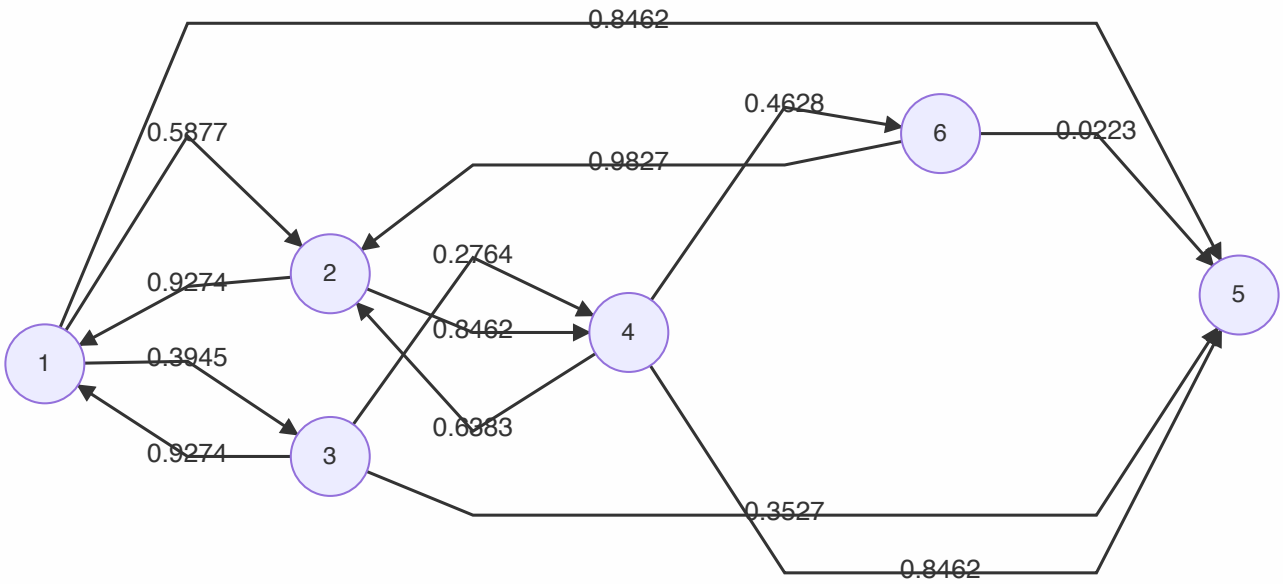
William Clift
Algorithms
14 May 2020

Karger-Stein Minimum Cut Graph Algorithm

Concepts

In this assignemnt, we explored algorithms that were developed to solve the "minimum" cut problem for graphs. This is done using a random algorithm, Karger's Algorithm, which randomly selects edges and contracts the graph until only two nodes remain. At this point, we sum up with edges between the two nodes and that gives us a cut which is relatively small. This algorithm is very fast, and thus we have a tradeoff of acccuracy. To counter this, we run the algorithm multiple times in `mincut()`. After a certain number of runs, the probability of smallest of all of these iterations actually being the minimum cut grows. To improve on this more, we take into account the idea that the fault of Karger's algorithm is if we randomly select one of the edges contained in the minimum cut before we contract to 2 nodes. It is less likely that we will contract one of these edges when the graph is very big. So, we use the improved Karger-Stein algorithm. This contracts the graph down to t nodes in many iterations and then runs mincut on the much smaller graph. With this, we can get the same accuracy with a much quicker algorithm.



The graphs that we analyze in this exercise may look something like this. Notice that each edge has a weight. In this example, the direction of the edge does not matter. The minimum cut is the smallest sum of weighted edges that can split the graph into two completely separate graphs.

Implementation

Our first task was to implement the code for Karger and add the functionality for weighted edges. The `karger()` algorithm runs only one iteration. It first seaches for separated subsets and contracts an edge that connects them, repeating until only two nodes remain. This final edge between the nodes is the sum of the weights that add to a candidate for minimum cut. The `mincut()` runs this multiple iterations and returns the lowest one. I made the choice to run it $2 \cdot V^2$, where V is the number of nodes, in order to get

98 percent confidence. In our `fastKarger()`, referring to the Karger-Stein algorithm, we have run `contract()` down to t nodes and complete `mincut()` on the small graph. This is done in `fastMinCut()` by running multiple iterations and taking the smallest one to be the mincut.

Parallelization

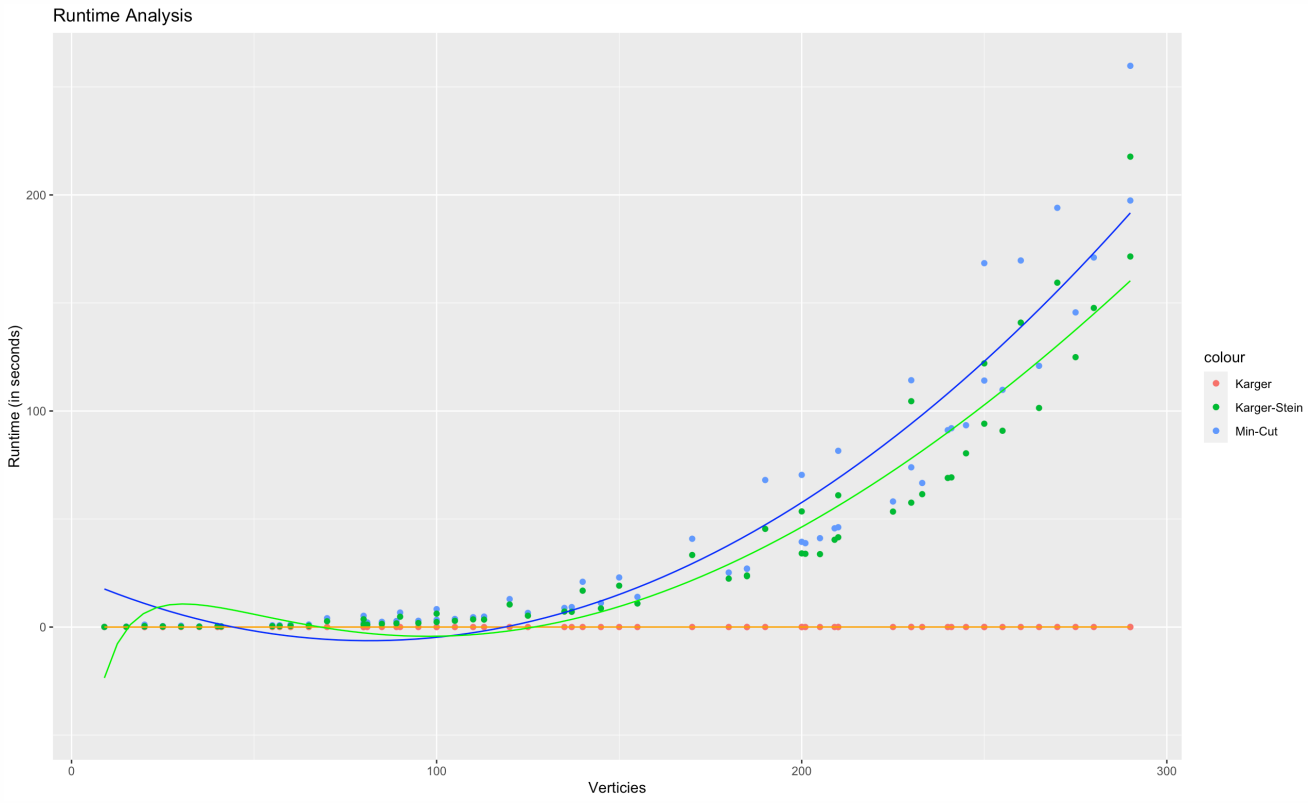
Our implemenation of `minCutThreaded()` creates new threads for each iteration while there are still cores on the machine available. For each new thread, we `implement Runnable` by overriding `run()`. This method calls `karger()` and adds the result to the `ConcurrentSkipListSet<Double>` set. This is improved with `fastMincutThreaded()` which `extends RecursiveAction` and overrides `compute()`. This method contracts the graph t times and then runs calls `fastMincutThreaded()` on the result, hence the recursion. Then, we continue on until we have a small enough graph to call `mincut()` and the result is added to the concurrency list for each thread and iteration.

Empiricle Analysis

In this section, we analyze how the algorithms will scale as we increase the number of verticies (nodes) in the graph. We are interested in seeing what kind of improvement we should see from Mincut to Karger-Stein, and how Multi-Threading can improve our results further.

Our first graph is that which compares the `karger(), mincut(), fastMincut()` algorithms. As we can see below, the `karger()` algorithm takes little time at all. As we discussed before, this quickness is a tradeoff for the accuracy, given it is only one iteration.

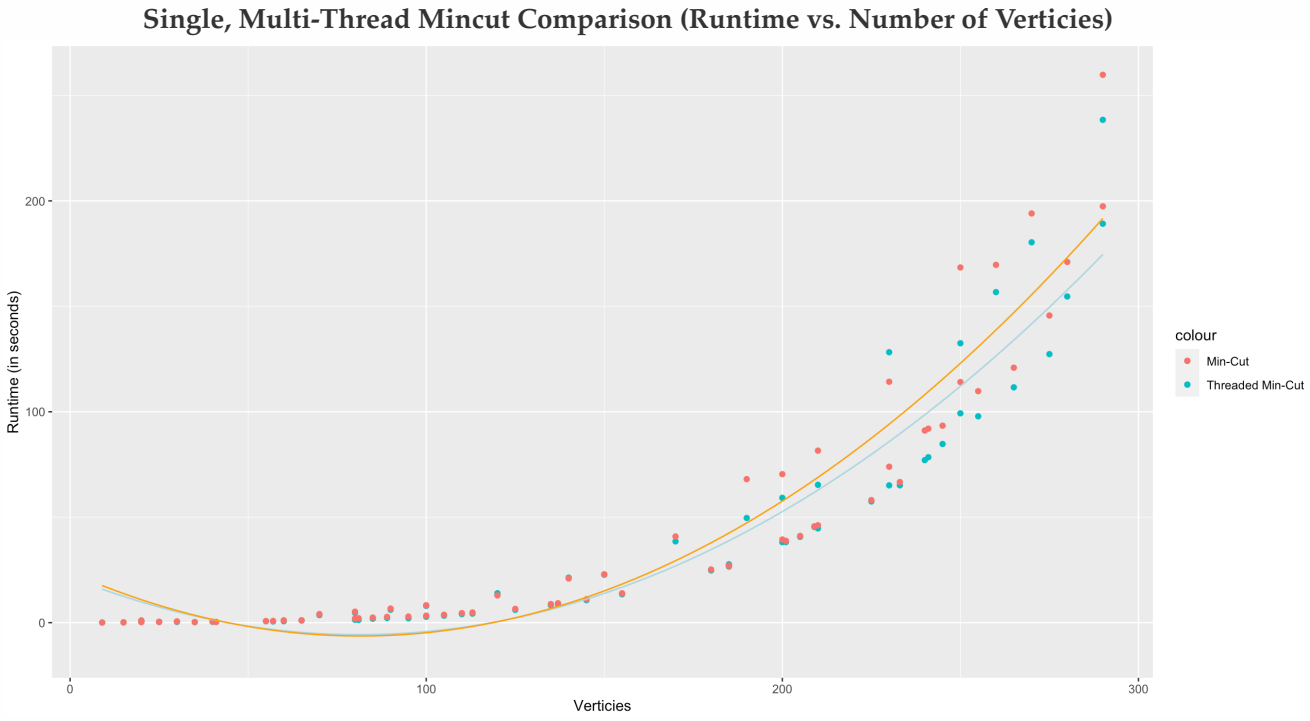
Karger, Mincut, Karger-Stein (Runtime vs. Number of Verticies)



We see a significant speedup fromt the `mincut()` algorithm to the `fastMinCut()` because we prioritize time we spend on the smaller portions of the graph, than on the larger parts. This allows us to run more iterations quicker without too much of a tradeoff in accuracy. The regression for Karger is based on a worst case $O(n^2)$, Min-Cut is based on an analysis for runtime of $O(n^4)$, and The Karger-Stein is

$O(n^2 log(n))$.

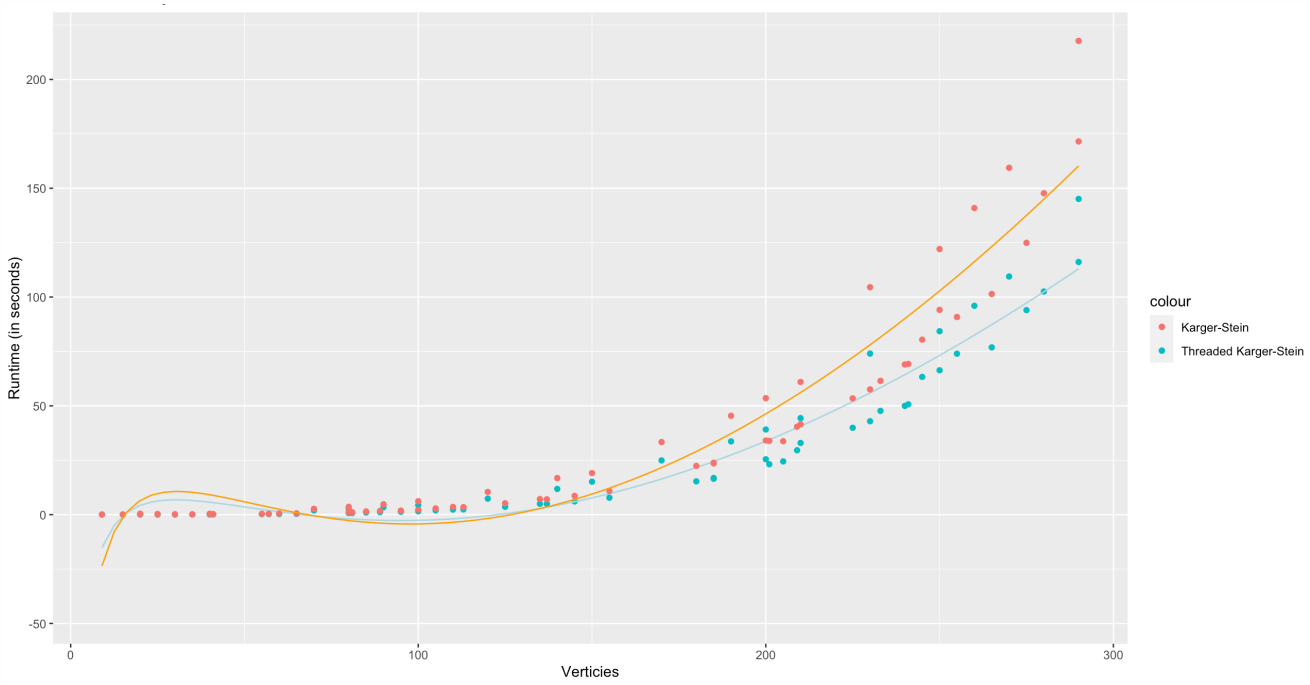
The following is a comparison of the mincut algorithm from single thread to multiple threads.



We see a slight speedup in the algorithm, but a lot of variation between the points, and a lot of overlap in the graph as well. Although this speedup is noticeable in the linear regression, it is not clear that in general we see significant decrease in time. At least, we cannot be sure without further investigation.

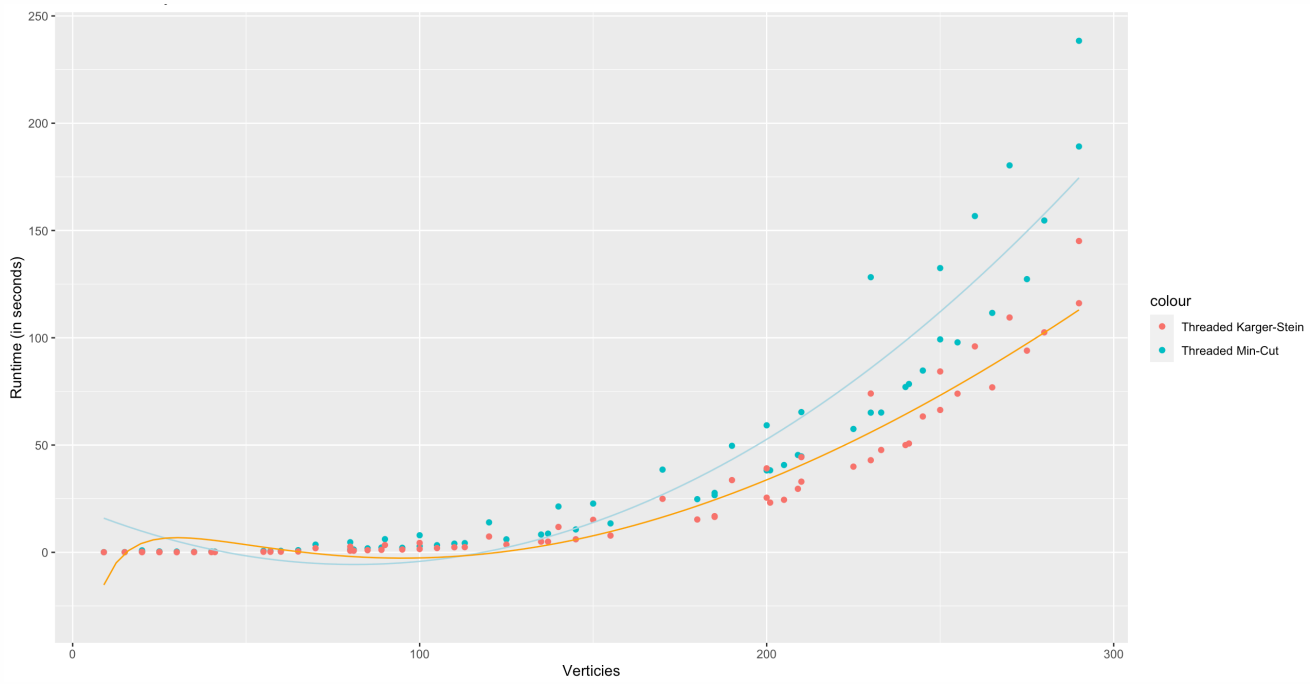
This next graph is that which compares the single and multi-threaded Karger-Stein implementations. This recursive algorithm should be much faster than the minimum cut because we trade of memory space on the program stack for time.

Single, Mult-Thread Karger-Stein Comparison (Runtime vs. Number of Verticies)



As seen above, we have a very significant speedup from the single to the multi-threaded Karger-Stein approaches. Using multiple-threads divides up an algorithm that is already much quicker. And, we see introducing parallelization gives us an improvement as the input nodes increases.

Multi-Threaded Mincut & Karger-Stein (Runtime vs. Number of Verticies)



Finally, we analyze both of the threaded algorithms. Here, our mincut algorithm clearly takes more time than our Karger-Stein.

Thus we have seen by our regression, if V is the number of verticies the following runtime is:

Karger: $O(V^2)$
MinCut: $O(V^4)$
KargerStein: $O(V^2 Log(V))$

Conclusion

For the behaviour of the graphs that we noticed, we can say that we see a significant speedup in the algorithm used for Karger-Stein, which means that our prioritization of the small graphs paid off in the runtime. In this analysis, I used a 4-core machine for the parallelization which gave us 8 virtual cores. Keeping in mind that on a machine with more cores, we will see an even greater speedup in the runtime transitioning to multiple threads.