Testing Algorithms for Generating Low Stretch Spanning Trees

Jeffrey Lai

December 16, 2016

Abstract

There are many real applications for finding low-stretch spanning trees where reducing the stretch by a constant factor has real world payoffs. This project tested a new algorithm for generating low-stretch spanning trees. The algorithm is loosely structured on Prim’s algorithm for minimum spanning tree. The key difference was the inclusion of multiple starting points, allowing for growth from different parts of the graph. We tested out multiple different schemes for generating starting points, and currently there is a good framework in place for testing other schemes in place. The schemes we have tested have not shown significant improvement over existing algorithms, and we have managed to definitely rule out certain schemes by providing counter examples.

1 Introduction

Let $G = (V, E, w)$ be a weighted connected graph where $w$ is a function from $E$ to positive real numbers. Given a spanning tree $T$ of $G$ we define the distance in $T$ between $u, v \in V$ to be the length of the unique path in $T$ from $u$ to $v$. 


We then define the stretch of an edge \((u, v) \in E\) as

\[
\text{stretch}_T(u, v) = \frac{\text{dist}_T(u, v)}{w(u, v)}
\]

When studying the stretch of spanning trees, the important value to look at is the average stretch of the spanning tree, defined as

\[
\text{ave-stretch}_T(G) = \frac{1}{|E|} \sum_{(u, v) \in E} \text{stretch}_T(u, v)
\]

There are several slight variations on the definition of stretch that are used in the literature, however transformations on the weight function accounts for most, if not all, of these variations (e.g. \(w \rightarrow 1/w\)).

## 2 Approach

### 2.1 The Algorithm

The algorithm we tested is loosely based on Prim’s algorithm. Recall that Prim’s algorithm is a greedy algorithm for calculating a minimum spanning tree of a graph. The algorithm arbitrarily picks some starting vertex. Then grow outwards from that vertex by greedily picking the minimum distance edge that would not create a cycle, terminating when all vertices have been added to the spanning tree. It is important to note that the minimum spanning tree is not always a low-stretch spanning tree. For example consider the complete graph. One minimum spanning tree of the complete graph is a path graph, which is a high-stretch spanning tree.

Nevertheless a randomized version of Prim’s algorithm does produce low-stretch spanning trees regularly. The algorithm is randomizes the order in which edges are added, adding edges with probability proportional to its length. Specifically instead
of adding the smallest edge to the spanning tree, when edges are first seen they are
assigned an arrival time based on an exponential random variable scaled by its length.
These arrival times are stored in a min-heap and are added in whatever order they
end up.

As mentioned before, the algorithm we tested extended randomized Prim. The
difference is the starting points. Instead of growing out of a single, random vertex, our
algorithm picked various ways of choosing starting points, and once again assigned
arrival times to all the edges in each component. The starting points were also
assigned offsets which were added to the arrival times, so that not all starting points
would start growing immediately. Thus the code was split into two parts, one part
which generated the starting points, and the other which took a set of starting points
and ran randomized prim’s algorithm starting from those starting points.

2.2 Testing

To test the algorithm, we used the “chimera” code in Laplacians.jl used to randomly
generate graphs. These graphs are intended to produce extreme and unusual graphs
that might cause any graph algorithm to fail. The code for this can be found in
src/graphGenerators.jl under the chimera and wtedChimera methods. Thus to test
the algorithm, the algorithm was run on many such randomly generated graphs, with
the stretch being calculated and recorded. In order to benchmark the algorithm, two
other algorithms were used: randishPrim and akpw. randishPrim is the randomized
form of Prim’s algorithm described above. akpw is another low-stretch spanning tree
algorithm that is based off an algorithm described by Alon et al. [AKPW95].

Each variation of the algorithm was tested on 100,000 to 200,000 graphs with
10,000 to 15,000 nodes. Both weighted and unweighted graphs were tested. In order
to evaluate the results afterwards the ratio of the stretches generated by the algo-

---

**CPSC 490**
distribution of these ratios over each all the graphs was used to estimate how well the new algorithm performed.

2.3 Results

Two main different forms of starting point generation were tested. The first was a deterministic doubling scheme. This scheme started with one random point, and proceeded in rounds, doubling the number of starting points each round. In order to pick the next round of starting points, each vertex in the graph was associated to whichever starting point it was closest to, effectively partitioning the graph into balls. Within each ball, the furtherest point was picked as a new starting point.

This method of choosing starting points was found to be quite similar to randomized Prim early on. After adding additional code that tracked how large each component grew from each starting point, it was found that it was in fact behaving almost exactly like randomized Prim, with the entire spanning tree growing out of basically just one vertex. In part this was because set the offset of newly generated points to be the distance between the points, however even after testing and fiddling with this offset, it was determined that this method of start point generation was systematically flawed. The reason was that while initially this idea seemed like it might “evenly” space starting points throughout the graph, there are many examples in which this does not happen. For example relatively isolated and sparse parts of the graph that are “far” from the rest of the graph will be preferentially picked, which if there is a low-density of vertices on these outlying parts, will mean that growing out there provides little to no benefit over randomized Prim, since it affects so few nodes. Thus this idea was deemed a bad heuristic. The relevant code is in the getNewStartPoints and makeStartPoints code in src/fractalPrim.jl.

The second method was somewhat inspired by ideas from [MPX13]. Instead of picking a specific subset of points to be starting points, instead every node was con-
sidered a starting point, with offsets being determined in some manner. The ways in
which different offsets were tried are as follows:

1. Uniformly, with some scale factor. Scale factors included
   - \( c \cdot \log n \) for some constant \( c \)
   - \( \text{diam} G \), estimated by sampling the furthest distance from some vertices
     and doubling it
   - An estimate of the average distance in the graph

2. The weighted degree of the a node. In particular since the graphs have \( 1/w_{ij} \)
   as the distance of edge \((i, j)\) with weight \( w_{ij} \), we use the harmonic sum of the
   weights.

Overall none of these were particularly successful, with on average most of them doing
about as well as randomized Prim and worse than akpw. These non-localized ways
generally do not seem fruitful and it is likely that a more in depth and localized
method would have to be employed if this method is to be successful. Some proposed
candidates for this are scaling the offsets based on a linear combination of the weighted
degree and the weighted degrees of the neighbors, and the size of a ball of fixed radius
around each vertex. To see the methodology and charts more in depth, please refer
to the sample notebook provided.

3 Conclusion

Overall this project has achieved the basic goals set out in the proposal. The original
idea was shown not to be fruitful, and variations upon this idea have also show little
evidence of improvement. While this does not give us a new algorithm for tackling
this problem, it does rule out some initially promising ideas. In addition, a modular
framework has been established that will allow us to quickly test any other ideas
along this vein. I hope to test out a few more scheme of start point generation before I graduate.

4 Acknowledgements

I would like to thank Dan for the interesting project and advice throughout the semester. I would also like Chi Tong for some helpful discussions regarding [MPX13].

5 Appendix

5.1 Laplacians.jl

Laplacians.jl is a package in Julia developed by Professor Spielman’s group containing graph a suite of graph algorithms, with a focus on spectral and algebraic graph theory. The code is maintained on github, with documentation here. The code I contributed is on the branch fprim, with the main contributions being made to src/fractalPrim.jl, notebooks/Fractal Randish Prim.ipynb, and notebooks/Start Points.ipynb.

References
