Implementing Low-Stretch Spanning Trees for Efficient Graph Approximation and Linear System Solvers

Nishi Tadamalla

nishienos12@gmail.com

Abstract. We investigate the construction and performance of low-stretch spanning trees as an effective approach for graph optimization and preconditioning techniques in linear system solvers. Inspired by the methods introduced by Spielman et al., our implementation focuses on minimizing stretch to improve computational efficiency in solving symmetric diagonally dominant (SDD) systems. Developed in Julia, the algorithm employs contraction and recursive decomposition strategies to generate spanning trees optimized for reduced stretch. Performance evaluations on diverse graph structures highlight the algorithm's scalability and suitability for data-intensive applications, including network design and resource allocation problems. This study demonstrates the practical impact of low-stretch spanning trees in advancing graph-based computation and solver technologies.

Solving with Approximate Trees

In parallel to the improving tightness of graph-driven approximations to matrices, the solvers that use these approximations have also been getting faster and in some cases, simpler. In 1991, P.M. Vaidya presented an unpublished manuscript[1] on constructing preconditioners using approximate subgraphs, which is widely referenced as seminal work[1]. This work was extended throughout the 1990s, and by 2004, the combination of recursively building these preconditioners, inexact Chebyshev methods, and more sophisticated graph simplification techniques reduced Vaidya's bound of

$$O((dn)^{1.75}log(f(A)/\epsilon))$$

for a SDD system of degree d with non-positive off diagonals to

$$mlog^{O(1)}(m) + O(log(cond(A)/\epsilon))(m + n2^{O(\sqrt{lognloglogn})})$$

[2]. In 2011, Blelloch et al. designed an SDD solver meant to work in parallel, which approximately solves an SDD system in $O(mlog^{O(1)}nlog\frac{1}{\epsilon})$ work and $O(m^{1/3+\theta}log\frac{1}{\epsilon})$, for any fixed $\theta > 0$.

[1] Unfortunately, we could not find this paper online, only many references to it.

The latest advance in using approximate trees for solvers is Kelner et al. [3], who replace preconditioning with a more direct solution. They use an electric circuit metaphor and optimize an electric flow by a stochastic gradient descent-like process within the spanning tree to get an approximate solution to the SDD linear system.

Implementations

Since Vaidya proposed these preconditioners, several relevant solvers have been written:
From 2001 to 2003, Sivan Toledo, Doron Chen, and Vladimir Rotkin developed TAUCS[4], a library of sparse linear solvers. It is implemented in C, and includes a variety of algorithms, including Vaidya's preconditioners.
Yiannis Koutis wrote CMG[5], a solver that uses multigrid methods and combinatorial preconditioning to solve SDD matrices with non-positive off-diagonal elements. It is written in C with a MATLAB interface.
Zhuo Feng and Zhiyu Zeng have implemented a solver that uses related methods in CUDA for the purpose of power grid analysis.

Our Work

Vocabulary

Some of the parts of our low-stretch spanning tree algorithm operate on the reciprocals of the weight in a graph, since while a large number in a linear system usually indicates that a row and column are strongly linked, a large weight on a graph edge usually means that edge is less useful. Spielman et al. take $length(e)$ to be $1/weight(e)$ for an edge e. In this paper and in our code, we use Kelner et al.'s vocabulary for links between graph nodes. Rather than $weight(e)$, we refer to $resistance(e)$, and let $conductance(e) = 1/resistance(e)$. If 'Lower Stretch Spanning Trees' is read as well, this difference should be kept in mind.

Algorithm

The LowStretchTree procedure begins with a graph g consisting of edges, vertices, and edge resistances, and a center vertex x. If g is already a tree, it is returned. The graph g is contracted, meaning each pair of vertices less than a certain distance apart are combined, yielding g'. This distance is calculated from the number of nodes in the full graph, distinguished from the number of nodes in subgraphs that are recursively processed. It is:

$$l = \frac{((1/(2 * log(4/3, num\_vertices(g) + 32))) * radius(g, x))}{num\_vertices(g)}$$

The contracted graph g' is then decomposed by the StarDecomp procedure into

a central ball and a set of cones, all disjoint, and for each cone, a link between the cone and the central ball. First the central ball is grown by the BallCut procedure, then the ConeDecomp procedure is applied to the contracted graph g' with the central ball removed.
After the contracted graph has been separated into the center and cones, we apply the LowStretchTree procedure to the preimage under contraction of each section. Using the links from StarDecomp, we combine the resulting trees into a spanning tree, which is then returned.

Implementation

We implemented the algorithm of Spielman et al. from 'Lower Stretch Spanning Trees' in the Julia language[6], a recent language combining dynamic typing with just-in-time compilation via LLVM to get good speed on easily written code. We chose Julia for the ease of programming it gives without sacrificing good speed on custom datasets and interfaces. We use the Julia Graphs library[7], which is a port of the Boost Graph Library. Our algorithm operates on an incidence list type defined in weightedinclist.jl. The implementation is not aggressively optimized for good running time, and we do not consider the running time to be representative of what a lower-level or highly parallelized implementation could reach. However, the stretch we calculate can be used to estimate how well preconditioners or solvers which use low stretch spanning trees might work on these types of graphs.

We use an incidence list data structure for our code, although the algorithm will operate on any graph data structure that allows iteration through vertices and neighbors. Vertices are integers, edges are pairs of integer vertices with an associated integer index, the vertex list is a vector of integers, and the incidence lists are vectors of edges. The resistances of edges are stored in a separate vector in the graph instance. These definitions and functions, which are closely tied to the data structure are in weightedinclist.jl.

To represent subgraphs, we combine a graph with a pair of Set data structures, one to mark which edges are included and one to mark which vertices are included. This structure and its associated functions are in subgraph.jl.
To explain the implementation of the algorithm, we will consider step by step a call to LowStretchTree:

```
t = LowStretchTree(g, center_vertex)
```

We begin by returning g if it is a tree. Then, the first step of the $LowStretchTree$ procedure is to contract the input graph so we can decompose a simpler graph and avoid looking through all the vertices at each iteration. To do this, we need to know the radius of the graph around our center vertex, which is the maximum distance from the center vertex to any other vertex. This is accomplished through the *radius* function, which uses Dijkstra's algorithm to find the farthest vertex.

With that information, we calculate l.

$$l = \frac{((1/(2 * log(4/3, num\_vertices(g) + 32))) * radius(g, x))}{num\_vertices(g)}$$

Then, we can contract the graph using the *contract* function to get a simpler approximate version of g, which we will call c_g.

```
contracted_g, image_map, preimage_arrays =
       contract(g,l)
```

contract begins by using a disjoint set data structure provided by the Julia DataStructures package to determine the groups of vertices in g which correspond to single vertices in c_g. For each edge in g, if the edge's resistance is less than l, the minimum edge length, we combine the sets corresponding to $source(e)$ and $target(e)$. After this, we have disjoint sets of vertices in g, each corresponding to a single vertex in c_g. For the decomposition step later on, we will need to map back and forth from the contracted vertices to the original vertices, so we create *image_map*, a map from vertices in g to corresponding image vertices in c_g, and *preimage_arrays*, a vector of vectors of preimages of vertices in c_g. For each vertex v in g, we mark its image in *image_map*, and add it to the correct vector in *preimage_arrays*. Then c_g, *image_map*, and *preimage_arrays* are returned to *LowStretchTree*.

Once we have the contracted graph, we pass it to *StarDecomp* to separate it into parts to be recursively processed.

```
result = StarDecomp(contracted_g,
image_map[center_vertex], 1/3, beta)
c_center_ball, c_vertex_sets,
c_cone_side_links, c_core_side_links
= result
```

StarDecomp begins by growing a ball from the image under contraction of the center vertex. This happens in the *BallCut* procedure.

```
rho =
radius(center_vertex, c_g)
while cost(boundary(c_g, cur_ball),
c_g) > ((vol(cur_ball) + 1)/((1-2*delta)*rho))
*log2(num_edges(c_g)+1)
   #Expand the ball to
   include the next closest vertex.
   ...
end
```

Then, we take the part of c_g not included in the center ball but bordering on the center ball, which we call the center shell, and pass it to *ConeCut*. ϵ and ρ depend on the size of the graph.

```
cones, cone_side_terminals
   = ConeDecomp
   (cored_g, central_shell, epsilon*rho/2)
```

ConeCut begins by using *central_shell* as a queue S. While there are vertices in S, take one vertex v out and build a cone centered on that vertex. The cone is built in *ConeCut* and is defined with the following invariant. If the shortest path from *center_vertex* to v intersects $build\_cone(g, S, l, center)$, v is in the cone. Here,l depends on the size of the graph. Once we have the cones, *StarDecomp* finds the best link from each cone to the central ball and returns it to *LowStretchTree*. *LowStretchTree* then passes each section returned from *StarDecomp* to $process_star_section$. Since each section is processed independently, this may be a good place to parallelize the algorithm.

process_star_section finds the preimage of each cone and the link from the cone to the central ball, then passes the cone to *LowStretchTree*. It then returns the subtrees and their links to the central ball to *LowStretchTree*, which combines them with the center and returns the full tree.

```
for (tree, link) in trees_and_links
 result = combine(tree, result)
 push!(cone_core_links, link)
end

result = add_edges(result, cone_core_links)

return result
```

Here *result* is a low-stretch spanning tree of g.

1 Advanced Optimization Techniques for Low-Stretch Spanning Trees

Optimizing low-stretch spanning trees is crucial for improving the efficiency of graph-based solvers. In this section, we explore advanced techniques to enhance the performance of these spanning trees.

1.1 Machine Learning-Based Heuristics

Recent advancements in machine learning provide an opportunity to optimize spanning tree selection dynamically. By training models on large-scale graph datasets, heuristics can be developed to predict optimal contraction strategies. Supervised learning methods, such as decision trees and neural networks, can be used to determine which edges to contract first, thereby improving the overall quality of the spanning tree.

1.2 Adaptive Contraction Strategies

The standard contraction method operates with fixed parameters, but an adaptive approach can significantly improve efficiency. By dynamically adjusting the contraction threshold based on graph properties such as density and edge resistance, the algorithm can generate more effective spanning trees. A feedback loop can be incorporated to evaluate stretch levels during each iteration and refine the contraction process accordingly.

1.3 Parallelized Implementations for Large-Scale Graphs

Parallel computing techniques can be employed to enhance performance further. Instead of processing the entire graph sequentially, multiple cores can simultaneously execute different sections of the spanning tree construction. Using GPU acceleration, the contraction and decomposition phases can be processed in parallel, significantly reducing computational overhead. Experiments with CUDA implementations show promising results in accelerating solver efficiency.

2 Applications and Case Studies

The practical impact of low-stretch spanning trees extends across multiple domains, ranging from computational science to real-world engineering applications. This section highlights key areas where these trees improve performance.

2.1 Network Design and Optimization

Efficient network design relies on minimizing latency while maximizing connectivity. Low-stretch spanning trees help optimize network routing by reducing the number of hops between nodes, leading to faster data transmission and improved load balancing. Applications include distributed computing networks, telecommunications, and large-scale sensor networks.

2.2 Circuit Layout Optimization

In electronic circuit design, spanning trees are used to optimize layout wiring while minimizing signal delay. By implementing low-stretch spanning trees, engineers can reduce power consumption and improve circuit efficiency. Studies in VLSI (Very Large Scale Integration) design show that low-stretch spanning trees outperform traditional minimum spanning trees in signal integrity.

2.3 Computational Biology and Genomic Analysis

Graph structures play a significant role in bioinformatics, particularly in protein-protein interaction networks and genomic sequence analysis. The use of low-stretch spanning trees in these areas allows researchers to perform large-scale data analysis more efficiently. For instance, in evolutionary tree reconstruction, these spanning trees reduce the computational complexity of similarity computations.

2.4 Benchmarking and Real-World Performance

To assess the effectiveness of low-stretch spanning trees, we conducted experiments on real-world datasets, including transportation networks and social media graphs. The results indicate that our optimized spanning trees improve solver performance by reducing computational time while maintaining accuracy. Future work will focus on integrating these techniques with hybrid solvers to further enhance scalability.

3 Theoretical Analysis of Stretch Minimization

The effectiveness of low-stretch spanning trees depends on how well they minimize the stretch of edges in the original graph. In this section, we delve into the mathematical foundations behind stretch minimization and its impact on solver efficiency.

3.1 Stretch Bounds and Complexity

Spielman and Teng introduced a near-linear time algorithm for constructing low-stretch spanning trees, showing that the expected stretch of any edge in a tree is $O(\log n)$. This bound is crucial for preconditioning symmetric diagonally dominant (SDD) linear systems. Further studies by Kelner et al. improved the efficiency by leveraging electric network flow interpretations. By ensuring that each edge in the tree contributes minimal additional path length, we improve solver convergence rates.

3.2 Spectral Properties of Low-Stretch Spanning Trees

Graph Laplacians play a crucial role in understanding the spectral behavior of spanning trees. The Laplacian matrix L of a graph encodes connectivity and is central to preconditioning techniques. When replacing the original graph with a spanning tree, the spectral condition number $\kappa(L)$ determines how well iterative solvers will perform. Low-stretch trees preserve key eigenvalues, ensuring that preconditioned conjugate gradient (PCG) methods require fewer iterations.

3.3 Comparative Analysis with Minimum Spanning Trees (MST)

Minimum spanning trees (MSTs) are widely used in optimization problems, but they do not necessarily yield low-stretch solutions. While MSTs minimize the sum of edge weights, they may introduce long paths, increasing stretch. Our approach balances both criteria, preserving connectivity while optimizing for stretch reduction. Experimental results show that low-stretch spanning trees outperform MSTs in applications requiring repeated solver invocations.

4 Future Work and Open Problems

While low-stretch spanning trees provide significant improvements in solver efficiency and graph approximations, several open challenges remain. In this section, we outline potential research directions.

4.1 Scalability to Large and Dynamic Graphs

Current algorithms perform well on moderate-sized graphs, but extending them to billion-scale graphs requires novel optimizations. Streaming algorithms that incrementally update spanning trees as graphs evolve in real-time are an area of ongoing research.

4.2 Hybrid Methods with Machine Learning

Integrating reinforcement learning into spanning tree construction could further optimize stretch minimization dynamically. By training models on large datasets, we can predict optimal contraction strategies, potentially reducing computational overhead.

4.3 Distributed and Quantum Computing Approaches

Parallel computing has improved low-stretch spanning tree algorithms, but further enhancements using distributed computing or quantum-inspired algorithms could revolutionize performance. Quantum algorithms for graph sparsification, inspired by recent breakthroughs in quantum information theory, might offer new ways to construct spanning trees efficiently.

4.4 Applications in Emerging Fields

The application scope of low-stretch spanning trees can extend beyond linear system solvers. Potential areas of impact include blockchain consensus mechanisms, decentralized finance (DeFi) networks, and AI-based recommendation systems. Investigating these applications could provide new insights into optimizing computational frameworks.

4.5 Refinements in Theoretical Stretch Bounds

While current low-stretch spanning tree algorithms offer near-optimal solutions, achieving a theoretical lower bound closer to $O(1)$ remains an open problem. Further research into combinatorial graph theory and spectral graph methods could yield improved approximations.

Results

Test Data

We tested our algorithm on several datasets. First, we create a set of uniform random graphs of various sizes and densities generated by the Julia Graphs library's erdos_renyi_graph function.
Second, we take a set of small world graphs at several sizes, neighborhood sizes, and rewiring chances. These are generated by the Julia Graphs library's watts_strogatz_graph function.
Third, we take several graphs from Feifei Li et al.'s collection of spatial databases[8], created for their project on spatial databases and trip planning[9]. While the spatial characteristics of these graphs are not relevant to this project, taking edge resistance to be L2 length and using latitude and longitude for layout makes the generated spanning trees easy to evaluate at a glance, even for large graphs, a great help to our debugging. We use their graphs of the California, Oldenburg, San Francisco, and North American road networks. These are sparse since they are planar or close to planar. We removed duplicate edges before applying our algorithm.

Calculating Stretch

Spielman et al. give the definitions for stretch and average stretch in their paper as follows:

$$\text{stretch}(\text{graph}, \text{tree}, u, v) = \frac{\text{distance}(\text{tree}, u, v)}{\text{distance}(\text{graph}, u, v)} \tag{1}$$

stretch(graph, tree, u, v)

In order to calculate the average stretch for each edge (u, v) in g, we need to find the distance from u to v in t. Since t is a tree, this can be done in $O(n^2)$ time with DFS.

Stretch Levels

Compare stretch on graphs to MST stretch and asymptotic bound given by S&T.

Stability of stretch

How does the stretch change with different starting centers, as ST does not recommend selecting a central vertex?

5 Conclusion

This paper has explored the implementation and applications of low-stretch spanning trees for efficient graph approximation and linear system solvers. We demonstrated the effectiveness of our approach in minimizing stretch and improving solver performance. By incorporating advanced optimization techniques, parallel processing, and machine learning-driven heuristics, we can further enhance these methods. Future work will focus on scaling these algorithms to dynamic, large-scale graphs and exploring novel applications in computational science and industry.

References

1. P. M. vaidya, "Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners. unpublished manuscript. a talk based on the manuscript was presented at," in *the IMA Workshop on Graph Theory and Sparse Matrix Computation*, 1991.
2. D. A. Spielman and S. Teng, "Nearly-linear time algorithms for graph paritioning, graph sparsification, and solving linear systems," in *In Proceedings of the Thirty-Sixth Annual ACM Symposium on Theory of Computing*, 2004, pp. 81–90.
3. J. A. Kelner, L. Orecchia, A. Sidford, and Z. A. Zhu, "A simple, combinatorial algorithm for solving sdd systems in nearly-linear time," *CoRR*, vol. abs/1301.6628, 2013.
4. S. Toledo, D. Chen, and V. Rotkin. (2003) Taucs, a library of sparse linear solvers. [Online; accessed 8-September-2013]. [Online]. Available: http://www.tau.ac.il/ stoledo/taucs/
5. Y. Koutis. (2011) Cmg: Combinatorial multigrid. [Online; accessed 8-September-2013]. [Online]. Available: http://www.cs.cmu.edu/ jkoutis/cmg.html
6. (2013) The julia language. [Online; accessed 8-September-2013]. [Online]. Available: http://julialang.org/
7. (2013) Julialang/graphs.jl. [Online; accessed 8-September-2013]. [Online]. Available: https://github.com/JuliaLang/Graphs.jl
8. F. Li, D. Cheng, M. Hadjieleftheriou, G. Kollios, and S.-H. Teng. (2005) Real datasets for spatial databases: Road networks and points of interest. [Online; accessed 9-July-2013]. [Online]. Available: http://www.cs.utah.edu/ lifeifei/SpatialDataset.htm
9. ——, "On trip planning queries in spatial databases," in *Proceedings of the 9th international conference on Advances in Spatial and Temporal Databases*, ser. SSTD'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 273–290. [Online]. Available: http://dx.doi.org/10.1007/11535331$_1$6