# Distributed Min Cut

Eric Lax, Andreas Santucci
Stanford University

June 3rd, 2015

## 1 Overview

We set out to solve the problem of finding a min cut of a large, un-weighted and un-directed graph $G$, where the number of vertices fits on a single machine but the number of edges does not. In this paper, we will briefly outline previous approaches to this problem, and then explore two new avenues.

The first is a Karger variant, which finds the min cut with very high probability. To do this, we develop a distributed minimum spanning tree algorithm to simulate each iteration of Karger's algorithm. The run-time of finding the MST is $O(\log(n)(\frac{m}{B} + n\log(n)))$. The shuffle cost associated with finding MST is $O(n\log(n) + B\log^2(n))$. In order to find the min cut, this must be run $O(n^2\log(n))$ times.

The second approach is an approximation algorithm based on the intuition that when sampling a graph $G$, the cut that is most likely to be disconnected is the min cut. We can find a $(1+\epsilon)$ approximation of the min cut in $O(\frac{m\log(n)}{\epsilon^2})$, with a shuffle of size $O(\frac{mp\log(n)}{\epsilon^2})$. Using the same algorithm in a CONGEST framework [6], we achieve $O((D + \sqrt{n}\log^*(n))\frac{\log(n)}{\epsilon^2})$ run time with $O(n)$ processors.

Below we will show that these approaches yield the fastest distributed min cut approximation that we are aware of. Finally, we will explore further areas of research, including potential improvements to our algorithm from effective resistance, and applying the same methodology to the S-T min cut problem.

## 2 Literature Review

**Karger** In David Karger's 1993 paper [2], "Global Min-cuts in RNC and other Ramifications of a Simple Mincut algorithm", he developed an $O(mn^2\log(n))$ algorithm which finds the min cut with high probability. The algorithm repeatedly contracts edges at random until only one cut remains. This process is repeated $O(n^2\log(n))$ times. It was later improved upon in the Karger-Stein algorithm [3] to run in $O(n^2\log^3(n))$ time.

**Skeleton Graphs** In his 1999 paper [5], "Minimum Cuts in Near-Linear Time", Karger develops a general methodology computing a skeleton graph and a tree packing on the resulting skeleton graph, then calculating cut sizes determined by trees in the packing. This algorithm finds the min cut in $O(m\log^3(n))$ time.

**Random Sampling Techniques** In his 2000 paper [7], "Random Sampling in Cut, Flow, and Network Design Problems", Karger turns this approach into a $(1+\epsilon)$ approximation which runs in $O(m+n(\frac{\log(n)}{\epsilon})^3)$ time. This approach has been repeatedly adapted and improved upon.

**Most Recent Approximation Algorithm** The most recent and best approximation algorithm we could find to date was published by Danupon Nanongkai and Hsin-Hao Su, [6] in a paper entitled, "Almost-Tight Distributed Minimum Cut Algorithms", which achieves a $(1+\epsilon)$ approximation in $O((\sqrt{n}\log^*(n)+D)\epsilon^{-5}\log^3(n))$ time, where $D$ denotes the diameter of the graph, and $\log^*(n)$ denotes the iterated logarithm. This approach distributes a variant of Karger's skeleton approximation with a CONGEST model, in which each node only has knowledge of its neighbors and $n$ processors are used, each with infinite computational power. They compute run-time simply as the worst case number of rounds of communication. This makes comparison with other algorithms difficult. For example, under this model connected components run in $O(D+\sqrt{n}\log^*(n))$ time. We will show that our algorithm, when carried out in a CONGEST model, actually produces a $(1+\epsilon)$ approximation in lower run time.

**Karger Variants** The 2011 paper [9], "Filtering: A Method for Solving Graph Problems in MapReduce", Lattanzi, Moseley, Suri, and Vassilvitskii is the only paper we were able to find which attempts to find min cuts for large graphs which don't fit on a single machine.

Their general approach is to arbitrarily weight edges in the range $(0,1)$, find a threshold $t\in(0,1)$ such that contracting all edges with weight less than $t$ results in the largest contracted graph capable of fitting on a single machine. They then apply Karger-Stein's algorithm. A downside to this approach is that a connected components analysis is simultaneously run for many values of $t$ to find the optimal value of $t$, leading both to extremely high communication cost and machine usage. The paper does not actually report the actual run time for the algorithm. For a point of comparison, we have tried to compute it using their methodology. In this approach we were generous in order to give us a higher standard of comparison.[1]

Each connected components analysis runs in $O(m\alpha(m,n))$, and its executed a total of $n^y$ times in parallel, where a machine can store $O(n^{1+y})$ information. Karger-Stein is then run locally on the remaining graph $G'$, with $\hat{n}$ nodes, costing $O(\hat{n}\log^3(\hat{n}))$ time. This process is repeated $n^{1-\frac{y}{2}}$ times in parallel, using a total

[1]For example, the contraction algorithm run time was unclear, so we omitted it entirely.

of $O(n^{1-\frac{y}{2}+k})$ machines, where $m = \frac{k(n)(n-1)}{2}$. Using this many machines, this whole process is repeated $\frac{1}{y^2}$ times, leading to a total run time of $O(\frac{m\alpha(m,n)}{y^2} + \frac{\hat{n}\log^3(\hat{n})}{y^2})$.

# 3 Distributing Karger through MST

**Relating MST and Karger** Due to the cut property of MST's, the lowest weight edge leaving each connected component must be contained in the minimum spanning tree. Prim's algorithm [1] starts with each node as its own connected component, the lowest weight outgoing edge is found, and used to extend the connected component. Each iteration across all connected components reduces the number of connected components by at least 1/2, therefore there are at most $\log(n)$ rounds required.

## 3.1 Algorithm

**Algorithm 1:** Distributed Karger

```
Partition:
    Place all edges for each node on a single machine
Set min-cut to ∞
for i = 1 to n^2 log(n) do
    Assign arbitrary weights to each edge
    Perform Distributed MST
    Remove maximum weighted edge (We now have a cut)
    Perform Connected Components Analysis to determine how nodes are
    partitioned (return a hash set for each partition, containing the nodes
    on each side)
    Map:
        Across edges, check for an edge between two partitions using the
        two hash-sets
    Reduce:
        Return the number of edges between two partitions
    if smallest cut thus far then
        update min cut
    end
end
Return min-cut
```

In order to analyze this algorithm, we must analyze how we distribute MST. We will go over two approaches.

### 3.1.1 Distributed MST Approach - Low Shuffle Size

**General Approach** Note that in our Karger algorithm, edges are already partitioned such that all edges from a vertex are stored on a single machine.

To start, consider each node as its own connected component. We perform a map to find the lowest weight edge from each node, and then a reduce step to find the lowest weight edge leaving each connected component. Results are sent back to the driver. The driver finds the lowest weight edge leaving each *connected component*, and broadcasts this list to the workers [2], at which point each machine locally combines connected components. In subsequent iterations, each machine finds the lowest weight edge from each node leaving its connected component, and send these results back to a reducer. This is process is repeated until there are no edges remaining which leave connected components.[3]

**Algorithm 2:** Distributed MST (low shuffle size)

```
Set each node as a connected component
while Edges exist leaving connected components do
    Map:
        For each node, find the lowest weight edge leaving the connected
        component which contains that node
    Reduce:
        Find the lowest weight edge leaving each connected component
    Broadcast:
        Lowest weight edge leaving each connected component to all
        machines
    On each machine, merge connected components together, and in the
    process create a hash set for each connected component, which stores
    nodes belonging to that component
end
Return MST
```

## Algorithm Analysis

**Finding Lowest Weight Edge Leaving Each Node** Finding the lowest weight edge from each node requires checking each edge, which costs $O(\frac{m}{B})$. When iterating over edges in the form of $(i, j)$, we check if node $j$ is in the same connected component as $i$ in constant time using the hash set associated with the connected component for $i$. This process is explained in detail below. If $j$ is not contained in the set, we know the edge leaves the connected component. Since there are $\log(n)$ iterations, the total cost of this operation across iterations is $O(\frac{m \log(n)}{B})$.

**Finding Lowest Weight Edge Leaving Each Component** The reduce step requires an all to one communication of $O(n)$, since each node sends back

[2] It is worth noting that we can reduce the communication cost and run-time by caching the lowest weight edge for each node on the driver in the event that it leaves the newly formed connected component. In the next iteration, we don't have to check for that nodes lowest weight edge, which also means less communication.

[3] This algorithm emerged from a discussion with Swaroop Ramaswamy who discussed with us how to capture benefits from both of our approaches to MST. Some of the underlying ideas in this model come from his approach.

the lowest weight edge. To actually find the lowest weight edge leaving each connected component is upper bounded by $O(n)$. In total, the shuffle size is $O(n \log(n))$ across iterations.

**Broadcasting Connected Components** In each step, we have a list of edges which we wish to add to our connected components to combine them. There is one edge per connected component. We will broadcast this to machines such that connected components may be updated locally.

Since the number of connected components is reduced by at least $1/2$ on each iteration, the first broadcast requires shuffle size $n$, the second requires $n/2$, the third requires $n/4$. In general, on each iteration, the shuffle cost of the broadcast is at most $2^{-i}n$, where $i$ denotes the iteration number.

Notice that the size of the broadcast is $O(n)$. Since it is broadcast to $B$ machines, the shuffle cost is $O(B \log(n))$ using bit-torrent broadcasting.

**Merging Connected Components** In order to merge connected components together efficiently and keep track of which nodes belong to each component, we require the use of several data structures. We first create a hash map where the keys consist of the unique node id's, and the values consist of pointers to hash sets. When the lowest weight edge leaving each connected component is broadcast, for each edge $(i, j)$, we use our hash map to look up in constant time both the hash set containing $i$, and the hash set containing $j$. We then merge these two hash sets, by adding all the elements from the smaller hash set to the larger hash set. We update the pointers accordingly for the nodes belonging to the smaller hash set. The smaller hash set is then discarded and no longer used.

This process is deterministic. So, the same hash-sets and maps will appear on each machine and the driver.

It's worth noting that we will only move $O(n)$ elements in each iteration. Since adding an element to a hash set is $O(1)$, this process runs in $O(n)$ time. Similarly, we are updating maximally $O(n)$ pointers, each pointer can also be update in $O(1)$ time. Therefore, merging connected components costs $O(n)$. Since there are at most $\log(n)$ iterations required in order to form a minimum spanning tree, the cost of merging connected components is total $O(n \log(n))$.

**Run Time Analysis - Distributed MST (low shuffle cost)** Arbitrary weighting costs $O(m/B)$. Finding the lowest weight edge from each node is $O(\frac{m \log(n)}{B})$ across iterations. Finding the lowest weight edge from each connected component is $O(n)$. Merging connected components is $O(n \log(n))$. So, total run time is given by $O(\frac{m \log(n)}{B} + n \log(n))$.

**Shuffle Size - Distributed MST (low shuffle cost)** Finding the lowest weight edge leaving each connected component requires shuffle size $O(n \log(n))$. Broadcasting the connected components costs $O(B \log(n))$. The total shuffle size is given by $O(n \log(n) + B \log(n))$.

### 3.1.2 Distributed MST Approach - Low Number Map Reduces

We first find the lowest weight $y$ edges leaving each vertex. Note that in our Karger algorithm, edges are already partitioned such that all edges from a vertex are stored on a single machine. Using a max-heap data structure of size limited to $y$, we iterate through all $\deg(v_i)$ edges leaving node $v_i$, and for each, check to see if its smaller than the max element in the heap $O(1)$ time. If it is, we add it to the heap which costs $O(\log(y))$ time, and remove the max. If it's bigger, we continue to the next item. We then sort the heap into an array in $O(y \log(y))$ time. The total run time across all vertices is given by

$$\frac{\sum_i \deg(v_i) \log(y)}{B} = \frac{2m \log(y)}{B}$$

Therefore the run time is $O(\frac{m \log(y)}{B})$.

We then perform an all-to-one communication and send the results back to the driver, with shuffle size $O(ny)$. The driver then checks for the lowest weight edge leaving each connected component. In order to do this, the driver simply checks the lowest weight edge leaving each node, meaning the driver will check $n$ nodes. Since the $y$ edges are already sorted, each node can be checked in constant time, so this takes a total $O(n)$ time each iteration.

**Caveat** Note, however, that we must also deal with the case where all $y$ edges leaving a single node are exhausted in the search for an MST, but we have more than one connected component remaining. It is then possible that the minimum weight edge leaving that connected component does not reside within the driver. Therefore, we no longer attempt to augment this connected component with any of the remaining outgoing edges from other nodes within the connected component. If this becomes true of all our connected components, we can no longer add edges to reduce the number of components.

Since each connected component is at least size $y$, there are at most $n/y$ connected components. The size of the broadcast is $n$, since all we need to know is which connected component contains each node. We broadcast the list of connected components back to all machines.

We repeat the process above, considering the next lowest weight $y$ edges from each node leaving each connected component. The results are sent back to the driver, with a shuffle size of $ny$. Notice that the smallest connected component we could have before running out of outgoing edges on this iteration would be of size $y^2$.

In the worst case, the number of iterations performed for this process is given by $y^x(n) \geq n$, such we that have removed at least $n-1$ connected components. We then have

$$x = \frac{\log(n)}{\log(y))}$$

Although this is a lower total number of iterations, the shuffle size per iteration is high enough to make our previous algorithm more efficient in the worst

case. However, since in applying Karger's Variant the edge weights are assigned randomly, the worst case is unlikely to occur[4]. In future research, we plan to delve more deeply into finding the expected run time given random assignment of edge weights. If latency is a factor, this algorithm exhibit increasing returns.

**Algorithm 3:** Distributed MST (low map-reduces)

```
Set each node as a connected component
while Edges exist leaving connected components do
    Map:
        Across each node: search for the y lowest weight edges leaving
        that node's connected component
    Send y lowest weight results back to driver
    while True do
        if One connected component remains Or At least one node in
        each connected component has exhausted all y edges then
            break
        end
        for Each Connected Component such that none of the nodes have
        exhausted all y outgoing edges do
            Take minimum edge leaving component and use this to merge
            two components
        end
    end
    Broadcast:
        Connected Components to all machines
end
Return MST
```

**Run Time Analysis (per iteration, low map-reduces)** Computing the minimum spanning tree first requires finding the lowest weight $y$ edges leaving each vertex. To do this, we use a max-heap data structure, taking $O(\frac{m \log(y)}{B})$ time. Computing the MST on the driver takes $O(ny)$ time. The total run time is given by $O(\frac{m \log(y)}{B} + ny)$.

**Shuffle Size (per iteration, low map-reduces)** The algorithm only uses one all to one communication of size $O(ny)$ and one bit-torrent broadcast costing $O(B \log(n))$. For each node, the lowest weight $y$ edges are sent back, so on each iteration the shuffle cost is given by $O(ny)$.

## 3.2 Algorithm Analysis

**Run Time Analysis - Distributed Karger** Notice that in the algorithm, after weighting edges randomly and finding an MST, the largest weight edge

[4]The worst case is that in each iteration, we only reduce the number of connected components by a factor of $\frac{1}{y}$ by exhausting all edges leaving a specific node.

is removed and connected components analysis is performed to determine how nodes are partitioned across the cut. The connected components analysis is $O(n)$ since there are $O(n)$ edges, and can be performed using a simple Prim's algorithm. While performing connected components analysis, we accumulate a hash-set of nodes stored in each side of the partition in a similar fashion to the method described for the low-run time MST. We then map across edges, checking to see if the edge spans the cut. Using the hash-set, each edge requires a constant time check, so the total cost when computed in parallel is given by $O(m/B)$.

Since computing MST is the bottleneck, and since we need $n^2 \log(n)$ iterations to run Karger, the total run-time is given by
$O((n^2 \log(n))(\frac{m \log(n)}{B} + n \log(n)))$. Note that for $B \geq \log(n)$, this is faster than Karger's algorithm.

**Shuffle Size - Distributed Karger** We have an all to all communication required by partitioning edges. The cost of this operation is $O(m)$. Notice that the bottleneck is MST. The total shuffle size is given by $O(n \log(n) + B \log^2(n))$.

**Limitations in Scaling** One limitation in scaling is that machines can only hold $O(n)$ information.

**How strongly does Algorithm scale** Although this algorithm beats Karger under certain conditions, it is far from optimal. Even with a large number of machines, we still have a $n \log(n)$ term incurred from MST, and we require $n^2 \log(n)$ iterations.

# 4 Distributed Min Cut Approximation Algorithm

Let $c$ denote the size of the true min cut of $G$. First, we are going to sample each edge with probability $p$. Let $G'$ be the sub-graph from sampling edges in $G$ with probability $p$. We are going to choose $p$ such that $G'$ is disconnected in expectation.

Let us examine a simple fully connected graph with nodes $A, B, C, D$. In $G'$, the probability that nodes $A$ and $B$ are disconnected from $C$ and $D$ is a function of the cut-size $AB$, defined as $\xi$. More precisely, the probability that they are disconnected in $G'$ equals $(1-p)^{\xi}$, since no edge that crosses that cut can be chosen to be in our sub-graph. The larger the cut, the less likely it is to be disconnected in $G'$. By definition, the min-cut is most likely to be disconnected. If we generate enough random sub-graphs $G'$, the cut which is most often disconnected will have size less than $(1+\epsilon) \cdot c$ with high probability.

Let us define an $\alpha$ cut as a cut whose size is less than or equal to $\alpha c$. From Karger[2], the number of $\alpha$-cuts is less than $n^{2\alpha}$. Let $f(\alpha)$ be the number of cuts of size $\alpha c$.

Let $Z(\alpha) = \#$ times a cut of size $\alpha c$ is disconnected, $y = \#$ of iterations, and let $x$ denote the probability the min-cut is disconnected, which is given by $(1-p)^c$. Let $\mu = E[Z(\alpha)] = y(1-p)^{\alpha c} = yx^\alpha$.

**Theorem 1.** *It's highly improbable that cuts of size $\alpha c$, for $\alpha \geq 3$, will be disconnected most often if we have more than $\frac{4}{\sqrt{2}} \cdot 2 \cdot 3 \cdot (1-\epsilon)^2 \ln(n)$ iterations.*

*Proof.* Recognize that $Z(1)$ is the number of times the min-cut is disconnected. Then, using a Chernoff bound,

$$\begin{aligned}\Pr\Big(Z(1) < (1-\epsilon)\mu\Big) &\leq e^{(-\mu\epsilon^2)/2} \\ &\leq e^{(-xy\epsilon^2)/2}\end{aligned}$$

If $y \geq \frac{1}{x}\ln(n) \cdot \frac{1}{2\epsilon^2}$, then $\Pr\Big(Z(1) < (1-\epsilon)\mu\Big) \leq e^{-\ln(n)} = n^{-1}$. So we may say this event is highly unlikely.

Let $\Pr(\alpha) = \Pr\Big(Z(\alpha) > Z(1)\Big)$. From Karger[2] the probability that any cut is disconnected more often than $Z(1)$ is simply given by the sum of $\sum_\alpha \Pr(\alpha) f(\alpha)$. Define $F(x) = \sum_{\alpha \leq x} f(x)$, where $F(x) \leq n^{2x}$. Taking the worst case scenario where $F(x) = n^{2x}, \forall x$, we can further relax $F(x)$ to be a real-valued function rather than restricting it to the space of integers, in which case $f(a) = dF/d\alpha$, we can then take the integral

$$\int_1^\infty \Pr(\alpha) \frac{dF}{d\alpha} d\alpha$$

Since it is highly unlikely that $Z(1)$ is less than $(1-\epsilon)xy$, then $\Pr(\alpha) \leq \Pr\Big(Z(\alpha) > (1-\epsilon)xy\Big)$. We may upper bound the probability that a cut is more often disconnected than the min-cut. For $\alpha \geq 3$:

$$\begin{aligned}\Pr(Z(\alpha) > (1-\epsilon)yx) &= \Pr(Z(\alpha) - \mu > (1-\epsilon)yx - \mu) \\ &\leq \Pr(|Z(\alpha) - \mu| > (1-\epsilon)yx - yx^\alpha) \\ &= \Pr\Big(|Z(\alpha) - \mu| > \mu((1-\epsilon)x^{-(\alpha-1)} - 1)\Big) \\ &\leq \exp\Big[-yx^\alpha\Big((1-\epsilon)yx^{-(\alpha-1)} - 1\Big)^2/3\Big] \\ &\approx \exp\Big[-yx^\alpha\Big((1-\epsilon)x^{-(\alpha-1)}\Big)^2/3\Big] \\ &= \exp\Big[-y(1-\epsilon)^2 x^{-(\alpha-2)}/3\Big]\end{aligned}$$

Now, let $y = \frac{4}{\sqrt{2}} \cdot 2 \cdot 3 \cdot (1-\epsilon)^2 \ln(n)$, then

$$
\begin{aligned}
&\int_3^\infty n^{2\alpha} \cdot \ln(n^2) \Pr(\alpha) d\alpha \\
=&\int_3^\infty n^{2\alpha} \ln(n^2) \exp\left[ - y x^{-\alpha+2} \cdot (1-\epsilon)^2/3 \right] d\alpha \\
=&\int_3^\infty n^{2\alpha} \ln(n^2) \exp\left[ - \ln n \left( \frac{1}{(1-\epsilon)^2} \right) \cdot \frac{4}{\sqrt{2}} \cdot 6x^{-\alpha+2} \frac{(1-\epsilon)^2}{3} \right] d\alpha \\
=&\int_3^\infty n^{2\alpha} \ln(n^2) n^{-(\frac{4}{\sqrt{2}} \cdot 2x^{-\alpha+2})} d\alpha \\
=&\int_3^\infty n^{2(\alpha - \frac{4}{\sqrt{2}} \cdot \frac{1}{x^{\alpha-2}})} \ln(n) d\alpha \qquad \text{We may non-problematically choose } x < \frac{1}{\sqrt{2}} \text{(See section on choosing x)} \\
\leq&\int_3^\infty n^{-2\alpha} \ln(n^2) d\alpha \\
\leq&\frac{1}{n^6}
\end{aligned}
$$

□

So clearly for $\alpha \geq 3$, we can disregard the possibility that a cut of size $\alpha c$ is disconnected the most.

**Theorem 2.** *For $\alpha < 3$, the probability that there exists a cut of size $\alpha$ such that $Z(\alpha) > (1+\epsilon)E[Z(\alpha)]$ is minimal.*

*Proof.* Let $y = \frac{6\ln(n)}{x^3\epsilon^2}$. Define $\Pr_1(\alpha) = \Pr(Z(\alpha) > (1+\epsilon)E[Z(\alpha)])$

Using the logic from above, the probability the theorem holds for all $\alpha < 3$ is given by $\int_1^3 \Pr_1(\alpha) \frac{dF}{d\alpha} d\alpha$

$$\Pr_1(\alpha) = \Pr(z > \mu(1+\epsilon)) \leq \exp\left[ - \mu \frac{\epsilon^2}{3} \right] = \exp\left[ - y x^\alpha \cdot \frac{\epsilon^2}{3} \right]$$

where the inequality follows from a Chernoff Bound. Now taking the integral, and plugging in for $y$ and $\frac{dF}{d\alpha}$, we get a value less than $\frac{1}{n}$.

$$\begin{aligned}
\int_1^3 \Pr_1(\alpha)\frac{dF}{d\alpha}d\alpha &= \int_1^3 \exp\left[-yx^\alpha \cdot \frac{\epsilon^2}{3}\right]\frac{dF}{d\alpha}d\alpha \\
&= \int_1^3 \exp\left[-\frac{3\cdot 6\ln(n)}{x^3\epsilon^2}\cdot x^\alpha \cdot \frac{\epsilon^2}{3}\right]\ln(n^2)d\alpha \\
&\leq \int_1^3 (\exp[\ln(n)])^{-6}\ln(n^2)d\alpha \\
&\leq \int_1^3 n^{-2\alpha}\ln(n^2)d\alpha \\
&\leq \int_1^\infty n^{-2\alpha}\ln(n^2)d\alpha \\
&= \frac{1}{n}
\end{aligned}$$

where the last inequality stems from the fact that $n^{-2\alpha}\ln(n^2) > 0$.

□

**Theorem 3.** *With high probability, the cut which is disconnected most frequently will have a cut size within* $(1+\epsilon)$ *of the true min-cut* $c$*. Let* $y = \frac{6\ln(n)}{x^3(\epsilon/2)^2}$*, and let* $x \leq \frac{1}{e}$*.*

*Proof.* By theorem 1 we can ignore the case when $\alpha \geq 3$. For $\alpha < 3$, we know from theorem 2 that it's highly probable that $Z(\alpha) < (1+\frac{\epsilon}{2})yx^\alpha$ (we divide $\epsilon$ by 2 based on our choice for $y$). Similarly, we know that $Z(1) > (1-\frac{\epsilon}{2})yx$ with high probability. Further, if $Z(\alpha) > Z(1)$, then it's highly probable that $yx^\alpha > (1-\epsilon)yx$, so solving for $\alpha$ we get $\alpha = 1 + \frac{\ln(1-\epsilon)}{\ln(x)}$. Substituting in for $x = \frac{1}{e}$ we see that $\alpha = 1 - \ln(1-\epsilon) \approx 1+\epsilon$. Note that for lower values of $x$ the approximation is closer to 1. □

# 5 Distributed Min Cut Approximation Algorithm

## 5.1 Binary Search

Let $\theta$ denote the proportion of times that the number of connected components is in $[2, \log(m)]$. We want to perform binary search to find $p$ such that $\theta > s$ for some $s \in \mathbb{R}$ such that $s \in (0,1)$. When we create sub-graphs, we will throw out in expectation less than $1/s$ which do not meet our criterion. This means the running time of our algorithm will simply increase by a constant factor. So, when counting the cuts between connected components, i.e. disconnected cuts, a full enumeration is $2^{\log(m)} = O(m)$.

We will do this by performing a binary search for $p$, starting with $p = 1/\delta$, where $\delta$ denotes the min degree of the graph. Sample edges with probability $p_1$

to yield graph $G'_{1,1}$, run connected components, count number of connected components. For $p_i$, let us create $\eta$ sub-graphs, denoted by $G'_{ij}$ for $j = 1, 2, \ldots, \eta$. For each sub-graph $G'_{ij}$, define

$$\hat{\theta}_i = \frac{1}{\eta} \sum_{j=\eta}^{\eta} \mathbf{1}_i \{\#cc \in [2, \log(m)]$$

and,

$$\mathbf{1}_{\uparrow i} = \{\#cc = 1\}$$
$$\mathbf{1}_{\downarrow i} = \{\#cc > \log(m)\}$$

and correspondingly

$$\hat{\theta}_{\uparrow i} = \frac{1}{\eta} \sum_{j=\eta}^{\eta} \mathbf{1}_{\uparrow i}$$
$$\hat{\theta}_{\downarrow i} = \frac{1}{\eta} \sum_{j=\eta}^{\eta} \mathbf{1}_{\downarrow i}$$

The binary search will sample $\eta$ times to come up with an estimate for each of $\hat{\theta}_i$, $\hat{\theta}_{\uparrow i}$, and $\hat{\theta}_{\downarrow i}$. If we are confident that $\theta_i > s$, we may stop our binary search. Else, we must decide whether to increase or decrease $p_i$. If $\hat{\theta}_{\uparrow i} > \hat{\theta}_{\downarrow i}$, we decrease $p_i$, and similarly if $\hat{\theta}_{\downarrow i} > \hat{\theta}_{\uparrow i}$ we increase $p_i$.

**Theorem 4.** *We claim we only need a constant number of iterations to determine whether to increase or decrease each $p_i$, or stop the search entirely.*

*Proof.* Notice $\sum_j \mathbf{1}_j$ is a binomial random variable, and the same is true for the random variables described by $\{\uparrow j\}$ and $\{\downarrow j\}$, where for $j = 1, 2, \ldots, \eta$ we have independent and identically distributed trials. This allows us to take advantage of a Chernoff bound.

$$\Pr\left(\sum_j \mathbf{1}_j < \eta\theta_i(1-\epsilon)\right) < \exp\left[\frac{-\eta\theta_i\epsilon^2}{2}\right]$$
$$\Pr\left(\hat{\theta}_i < \theta_i(1-\epsilon)\right) < \exp\left[\frac{-\eta\theta_i\epsilon^2}{2}\right]$$

Similarly,

$$\Pr\left(\sum_j \mathbf{1}_{\uparrow j} < \eta\theta_{\uparrow i}(1-\epsilon)\right) < \exp\left[\frac{-\eta\theta_{\uparrow i}\epsilon^2}{2}\right]$$
$$\Pr\left(\hat{\theta}_{\uparrow i} < \theta_{\uparrow i}(1-\epsilon)\right) < \exp\left[\frac{-\eta\theta_{\uparrow i}\epsilon^2}{2}\right]$$

and further

$$\Pr\left(\sum_j \mathbf{1}_{\downarrow j} < \eta\theta_{\downarrow i}(1-\epsilon)\right) < \exp\left[\frac{-\eta\theta_{\downarrow i}\epsilon^2}{2}\right]$$

$$\Pr\left(\hat{\theta}_{\downarrow i} < \theta_{\downarrow i}(1-\epsilon)\right) < \exp\left[\frac{-\eta\theta_{\downarrow i}\epsilon^2}{2}\right]$$

Notice that we have exponential convergence for each of our random variables, provided the true parameters are *not* vanishingly small. By definition, $\hat{\theta}_i + \hat{\theta}_{\uparrow i} + \hat{\theta}_{\downarrow i} = 1$. Therefore, at most two parameters are vanishingly small. Consider three cases.

**Case 1:** All of $\theta_i, \theta_{\uparrow i}$, and $\theta_{\downarrow i}$ are not vanishingly small. We have exponential convergence for all estimators.

**Case 2:** One of three parameters is vanishingly small. Without loss of generality, let $\theta_i \approx 0$. We then have $\hat{\theta}_{\uparrow i} \to \theta_{\uparrow i}$ and $\hat{\theta}_{\downarrow i} \to \theta_{\downarrow i}$. Then $\hat{\theta}_i = 1 - \hat{\theta}_{\uparrow i} - \hat{\theta}_{\downarrow i} \to 1 - \theta_{\uparrow i} - \theta_{\downarrow i} = \theta_i$. Again, we have exponential convergence for all estimators.

**Case 3:** Two of three parameters are vanishingly small. Without loss of generality, let these be $\theta_{\uparrow i}$ and $\theta_i$. So, $\theta_{\downarrow i} \approx 1$. Since $\hat{\theta}_{\downarrow i}$ converges exponentially fast, we have $\hat{\theta}_{\downarrow i} \approx 1$. Since $\hat{\theta}_{\uparrow i} + \hat{\theta}_i = 1 - \hat{\theta}_{\downarrow i} \approx 0$, we know that $\hat{\theta}_{\uparrow i} \approx 0$ and that $\hat{\theta}_i \approx 0$. Since these values are non-negative, we can say with certainty that $\theta_{\downarrow i}$ is greater than either of the other two parameters.

In all cases, we are able to determine the appropriate action to take in our binary search. Since we have exponential convergence in each case, we only need $O(\log(\eta))$ iterations before moving onto the next step of binary search. □

**Choosing x** Note in our proof we put an upper bound on $x$, however, this is not problematic for two reasons. An upper bound on $x$ implies a lower bound on $p$. Since low $p$ correlates to a higher number of connected components on the resulting sub-graph, it does not interfere with our binary search so long as $E[CC_p] > 1$. Given our bound for $x$, we know that the probability the min-cut is disconnected is $\frac{1}{\epsilon}$, therefore it's impossible for $E[CC_p] = 1$.

## 5.2 Algorithm

**Algorithm 4:** Distributed Min Cut

```
Binary Search for optimal p
In Parallel:
    Sample each edge with probability p
    Compute connected components on sub-graph
    if Number connected components ∈ [2, O(log(m))] then
        Return connected components
    end
    else
        Discard
    end
end
Count number of times each cut is disconnected
Maximally occurring cut is highly likely to be a 1 + ϵ approximation of
min cut
```

## 5.3 Run Time and Shuffle Size Analysis

The run time varies drastically depending on the methodology used to sample connected components. The binary search requires sampling $O(\log(n))$ sub-graphs. Let $CC$ denote the time it takes to run connected components analysis. We then have to run $O(CC\log(n))$ connected components analyses to determine optimal $p$. Once we have $p$, we must sample another $\log(n)/\epsilon^2$ sub-graphs, and perform yet another $O(CC\log(n)/\epsilon^2)$ connected components analysis. Enumerating the number of disconnected cuts takes $O(m)$ with a carefully chosen $p$. Thus, finding the most often disconnected cut from these lists is upper bounded by $O(\frac{CC\log(n)}{B\epsilon^2})$, since the process is embarrassingly parallel provided $B < \log(n)/\epsilon^2$, since the sampling processes are independent of each other.

### 5.3.1 Standard Approach

If $G'$ fits on one machine, we may perform connected components locally in $O(m)$ time. Since each sub-graph will have $O(mp)$ edges, the total run time is $O(\frac{mp\log(n)}{\epsilon^2})$. Since edges of $G'$ must be grouped together for each $G'$, the shuffle cost is given by $O(\frac{mp\log(n)}{\epsilon^2})$.

### 5.3.2 When Sub-Graph is Large

If $G'$ is too large to fit on a single machine, we may use our distributed MST algorithm to find connected components. In the low-run time MST described above, the cost for finding an MST is $O(\frac{m\log(n)}{B} + B\log(n))$ with shuffle size $O(n\log(n)+Bn)$. The cost of the whole analysis is $O(\frac{\log(n)}{\epsilon^2}(\frac{m\log(n)}{B}+B\log(n)))$

The shuffle for each iteration costs $O(n\log(n) + Bn)$ so total shuffle for the algorithm is $O(\frac{\log(n)}{\epsilon^2}(n\log(n) + Bn))$.

#### 5.3.3 High Machines

In Karger's paper [4], "Fast Connected Components Algorithms for the EREW PRAM", he develops a fast connected components analysis which runs in $O(\log(n)\log(\log(n)))$ time, but which requires $(m+n)/\log(n)$ EREW processors (exclusive read, exclusive write). We recognize that this may not be feasible for larger graphs, however, if those resources are available, we perform a binary search for $p$ such that $E[\#\text{ connected components}] = \log(\log^2(n))$. Therefore enumeration of all disconnected cuts only takes $O(\log^2(n))$. Thus, the whole analysis is $O(\frac{\log^2(n)\log(\log(n))}{\epsilon^2})$, as the bottleneck is sampling and computing sub-graphs.

### 5.4 An Exact Algorithm

It's worth noting that by setting $\epsilon \leq \frac{1}{c}$, where $c$ denotes the size of the min-cut, the approximation algorithm becomes exact. This is because the approximation becomes $c(1+\epsilon) \leq c(1+\frac{1}{c}) = c+1$.

## 6 Comparing to Other Approaches

The two comparable papers are written by Lattanzi et.al [9] and Danupon Nanongkai and Hsin-Hao Su [6]. In the Lattanzi paper, the number of machines they used ends up being $O(n)$. If we use that many machines our approach becomes faster. Our approach also has the advantage of having lower shuffle cost.

The more interesting comparison is to Nanongkai's paper. Generalizing our model to the CONGEST model where running connected components costs $O(d + \sqrt{n}\log^*(n))$, we can adjust our binary search algorithm to find $p$ such that $E[\text{number connected components}] = \log(\sqrt{n})$ and run our algorithm for $\frac{\log(n)}{\epsilon^2}$ iterations. This leads to a $O((D + \sqrt{n}\log^*(n))\frac{\log(n)}{\epsilon^2})$ run time, in which case our algorithm is faster by a factor of $O(\frac{\log^2(n)}{\epsilon^3})$.

## 7 Conclusion

In conclusion, with high number of machines, our algorithm becomes exceedingly fast. Even in the absence of high machines, our process works regardless of the size of the sub-graphs, thus forming a new approach to approximating min cut. It's worth noting, that although we've dealt with graphs in an un-weighted context, we should be able to handle weights by sampling not with constant $p$ but with $p_i$ inversely proportional to each edges weight.

# 8 Areas Future Research

If we find an efficient methodology from going from a list of connected components to the most often disconnected cut without requiring full enumeration, it is possible to remove the binary search from this process, which frees up more flexibility on $p$, which allows us to sample graphs such that they are small enough to fit locally on a single machine.

Moreover, another interesting avenue of research is to sample edges with respect to effective resistances. Thus far, we know that sampling with respect to effective resistances makes the min-cut comparatively more likely to be disconnected, which should reduce the number of iterations required. However, determining a tight-bound will be left for future research.

Finally, the same approach generalizes to ST Min-Cut, if we can find a way to guarantee that $S$ and $T$ are separated a constant proportion of the time when sampling sub-graphs. The general approach to this methodology would be to perform a binary search for $p$ such that $S$ and $T$ are separated $1/2$ the time, and then only consider sub-graphs when they are disconnected.

# References


[1] Prim, R. C. *Shortest Connection Networks and some generalizations* 1957 3

[2] David R. Karger, *Global Min-cuts in RNC and other Ramifications of a Simple Mincut algorithm*, SODA, Philadelphia, 1993. 2, 4, 4

[3] David R. Karger, Clifford Stein, *A new approach to the minimum cut problem*, 1996. 2

[4] David R. Karger, Noam Nisan, Michal Parnas *Fast Connected Components Algorithms for the EREW PRAM* 1997. 5.3.3

[5] David R. Karger *Minimum Cuts in Near-Linear Time* 1998 2

[6] Danupon Nanongkai, Hsin-Hao Su *Almost-Tight Distributed Minimum Cut Algorithms* 2014. 1, 2, 6

[7] David R. Karger *Random Sampling in Cut, Flow, and Network Design Problems* 2000. 2

[8] Fan Chung, Paul Horn, Linyuan Lu *The giant in a random subgraph of a given graph* 2009

[9] Lattanzi, Mosely, Suri, Vassilvitskii *Filtering: A Method for Solving Graph Problems in MapReduce* 2011. 2, 6