Implementing AKPW: A Low-Stretch Spanning Tree Algorithm

Jackson Thea

December 22, 2015

1 Abstract

In the last twenty years, the process of generating low-stretch spanning trees has become a bottleneck for several approximation algorithms, including solving certain types of linear systems and finding minimum energy flows [1]. The goal of this set of algorithms is to generate a spanning tree, $T$, from a graph $G$, where the stretch averaged over all the edges is as low as possible. The stretch of a single edge is the length of the shortest path in $T$ between its endpoints divided by the length of the shortest path between its endpoints (usually the length of the edge itself) in $G$. Though there have been a few advancements in these low-stretch spanning tree algorithms over the last twenty years, few have been implemented. I implemented the first algorithm of this set published by Alon, Karp, Peleg and West in 1995 [2] (which I will refer to as AKPW, after the authors’ names), using a relatively new language called Julia. As a high level overview, AKPW partitions the graph into clusters, using the shortest edges first. It then creates a shortest path tree in each cluster, adding those edges to the final tree. Finally, it collapses each of those clusters into single vertices and recurses on the newly-formed graph. The algorithm ends when an iteration of the graph is collapsed into a single vertex. Once I completed the algorithm, I implemented several speed optimizations, making it approximately 10x faster and competitive with Kruskal’s minimum spanning tree algorithm. I also implemented several optimizations that improve its average stretch. The algorithm generates significantly lower stretch than other low-stretch algorithms including Randish Prim, the algorithm used by my advisor, Daniel Spielman, before this project.

2 The Algorithm

AKPW constructs a tree $T$ from a graph $G$ by using the following steps. First, the edge weights in $G$ are normalized so that the smallest edge weight is 1. Every edge in $G$ is then assigned to an edge class based on the edge’s weight, $w$, and a constant $Y$ such that

$$Y = X \ast \log n$$

where $n$ is the number of vertices in $G$ and

$$X = \exp\left(\sqrt{\log n \ast \log \log n}\right)$$

A given edge’s weight class is $\lceil \log_Y w \rceil + 1$. AKPW then enters a number of iterations of partitioning $G$ into clusters, then collapsing each cluster into a single vertex (each time producing a smaller graph with fewer vertices). This process is repeated until the resulting graph contains a single vertex, at which point the algorithm finishes. For each step of the iterative clustering process, edges are ignored if their weight class is greater than the iteration the algorithm is currently on. For example, if the algorithm is on iteration number 3, only edges from classes 1-3 are used in the clustering process – the rest are ignored.
The partitioning process for each cluster starts at a randomly assigned vertex in the graph (for simplicity, I initially navigated the vertices in their graph order). That starting vertex (which I’ll call start) is the root of a breadth first search (BFS) process that ends once the volume/boundary ≥ X, where the volume is the sum of the degrees of the vertices in the cluster, the boundary is the number of edges in the cluster, and X is as it is defined in (2) (here, n is the number of vertices in the current iteration’s graph). Once this constraint is satisfied or the cluster has no more neighbors, the algorithm moves onto the next cluster.

Once the graph is completely partitioned into clusters, a shortest path tree is constructed within each cluster, using each cluster’s BFS start as the root for the shortest path. The algorithm then adds each edge in this shortest path tree to the final AKPW tree (in this step, a single edge in the current graph iteration may correspond to many in the original graph, so I choose the one with the smallest weight).

The algorithm ends when the graph collapses into a single vertex. AKPW then constructs a single tree using all of the edges it has collected during the shortest path steps, and returns a tree constructed from those edges.

3 Testing

In order to test the algorithm for both speed and stretch, I used four different randomly generated graphs: grid, product, necklace and chimera graphs. Each was constructed using Professor Spielman’s graph generation algorithms, then the edges were assigned random weights between 0 and 1 (except chimera graphs, which already have edge weights). The grid, product, and necklace graphs each provided specific and consistent graphs with respect to speed and stretch, with limited variety. Chimera graphs provide a much larger breadth of graph formations, and are thus much more volatile with respect to speed and stretch.

I created a testing function (testAKPW) that runs AKPW on each of these graphs for a variety of sizes based on the function inputs. It then verifies that the result of the algorithm is a tree and that each of the edges in the tree are in the original graph.

4 Improving Stretch

4.1 Examining X

In their original paper, Alon, Karp, Peleg and West describe the threshold for cluster construction by using a variable X, which is a function of the number of vertices in the current graph iteration, n (as described above). As each cluster grows, X is repeatedly compared to the volume/boundary of the cluster, and when that ratio exceeds X, the cluster stops growing. In this way, X directly determines the size and number of clusters used to form the AKPW tree, a crucial factor in AKPW’s effectiveness. If there are too few or too many clusters the tree will be very similar to a shortest path tree, making it important to optimize X. The original paper used equation (2) as X. After some exploration, there seemed to be evidence that

\[ X = \log_2(n + 1) \]  

and a few other equations produced better stretch on average for most graphs (though it often produced worse stretch on a grid). However, after more extensive examination it was discovered that (2) is in fact better; on a sample of 500 random chimera graphs with 10k vertices each, (3) produced stretch that was, on average, 19% larger than (2) did.

4.2 Reshuffling Clusters

I also implemented several more complex strategies aimed at lowering AKPW’s stretch. The first is a process I term reshuffling clusters. This process stems from the hypothesis that there is a way
to shift a number of the vertices from one cluster to another in order to produce a more optimal set of clusters. Such a set would have fewer edges bridging multiple clusters, resulting in a more efficient partition and a better final tree.

I implemented this approach by creating a function that iterates through every vertex in the current iteration of the graph. For each vertex \( v \), the function finds the cluster that contains the majority of the vertex’s neighbors (which I will call cluster B). If that cluster is different from the cluster that contains \( v \) (which I will call cluster A), the function switches \( v \) to this new cluster (\( v \) gets switched from cluster A to cluster B). The algorithm then proceeds as usual, but with these slightly modified clusters. I also make sure this switching doesn’t split clusters (e.g. by switching vertices necessary for connectedness within a cluster) by only considering certain vertices (e.g. “leaf vertices” from the BFS) in each cluster for this process.

To test the effectiveness of this process, I again used a sample of 500 random chimera graphs with 10k vertices each. This technique seemed promising when tested on AKPW using equation (3) for X. In this case, shuffling produced stretch that was, on average, 7% less than the non-reshuffling algorithm. However, when the original (better) equation (2) is used for X, shuffling only gives a 1% advantage, making it not as promising as we had originally thought. Interestingly though, this finding may imply that reshuffling accomplishes many of the same improvements that changing X does.

4.3 Random Clusters

Another technique to improve the stretch is to pick a random starting vertex for each cluster. Before implementing this feature, I was picking the first non-clustered vertex based on the vertex-order given by the graph, because it was simpler to implement and often gave faster running time. The difference between these techniques obviously depends on the vertex-ordering of the graph, but it offers a feature for a user who knows that his or her vertex-ordering is deterministic or well defined. For instance, on a sample of 500 randomly-weighted grid graphs (in which the vertex order is very well defined) with 10k vertices, introducing random clustering produced 12% lower stretch on average. Here, the advantage is the same for both X functions.

4.4 Metis Clustering

The final technique I introduced to improve stretch is a well-known partitioning algorithm called Metis. I implemented a function that partitions the graph using an open-source Julia implementation of Metis (available online at https://github.com/JuliaSparse/Metis.jl) instead of the algorithm described by Alon, Karp, Peleg and West. Unfortunately, this technique was not as effective as the others when tested using chimera graphs. In a sample of 500 random chimera graphs, metis clustering produced, on average, 3 times the stretch than the AKPW clustering algorithm. It also produced 20% larger stretch for both necklace and product graphs. This is much worse stretch, so I wouldn’t recommend using this feature for most graphs. However, metis clustering also produced 9% lower average stretch on 500 randomly weighted grid graphs, so it appears it may be advantageous in some cases.

5 Improving Speed

After making sure the algorithm was correctly constructing AKPW trees, I implemented several speed optimizations, which I’ve outlined below. Before these optimizations, AKPW constructed a tree on a 250,000 vertex grid in 6.7 seconds. Afterwards, it completed a 2,250,000 vertex grid in about 5.5 seconds and a 3,240,000 grid in 8.5 seconds. This is approximately a 10x improvement in speed, bringing it within a factor of 2 of the running time of Kruskal’s minimum spanning tree algorithm on similar graphs.
5.1 Data Structures

I used several data structures in AKPW, some necessary for constructing the algorithm and some simply for speed. The first type of data structure were maps, or arrays that communicate information about a previous version of the graph to the algorithm’s current one. The first map is called vertexToCluster, whose indices correspond to vertices and values correspond to that index’s cluster. The indices in the second map, bigMapD, correspond to vertices in the original graph. Each value is the vertex in the most recent graph iteration that was collapsed from original vertex indicated by the index. The final map, bigEdgeMapReversed, has indices that correspond to edges in the current collapsed iteration of the graph. Each value is the smallest edge from the original graph such that both of its endpoints collapse to the same endpoints as the index’s edge (this is necessary for relaying the information from a shortest path tree in one graph to the original one).

I also built two more data structures to deal with partitioning the graphs. The first is partitionList. This is a list of clusters, such that each cluster is a list of vertices. This allows for quick retrieval of all of the vertices in a specific cluster. Additionally, I built vertexToClusterLocation, which gives me the index of a vertex inside its cluster in partitionList, allowing me to access and change these values in constant time (rather than iterating through an entire cluster looking for a single vertex). Without this last one, reshuffleClusters would be O((number of vertices)*(average number of vertices in a cluster)), rather than just O(number of vertices).

Another recent change I made was the addition of a hashtable to reshuffleClusters. Because the function requires keeping track of the number of neighbors every vertex has in certain clusters, this was originally O((number of vertices)*(number of Clusters)) because I had to initialize the array with nClusters zeros for every vertex. By introducing a hashtable I no longer needed to do this initialization, so the function runs in O(number of vertices).

5.2 Algorithms

I also made a few purely algorithmic changes to improve AKPW’s speed. The first was to calculate the volume and boundary of a cluster (during partitioning) on the fly, rather than once for each step in the BFS. This reduced these step from O(number of vertices In the Cluster) to O(1).

The other major algorithmic optimization involved the function getBigEdgeMapReversed(), in which I create the bigEdgeMapReversed data structure once for every iteration of the graph. This function cycles through every edge in the original graph, then finds the smallest edge in the latest graph that contains the same edge points. I sped this function up significantly by implementing a check to make sure that the endpoints of the original edge did not map to the same vertex in the latest graph before trying to find all of the edges that contain those two endpoints; in most iterations, this reduces the search for a single edge from O(number of neighbors of first endpoint) to O(1) because most endpoints collapse to the same vertex. This decreases running time especially for graphs where the vertices have many neighbors.

5.3 Reducing Memory Allocations

Finally, some of the most effective optimizations involved reducing the number of memory allocations in AKPW. Sparse and findnz, two functions required for creating a new graph, are unfortunately slow. Improving speed meant removing all unnecessary instances of these functions from AKPW, including inside the normalize and denormalize functions, as well as in building the shortest path trees. This change accounts for the running time difference between akpw, which initialized a copy of the graph, and akpw!, which uses the original graph provided.

The function reshuffleClusters moves vertices between clusters in an effort to improve the stretch. At first thought, this process would require many reallocations of partitionList; at least once for each cluster that was affected by a change. However, by pushing added vertices to their new cluster and replacing those vertices by -1 in their old clusters, I was able to remove most allocations from this
function. On the downside, this change produces messier code and requires multiple checks later on in the algorithm (functions must check and continue if a vertex is -1).

The final memory optimization was the most effective one I implemented, increasing the speed of my code 5-fold. Originally for every step involved in constructing a single cluster (each cycle of breadth first search, that is), I was instantiating a new array to keep track of the new vertices to add to the cluster at the end of the cycle. Now the function keeps a single, large array and initializes it.

6 How does AKPW compare?

The algorithm used by Professor Spielman before AKPW to generate low average stretch was an algorithm called Randish Prim. This is similar to Prim’s algorithm for generating minimum spanning trees, but with a randomized process for selecting edges. My implementation of AKPW runs in a similar amount of time as Randish Prim; on a sample of 500 randomly-weighted chimera graphs with 10k vertices, AKPW ran in 16.5 seconds, while Randish Prim ran in 8.7 seconds. On a smaller sample of larger graphs (50 chimeras with 100k vertices) AKPW ran in 27.1 seconds, while Randish Prim took 14.0 seconds. Although AKPW is slower than Randish Prim, the difference in speed not prohibitive.

Additionally, AKPW produces significantly better stretch than both Prim and Randish Prim. On a sample of 500 randomly-weighted chimera graphs with 10k vertices, Prim’s minimum spanning tree generated stretch on average 9 times larger than AKPW. In the same test, Randish Prim generated 31% larger stretch than AKPW. Figure 1 is a graph of AKPW and Randish Prim compared side by side, with the trials sorted by ascending AKPW stretch value.

Figure 1: AKPW and Randish Prim: Sorted by AKPW increasing order
References
