(n^{2-c})$, where c is the min-cut of the weighted graph.*

Proof. Network edge e fails with probability $p_e = n^{-w_e} = n^{(-cw_e)/c}$, which precisely simulates the weighted failure criteria in the previous corollary.

We use the following lemma to prove the next corollary:

LEMMA 7.1. *If (A, B) is a min-cut of weight c, then the subgraphs induced by the vertex sets A and B each have min-cut at least $c/2$.*

Proof. Full Paper.

COROLLARY 7.3. *If all edges of a graph are killed with probability $(n \log n)^{-4/c}$, then with probability $\Omega((n \log n)^{-4})$, the resulting graph has two connected components, each of which is one side of a min-cut.*

Consider a graph G with min-cut c, and consider the two subgraphs A and B induced by the two sides of the min-cut. By Lemma 7.1, each subgraph has min-cut at least $c/2$. Killing edges with probability $(n \log n)^{-4/c} = (n \log n)^{-2/(c/2)}$ ensures that with constant probability A and B are each connected. Independent of this, with probability $(n \log n)^{-4}$, all edges of the min-cut are killed.

8 Multi-way Cuts

With a small change, the Contraction Algorithm can be used to find a *minimum weight r-way cut*, which partitions the graph into r pieces rather than 2. The analysis need be only slightly changed.

THEOREM 8.1. *Stopping the Contraction Algorithm when r vertices remain yields a particular minimum r-way cut with probability at least*

$$r\binom{n}{r-1}^{-1}\binom{n-1}{r-1}^{-1}.$$

Proof. As before, the key to the analysis is bounding the probability p that a randomly selected graph edge is from a particular minimal r-cut. Suppose we choose $r-1$ vertices uniformly at random, and consider the r-cut defined by taking each of the vertices as one member of the cut and all the other vertices as the last member. Let f be the number of edges cut by this random partition, and m the number of graphs edges. The number of edges we expect to cut is

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

since the quantify in brackets is just the probability that a single edge is cut. Since f can be no less than the value of the minimal r-cut, $E[f]$ must also be no less than the min-cut. We can therefore deduce that the probability that a particular minimum r-cut survives the reduction process until there are r vertices remaining is at least

$$\prod_{u=r+1}^{n} (1 - \frac{r-1}{u})(1 - \frac{r-1}{u-1})$$

$$= \prod_{u=r+1}^{n} (1 - \frac{r-1}{u}) \prod_{u=r+1}^{n} (1 - \frac{r-1}{u-1})$$

$$= r\binom{n}{r-1}^{-1}\binom{n-1}{r-1}^{-1}.$$

This analysis yields a simple $\tilde{O}(mn^{2r-1})$ (sequential time or parallel processor) algorithm for finding a minimal r-way cut. This is a significant improvement on the previously best known sequential result of $O(n^{r^2-r+11/2})$ reported in [GH88]. As before, our algorithm in fact finds *all* the minimal r-way cuts.

COROLLARY 8.1. *The number of minimum r-cuts of a graph is no more than $\frac{1}{r}\binom{n}{r-1}\binom{n-1}{r-1}$, which is $O(n^{2(r-1)})$.*

COROLLARY 8.2. *The number of r-cuts within a factor of k of the optimum is $O(n^{2k(r-1)})$.*

COROLLARY 8.3. *Enumerating all the r-way cuts within any constant factor of the optimum is in $\mathcal{RNC}$ for any constant r.*

9 Open Questions

The min-cut problem has long been known to be $\mathcal{P}$-complete. However, the reduction of [GSS82] showed this to be true only for *directed graphs*. This paper shows that for *undirected* graphs the situation is entirely different, and that much remains to be done in this area. In particular, we have shown that the min-cut problem for undirected graphs is in $\mathcal{RNC}$. This immediately suggests that a similar result may be possible for the s-t min-cut problem on undirected graphs.

Questions are also raised regarding the closely related problem of max-flow. Unlike many min-cut algorithms, the Contraction Algorithm makes no use of max-flow computations. Is this an accident, or is the max-flow problem not parallelizable? Is it possible to use a min-cut algorithm in a non-trivial way as a component of a max-flow algorithm? If not, in what sense is the min-cut problem fundamentally easier than that of max-flow? In particular, what is the complexity ($\mathcal{RNC}$? $\mathcal{P}$-complete?) of finding a max-flow corresponding to a global min-cut?

The Contraction Algorithm uses a very simple rule to find min-cuts in a graph. Further analysis of the Contraction Algorithm may suggest more intelligent schemes for choosing edges. The goal, of course, would be to increase the success probability of the algorithm so as to decrease the number of trials needed. Another significant accomplishment would be to find a deterministic edge contraction rule which places min-cut in $\mathcal{NC}$.

10 Acknowledgement

Many thanks to Serge Plotkin, who has given a great deal of his time and asked numerous helpful questions related to this research. Thanks also to Daphne Koller who suggested numerous clarifications of the exposition.

References

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison Wesley, 1983.

[AS87] Baruch Awerbuch and Y. Shiloach. "New connectivity and msf algorithms for shuffle-exchange network and pram". *IEEE Transactions on Computers*, 36(10):1258–1263, October 1987.

[Bol85] Bela Bollobas. *Random Graphs*. Harcourt Brace Janovich, 1985.

[CDR86] S. Cook, Cynthia Dwork, and R. Reischuk. "Upper and lower bounds for parallel random access machines without simultaneous writes". *SIAM Journal on Computing*, February 1986.

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

[Col88] R. Cole. "Parallel merge-sort". *SIAM Journal of Computing*, 17(4):770–785, August 1988.

[DJP+92] E. Dahlhaus, D. S. Johnson, C. H. Papadimitriou, P. D. Seymour, and M. Yannakakis. "The complexity of multiway cuts". In *Proceedings of the 24th ACM Symposium on Theory of Computing*, pages 241–251. ACM Press, May 1992.

[FF56] L. R. Ford, Jr. and D. R. Fulkerson. Maximal Flow Through a Network. *Canadian Journal of Math.*, 8:399–404, 1956.

[FF62] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, Princeton, NJ, 1962.

[FT86] M. L. Fredman and R. E. Tarjan. "Fibonacci heaps and their uses in improved network optimization algorithms". *Journal of the ACM*, 36:596–615, 1986.

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

[Gaz86] H. Gazit. "An optimal randomized parallel algorithm for finding connected components in a graph". In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*. ACM Press, 1986.

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

[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 Endre Tarjan. "A new approach to the maximum flow problem. *Journal of the ACM*, 35:921–940, 1988.

[Has86] Johann Hastad. "Improved lower bounds for small depth circuits. In *Proceedings of the 18th Annual ACM Symposium on Theory of Computing*, pages 6–20. ACM Press, 1986.

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

[KPN92] David Karger, Michal Parnas, and Noam Nissan. "Fast connected components algorithms for the EREW PRAM". In *Proceedings of the 4th Annual ACM-SIAM Symposium on Parallel Algorithms and Architectures*, pages 562–572, 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.

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

[Mar75] G. A. Margulis. "Probabilistic characteristics of graphs with large connectivity (translated from russian)". *Problems in Information Transmission*, 9:325–332, 1975.

[MVV87] Ketan Mulmuley, Umesh V. Vazirani, and Vijay 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 of Discrete Mathematics*, 5(1):54–66, February 1992.

[NK92] Hiroshi Nagamochi and Tiko Kameda. "An efficient construction of cactus representation for minimum cuts in undirected networks". Manuscript, 1992.

[NSW92] Noam Nissan, Endre Szemeredi, and Avi Wigderson. "Undirected connectivity in $o(log^{1.5}n)$ space". In *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, pages 24–29. IEEE Computer Society Press, October 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.

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