Research Project on Embeddings of Graphs

Supervised by Prof. Michael Elkin.

Abstract

In this project, we implemented and analyzed Elkin's (2011) streaming algorithm for constructing sparse spanners in unweighted, undirected graphs, maintaining constant processing time per edge. The algorithm uses randomized label propagation to build a (2t−1)-spanner that approximates shortest paths. We conducted experiments by varying the number of vertices, edge probability, and stretch factor α, and examined the trade-offs between compression ratio and path quality. Our findings confirm theoretical guarantees and reveal additional behaviors, such as a gradual rise followed by a surprising drop in average stretch as α increases, and a stabilization of stretch standard deviation at high compression levels. These insights suggest that the algorithm produces compact yet high-quality spanners, even under streaming constraints. Future work may extend the analysis to larger and weighted graphs, and explore adaptive tuning of α based on user-defined optimization criteria.

Table of contents

Section 1: The Paper

The algorithm implemented in this project is from the paper by [1] Elkin, M. 2011: "Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners". The paper devises a streaming algorithm for the construction of sparse spanners for unweighted undirected graphs.

The stretch of an edge in a spanner graph refers to the ratio between the shortest path distance in the spanner and the original direct edge in the full graph.

Graph spanners are fundamental structures in algorithmic graph theory, used to approximate distances in large graphs with significantly fewer edges. The paper focuses on the problem of constructing (2t-1) spanners, that is for every edge (u,v) in the original graph, the distance between u and v in the spanner is at most (2t-1).

In addition to stretch guarantees, spanners are evaluated based on how well they compress the graph. A key metric is the compression ratio, defined in this project as |E| / |E'|, where |E| is the number of edges in the original graph and |E'| is the number of edges in the spanner. A higher compression ratio indicates better compression, meaning the spanner retains fewer edges while still approximating the original graph's distances. The challenge lies in achieving high compression without significantly increasing stretch, and Elkin's algorithm is designed to balance both goals effectively in the streaming setting.

Streaming models are frameworks where data arrives as a sequence of items, in this case graph edges, that must be processed online with much smaller memory than the input size. In this context the edges arrive in arbitrary order and the algorithm makes decisions about whether to include each edge in the spanner. This constraint is relevant for large graphs where processing and storing the entire graph in memory isn't possible.

Prior to this paper, a known algorithm is the streaming algorithm of [2] Feigenbaum et al. (2008) which had a processing time per edge of $O(t^2 \cdot \log n \cdot n^{1/(t-1)})$.

Elkin's 2011 paper presents a new algorithm for constructing sparse spanners in the streaming model which, compared to the previous algorithm, constructs a spanner with a smaller number of edges and with a smaller number of bits of space used, using far less processing time per edge without any costs.

The paper provides a streaming algorithm for constructing (2t−1)-spanners with an optimal processing time of O(1) per edge, while also achieving strong guarantees on spanner size and stretch. The algorithm itself uses $O\left(t * (\log n)^{2-\frac{1}{t}} * n^{1+\frac{1}{t}}\right)$ bits of memory and with high probability the spanner contains $O\left(t * n^{1+\frac{1}{t}} * (\log n)^{1-\frac{1}{t}}\right)$ edges.

Furthermore, the paper introduces the first fully dynamic algorithm to offer nontrivial bounds on both insertion and deletion update times, filling a gap left by earlier spanner algorithms that were either static or only efficient in limited scenarios.

These results hold for unweighted graphs and can be extended to some weighted cases with slight modifications. The algorithm presented combines small time edge processing, support for streaming and dynamic models, and efficient space and update performance.

Section 2: The Algorithm

The algorithm implemented in this project is a streaming spanner construction algorithm. It constructs a spanner that approximates the distances of the original graph within a factor of (2t−1), using a simple label propagation technique and minimal state per vertex.

1. Label Structure

   Each vertex v is assigned a label P(v), which encodes the values of a base identifier (initially, the vertex's own ID) and a level (initially 0).
   The label is stored as a single integer:
   P(v) = base + n * level, where n is the number of vertices.
   Labels are compared lexicographically according to level and if equal, according to base id.

2. Radius Sampling

   Each vertex independently draws a random radius r(v) from a truncated geometric distribution, which controls how far its label will propagate.

   Equation 1: truncated geometric probability distribution

   $P(r = k) = p^k\ (1 - p)$, for every $k \in [(t - 2)]$, and $P(r = t - 1) = p^{t-1}$

   This radius determines how many levels the label of a vertex can 'spread' to neighbors.

3. Streaming Edge Processing & Label Propagation

   Edges arrive one by one. For each edge (u, v), the algorithm does the following:

```
if (label of u is greater than label of v) {
    swap u and v;  // ensure u has the smaller label
}

if (u has not yet seen the base of v's label) {
    record the base of v's label in u's memory;

    if (the level of v's label is less than v's radius) {
        create a promoted version of v's label;
        schedule label propagation from u using the promoted label;
        add edge (u, v) as a tree edge;
    } else {
        add edge (u, v) as a cross edge;
    }
}
```

4. Tree and Cross Edges
   Tree Edges T(v) are edges used to propagate labels from node to node.
   Cross Edges X(v) are added when propagation stops — they ensure the spanner remains connected and satisfies the stretch guarantee.
   The spanner output is:
   H = union over v in V of T(v) ∪ X(v)

5. Stretch Guarantee

   The algorithm guarantees that for any edge (u, v) in the original graph, there exists a path in the spanner of length at most (2t - 1).

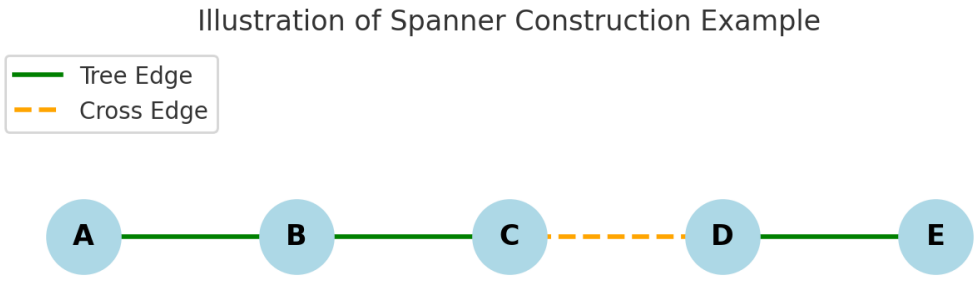
Example:

Consider a small graph of 5 vertices: A, B, C, D, E with edges arriving in order: (A, B), (B, C), (C, D), (D, E).

We can assume A gets a high label with radius 2, B,C and D got label with radius 1.
The label from A propagates to B and then to C because they are within A's radius.
(C, D) does not satisfy the propagation condition (e.g., D is outside the label's radius), so it is added to the spanner as a cross edge.

(D, E) also does not receive any label or propagation, and the edge (D, E) is added to the spanner to maintain connectivity.

The final spanner now includes:
Tree edges: {(A,B), (B,C), (D,E)}}
Cross edge: {(C,D)}



Section 3: Implementation of The Algorithm

This section describes the Python implementation of the streaming spanner algorithm inspired by Elkin's paper. The codebase is modular, and each component is encapsulated in its own file:

- **Main.py** – The main script. Loads configuration, initializes the graph, assigns radii and labels, processes edges, and constructs the spanner using the algorithm.
- **Graph.py** – Handles graph generation and management using the *networkx* library.
- **Vertex.py** – Defines vertex behavior and state, including label, radius, edge sets, and memory table.
- **Edge.py** – Simple representation of an edge between two vertices.
- **Label.py** – Implements label mechanics including hierarchical promotion and base/level management.
- **Spanner.py** – Implements the core algorithm logic, including functions like readEdge, generateRadiusValue, and the radius-label-based decisions.
- **config.py / config.json** – Manage parameters such as vertex count, edge probability, and stretch factor used across runs.
- **Experiments.py** – Automates running multiple experimental configurations and saving their results.
- **Aggregate_Results.py** – Collects output files from all experiment runs and merges them into a unified dataset.
- **Process_Results.py** – Generates visualizations and computes summary statistics from aggregated experiment results.
- **Alpha_Experiment.py / Process_Alpha_Experiment.py** – Scripts dedicated to running and analyzing a focused experiment that varies the stretch factor (α) independently.

Running the project:

1. **Get the Code**
Clone the repository from:
https://github.com/cijhho123/Embeddings-of-Graphs-Project
or unzip the attached project archive.

2. Set Up a Virtual Environment

python -m venv venv

source venv/bin/activate # On Windows: venv\Scripts\activate

3. Install Dependencies

pip install -r requirements.txt

4. Run Experiments
From the root directory of the project, run:

python src/Experiments.py

This will generate results in the output/ directory.

***5.* Post-Process Results**
To aggregate and process the results, run:

python src/Aggregate_Results.py

python src/Process_Results.py

These scripts will generate a summarized CSV and visualizations based on the experiment output on the /processed_output/ directory.

Key Elements:

- Vertices (Vertex.py)

  Each vertex object has a unique identifier labeled id, a label which is an instance of Label class, a radius value drawn from geometric distribution, two sets of edges – tree and cross, a table to track seen label bases (M(v) in the paper).

- Labels (Label.py)

  The labels are stored as an integer where label = level*n + base. Includes methods to promote a label (increment level) and to extract base and level from integer form and baseVertex that links the label to its origin.

- Edges (Edge.py):

  Object storing the two vertices it connects: labeled first and second.

Spanner Construction Flow:

First, Graph.py uses networkx.erdos_renyi_graph() to randomly generate a connected unweighted graph and each graph node is wrapped with a Vertex object thus initializing the vertices.

Second, we assign radii using generateRadiusValue() (in Spanner.py) using a geometric distribution as defined in **Equation 1** where each vertex independently samples a radius $r(v) \in [0, t-1]$.

Next, we initialize labels where each vertex starts with a label includes a base that is the vertex id and a level of 0.

Lastly, we process the edges and labels as implemented in readEdge() (Spanner.py). For each edge the vertices are compared by label, where label promotion happens if the radius allows it.

Both the tree edges and cross edges (respectively T(v) and X(v)) are collected accordingly. The union of all tree and cross edges from all vertices is extracted to a new Graph object as the final spanner.

In implementing the algorithm, we decided to use *networkx* for the graphs as it provides efficient graph structures and algorithms which simplify generation, visualization and actions done on the graph. We used a hash set for M(v), which allowed constant-time

checks and insertions for seen label bases, matching the paper’s goal of minimal state per vertex.

Section 4: Research question

The research question is how does the behaviour of the constructed spanner change when the underlying graph structure varies?

Specifically, we examine how the distribution of edge stretches in the resulting spanner is affected by:

- The number of vertices (n) in the graph
- The probability (p) of an edge being included in the Erdős–Rényi random graph model as described below:

The Erdős–Rényi random graph model is models for generating random graph or networks. Given **n** vertices, each edge (v, u) has a fixed probability of being present in probability **p** or absent in probability **(1-p)**, independently of the other edges.

The stretch of an edge refers to the ratio between the shortest path distance in the spanner and the original direct edge in the full graph. While the algorithm guarantees a worst case stretch of (2t−1), this project explores how the stretch behaves on average or in distribution when the graph is larger with an increased number of vertices and when the graph is denser by changing the edge creation probability.

In this project, the stretch parameter **t** is explored across a practical range, focusing primarily on values from 1 up to $\log(n)$ as suggested by Elkin's paper, where the algorithm offers its strongest guarantees.

For $t = \omega(\log n)$, the spanner size bound worsens, making such values less relevant in typical use cases. To examine the transition between efficient and degraded performance, we include a dedicated set of experiments later in the paper—referred to as the "alpha experiments", which systematically vary **t** (or equivalently the stretch factor $\alpha = 2t - 1$) to observe the algorithm's behaviour as it approaches and exceeds this theoretical threshold.

The underlying goal is to better understand how often edges in the spanner are stretched close to the bound, whether the average stretch remains low in practice on average and how the spanner size and structure are influenced by graph density and size.

By systematically modifying these parameters and measuring the resulting stretch distributions and spanner statistics, the aim is to uncover practical insights into the performance and scalability of the algorithm beyond its theoretical guarantees.

Section 5: Experiments

This section presents a systematic evaluation of the streaming spanner algorithm's empirical performance across a diverse range of graph configurations.

The experiments were conducted using the following controlled parameters:

- Vertex Count (n):
  5, 20, 50, 100, 150, 200, 250, 300

  Larger graphs beyond 300 vertices were excluded due to high computation time.

- Edge Probability (p):
  0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.95

  This parameter governs the density of edges in the graphs.

- Stretch Factor (α):
  1, 3, 5, 7, 15, 25, 35, 49

  A smaller α gives a tighter bound on stretch, whereas larger values allow more relaxed distances but potentially lead to smaller spanners. Larger α are not as relevant to the algorithm as beyond log(n) the restraint on the spanner weakens.

Each unique (n, p, α) configuration was executed multiple times to account for randomness. Results were aggregated to improve statistical reliability.

Experimental Workflow

For each run, the following were performed:

1. Graph Generation
   A random unweighted graph was generated using our model, with vertex count n and edge probability p.

2. Spanner Construction
   The streaming spanner algorithm was applied with the specified α.

3. Metric Collection
   The following metrics were computed:

   - Stretch Distribution: frequency of edge stretch values in the spanner.
   - Compression Ratio: |E| / |E'| — number of edges in original graph divided by the number of edges in the spanner.

- Statistical Summary: average, median, mode, max, and standard deviation of stretch.
- Output Files
  Each run generated two outputs: a CSV file summarizing the statistics and a PNG image showing a histogram of stretch values (e.g., 1.0 = no stretch; 2.0 = path doubled in length).

Example of a CSV output:

Metric,Value

compression_ratio,12.085972850678733

average,1.63485

median,1.5

std_dev,0.69508

mode,1.0

max,4.0

alpha,15

vertex_count,200

edge_probability,0.4

graph_type,unweighted

graph_seed,316181

Following the experiments, all CSV and image outputs were automatically collected and processed using a dedicated analysis script.

The goal was to uncover trends and tradeoffs between structural graph parameters and the performance of the streaming spanner algorithm. The following relationships were examined:

- Compression vs. Stretch Tradeoffs
  Exploring how the compression ratio relates to average, median, mode, standard deviation, and maximum stretch.

- Impact of α (stretch bound)
  investigating how varying the stretch factor affects spanner size and stretch distribution.
- Scalability with vertex count
  Analyzing how increasing the number of nodes impacts the compression ratio and stretch variability.
- Effect of graph density
  Assessing how changes in edge probability influence stretch characteristics and spanner compactness.
- Multi parameter interactions
  Studying “heatmaps” and 3D surfaces for combined effects, such as (α, n) vs. average stretch or (n, p) vs. max stretch.

These insights will inform when and how the streaming spanner algorithm is best applied in practice, highlighting the conditions under which it remains both efficient and accurate, as well as where compromises arise between sparsity and the path length bounded.

Section 6: Experiments Results

This section presents a detailed examination of the empirical behavior of the streaming spanner algorithm across various graph configurations.

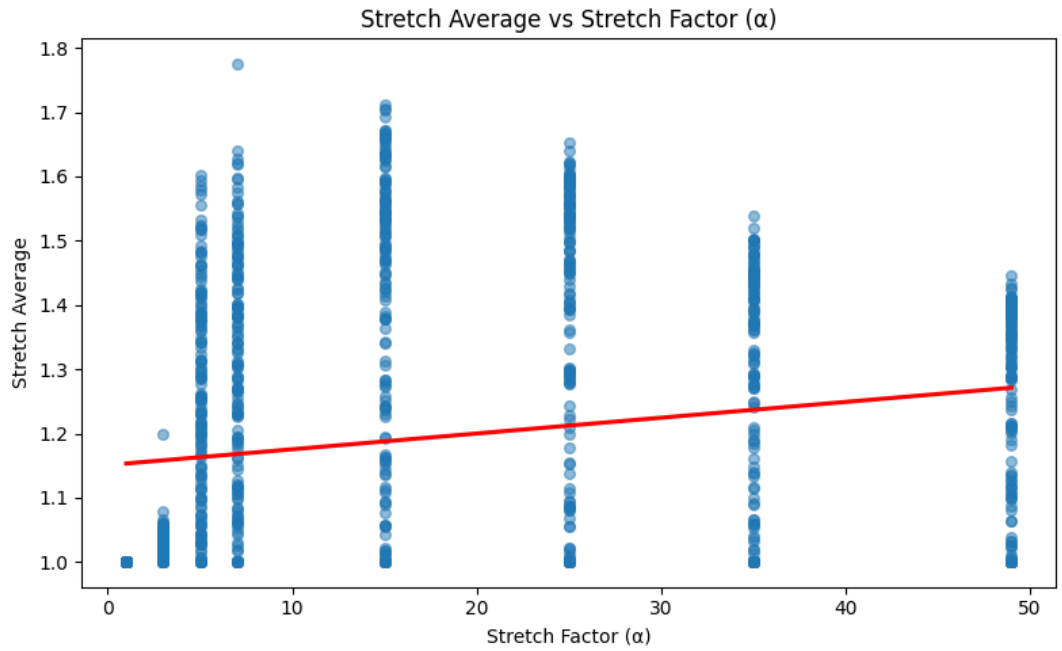
We begin by exploring foundational pairwise relationships, such as the effect of the stretch factor on average and maximum stretch, and the tradeoff between stretch and compression ratio.

From there, we progressively analyze how the structure of the input graph (e.g., size and density) influences these outcomes.

Finally, we examine more nuanced, multi-variable interactions using color-coded plots, highlighting how different parameters jointly affect the spanner's efficiency and accuracy.

Basic Pairwise Relationships

Stretch Average vs. Stretch Factor (α)



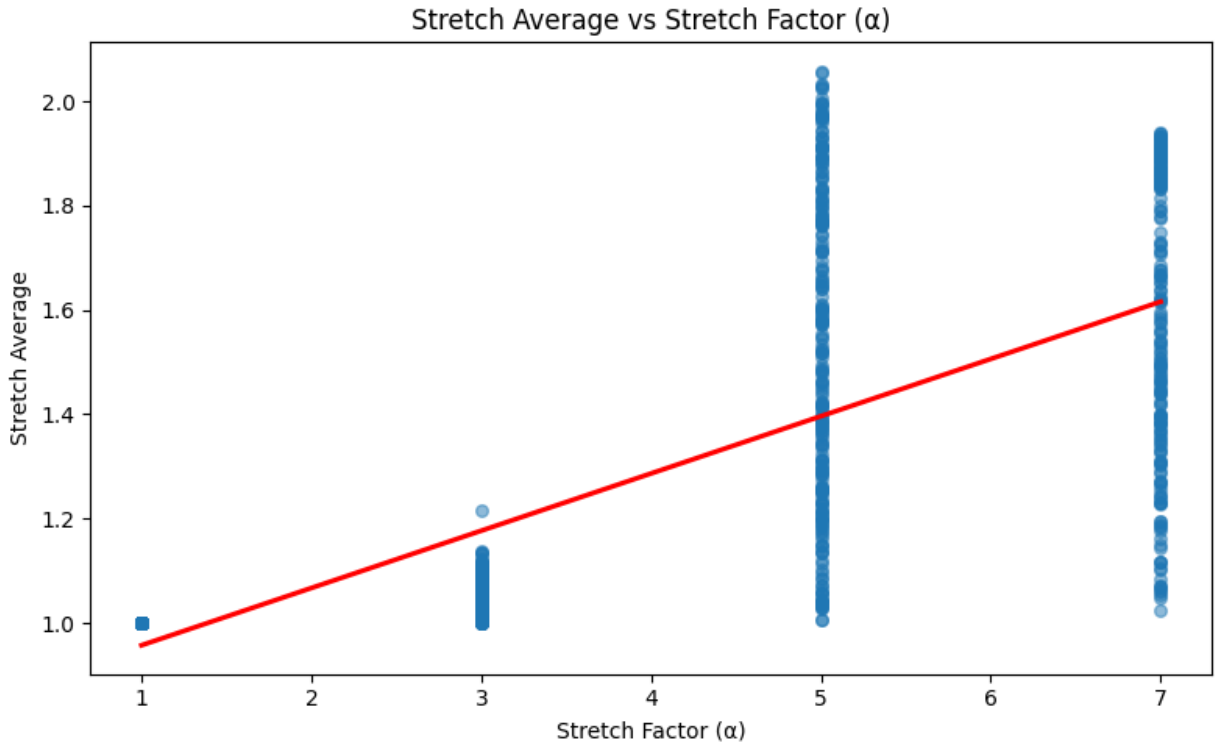
This plot examines how the average stretch of the spanner changes as the stretch factor α increases. As expected, we observe a modest upward trend: higher α values typically allow more relaxed path lengths, resulting in slightly increased average stretch.

However, as seen in the plot, increasing alpha beyond a certain point causes the average stretch to go down, probably due to t values that are higher than log(n) which cause the restraint on the spanner to be worse.

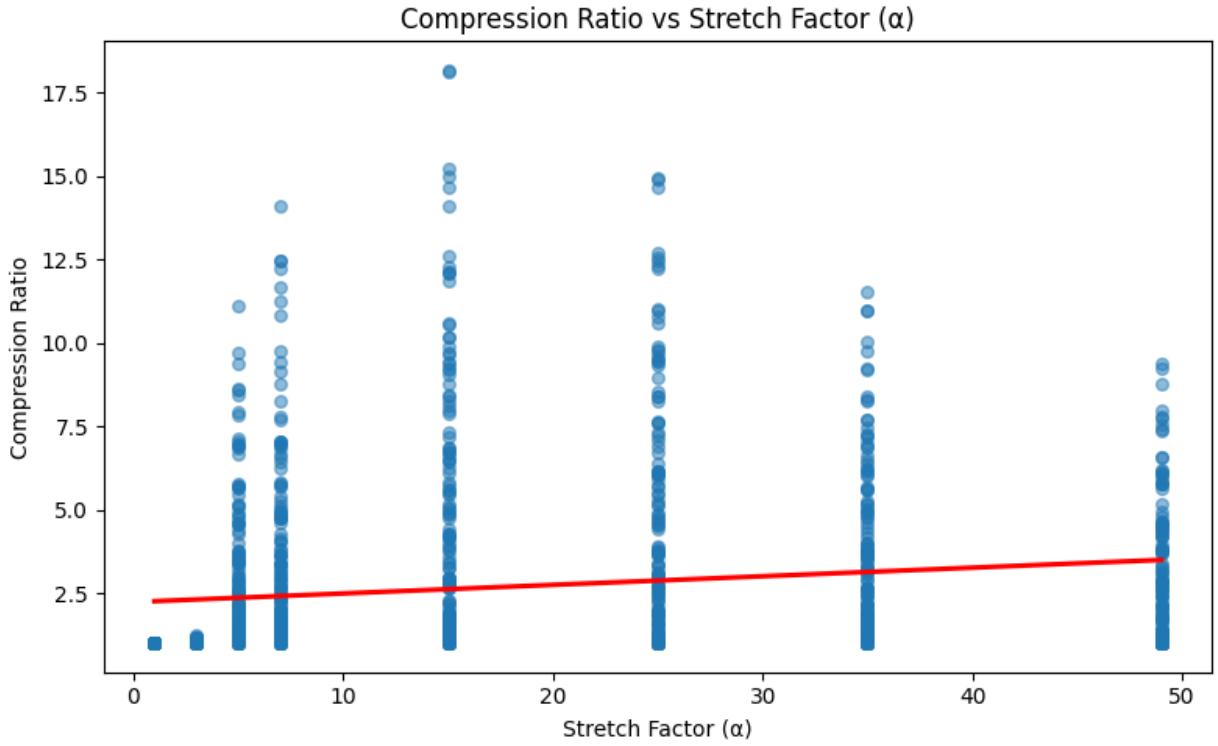
Overall, the graph suggests that the algorithm maintains reasonably efficient paths even when the allowed stretch factor is relaxed significantly.

To get a cleaner result, we also created a separate graph containing only the parameter combinations where **t** is strictly lower than log(n).

Stretch Average vs. Stretch Factor (α) – on graphs where $t \leq log(n)$



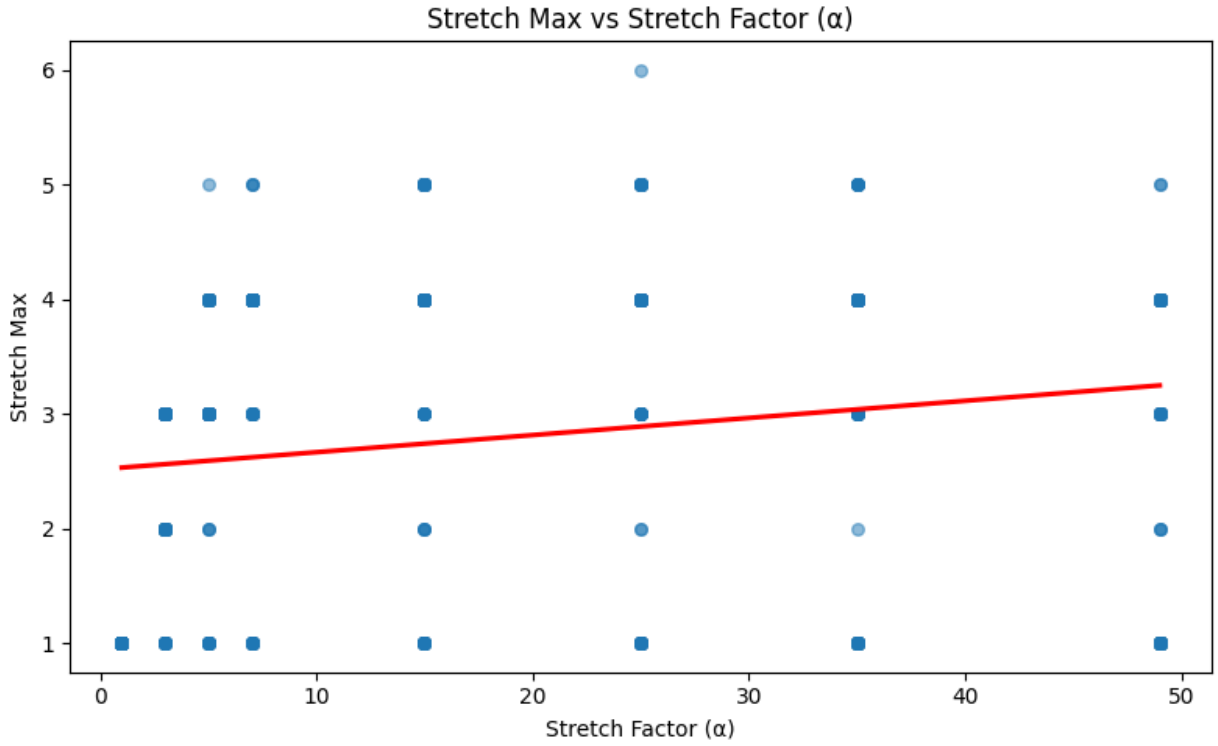
Compression Ratio vs. Stretch Factor (α)



The plot shows a clear positive correlation between the allowed stretch factor and the compression ratio for lower alpha values. This is expected as permitting greater stretch enables the algorithm to drop more edges while maintaining connectivity guarantees leading to sparser spanners.

From a certain value of alpha (around 15) the compression starts to deteriorate, as we go beyond $t \leq \log(n)$.

Stretch Max vs. Stretch Factor (α)

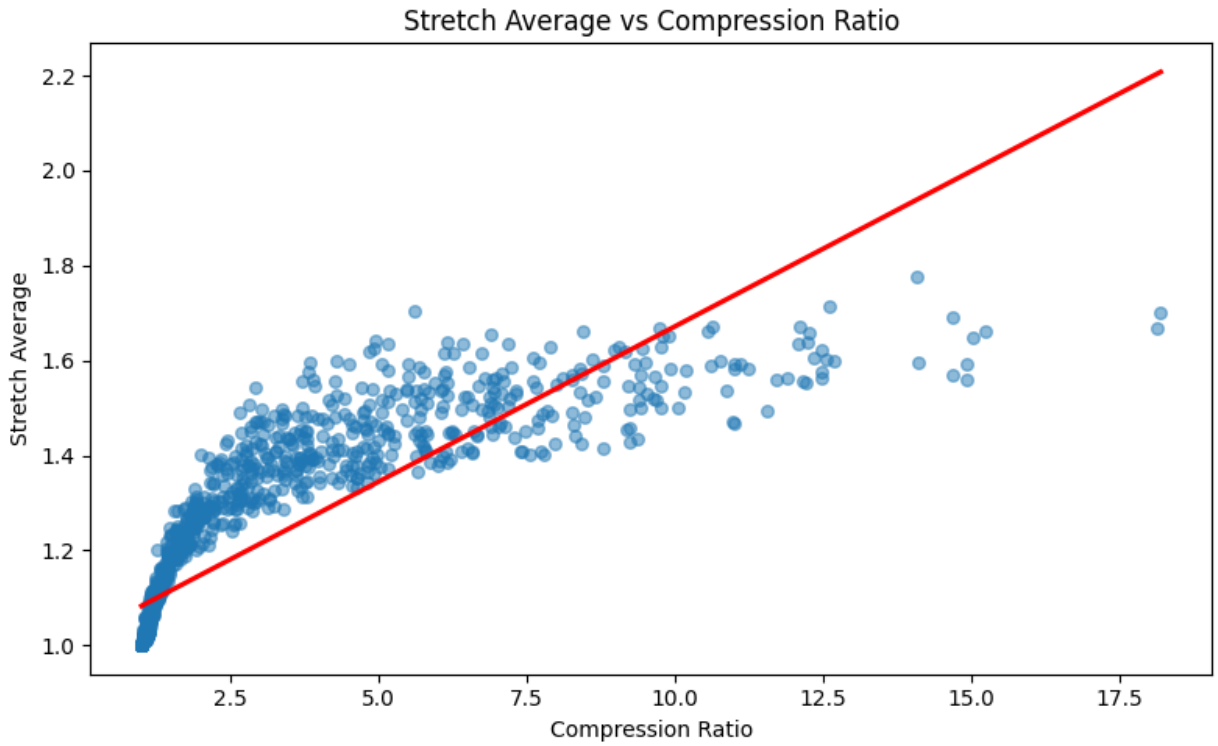


This plot explores how the maximum stretch across all edges varies with the allowed stretch factor α. As expected, the general trend shows a slight increase in maximum stretch as α increases.

However, the data also reveals substantial variability, with outliers appearing even at lower α values. This indicates that while increasing α generally allows for longer detours in the spanner, high-stretch outliers can occur regardless of α.

These results are consistent with theoretical behavior, but the relatively mild slope may again reflect the limited graph sizes used; larger graphs might amplify extreme stretch values more clearly.

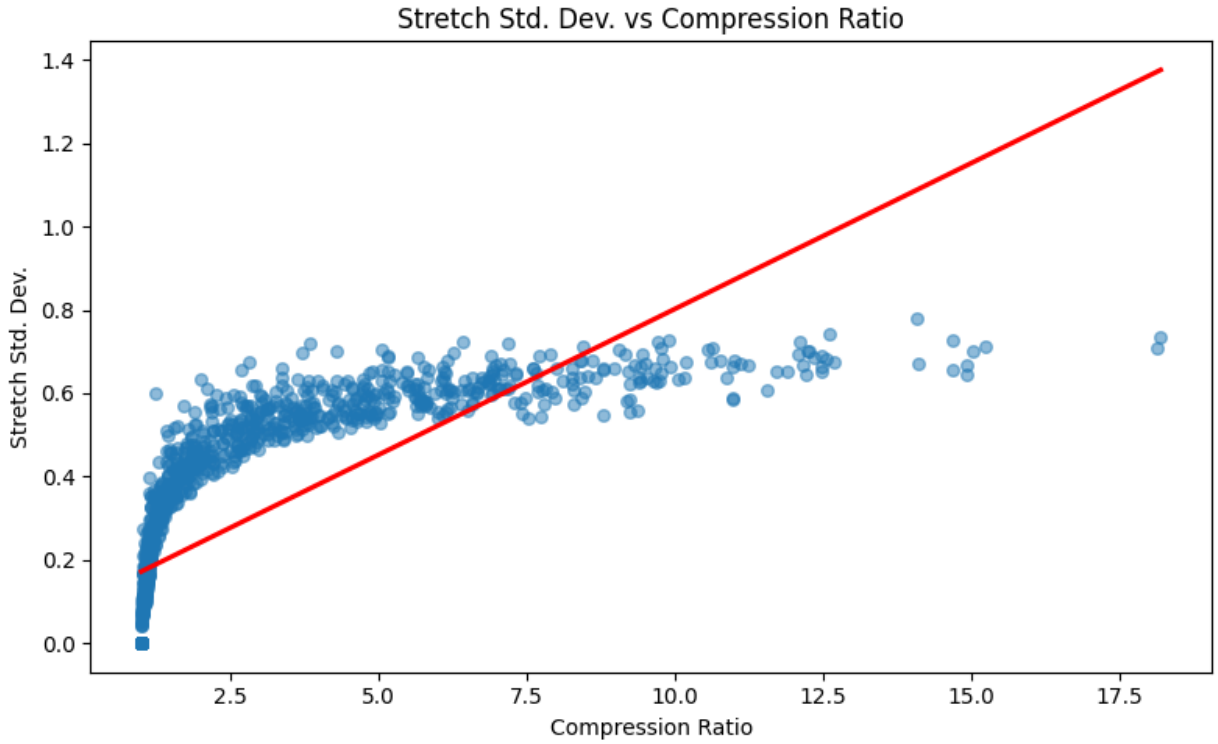
Stretch Average vs. Compression Ratio



This graph illustrates the tradeoff between spanner sparsity and path quality. As the compression ratio increases (indicating fewer edges retained), the average stretch also tends to rise.

This is expected: sparser graphs naturally introduce longer detours. Interestingly, while the lower bound of stretch increases gradually, the average seems to saturate or plateau around certain values, even at high compression ratios. This suggests diminishing returns beyond a certain point, further edge removal doesn't drastically worsen the average stretch.

Stretch Std. Dev. vs. Compression Ratio



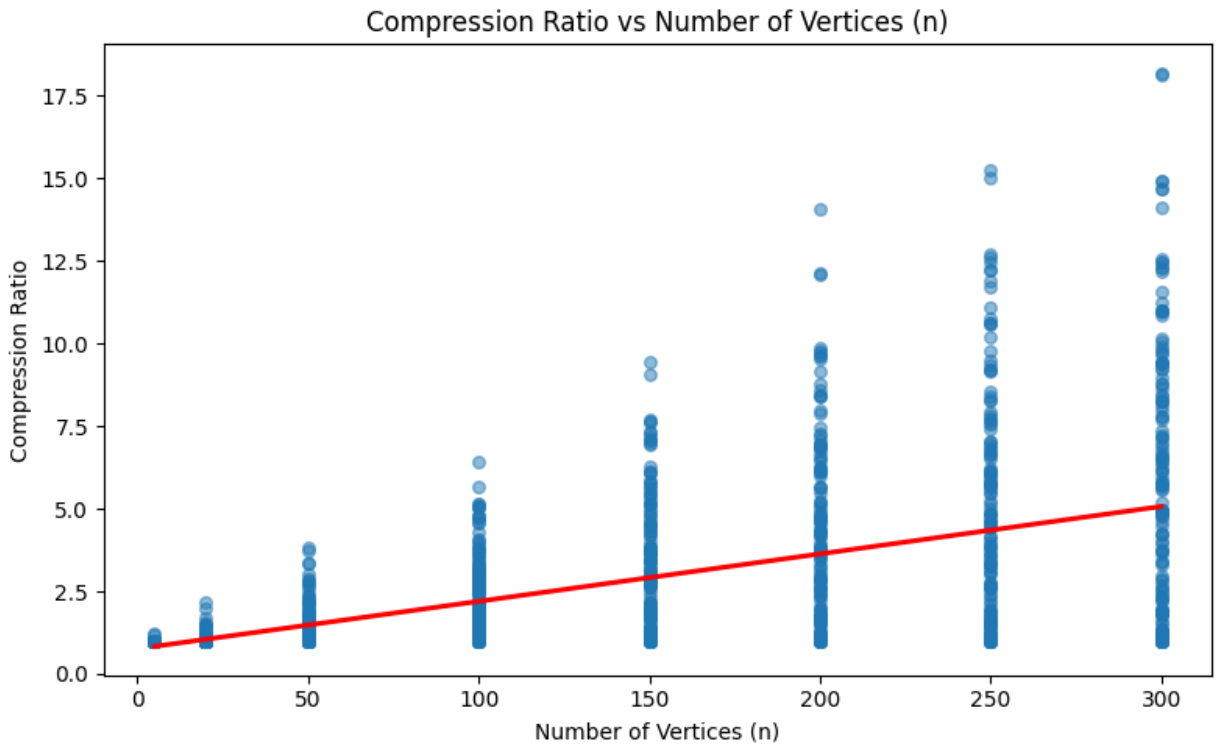
While the red trend line suggests a linear increase, a closer look at the scatter points reveals a plateauing effect: the standard deviation of stretch grows quickly at first but then stabilizes around 0.6–0.7 for higher compression ratios.

This indicates that beyond a certain level of sparsity, additional compression doesn't significantly increase variability in stretch.

This behavior is insightful, it implies that after some threshold, further compression may not worsen worst-case consistency much, even if average or max stretch might still grow.

Structural Effects (Graph Size/Density Effects)

Compression Ratio vs. Number of Vertices (n)

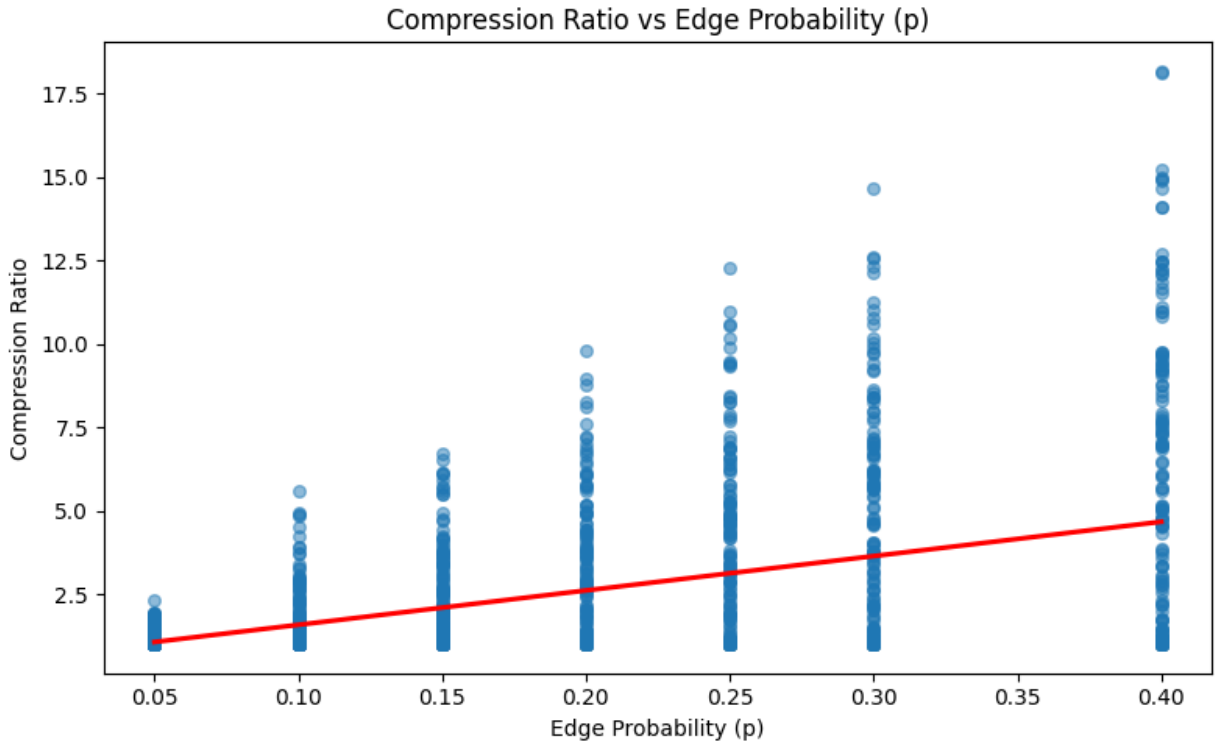


This plot shows that as the number of vertices increases, the compression ratio tends to rise as well, indicating that larger graphs offer more room for edge sparsification.

This trend is intuitive: in denser input graphs (which grow quadratically in edges with increasing n), the spanner can remove proportionally more edges while still maintaining the desired stretch guarantees.

The trend line supports this upward trajectory, and the variance also grows with n, suggesting that sparsifiability becomes more sensitive to other factors (like edge probability or stretch factor) as graph size increases.

Compression Ratio vs. Edge Probability (p)

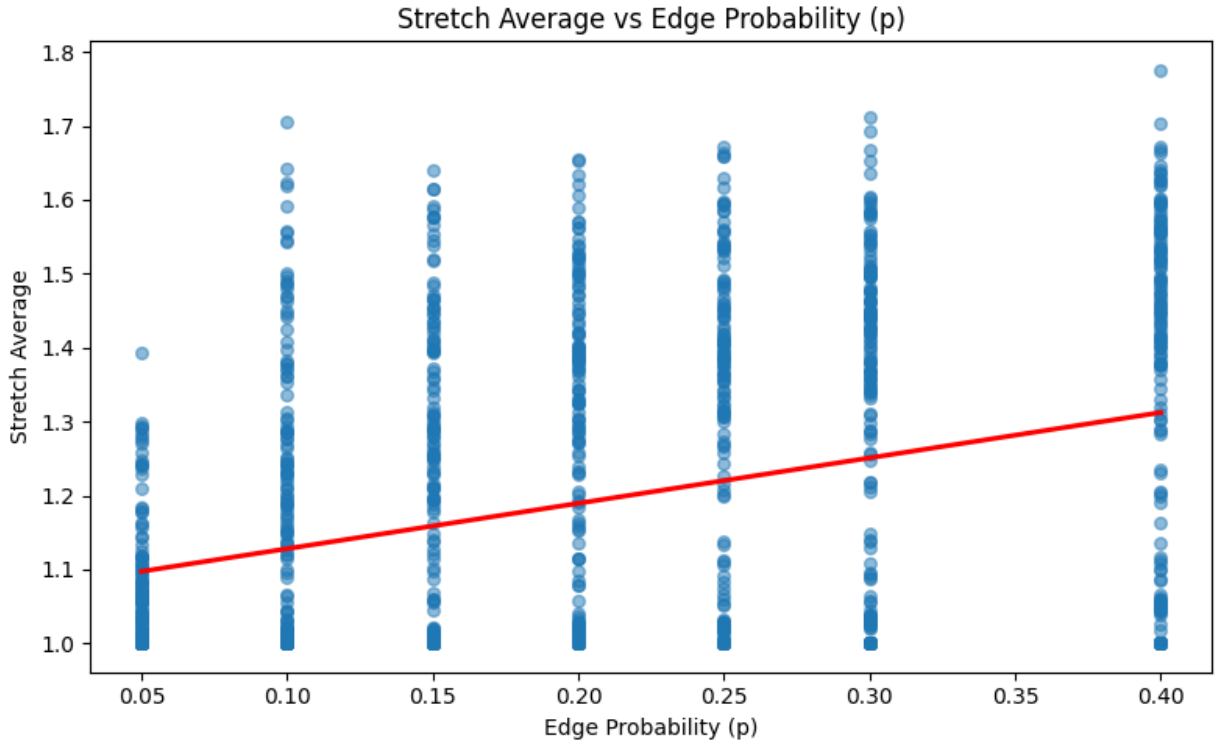


As expected, the graph reveals that higher edge probabilities (denser graphs) result in higher compression ratios.

This aligns with the intuition that in denser graphs, there are more redundant edges that can be safely removed by the spanner while maintaining the desired stretch.

The data points show a consistent upward trend, and the spread indicates a wide range of compressibility even at similar densities, likely due to the interplay with other parameters like n and α. This relationship confirms the algorithm's effectiveness in exploiting redundancy in dense graphs.

Stretch Average vs. Edge Probability (p)

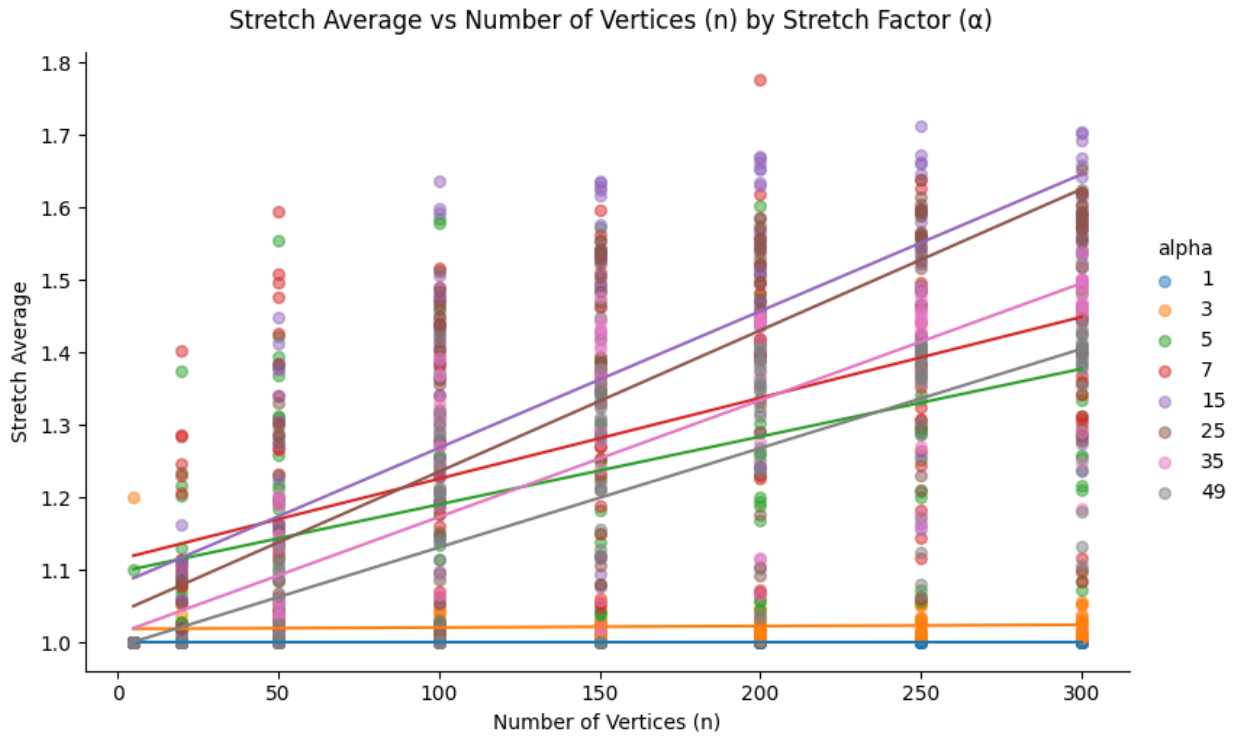


This graph shows a subtle yet noticeable upward trend in average stretch as edge probability increases.

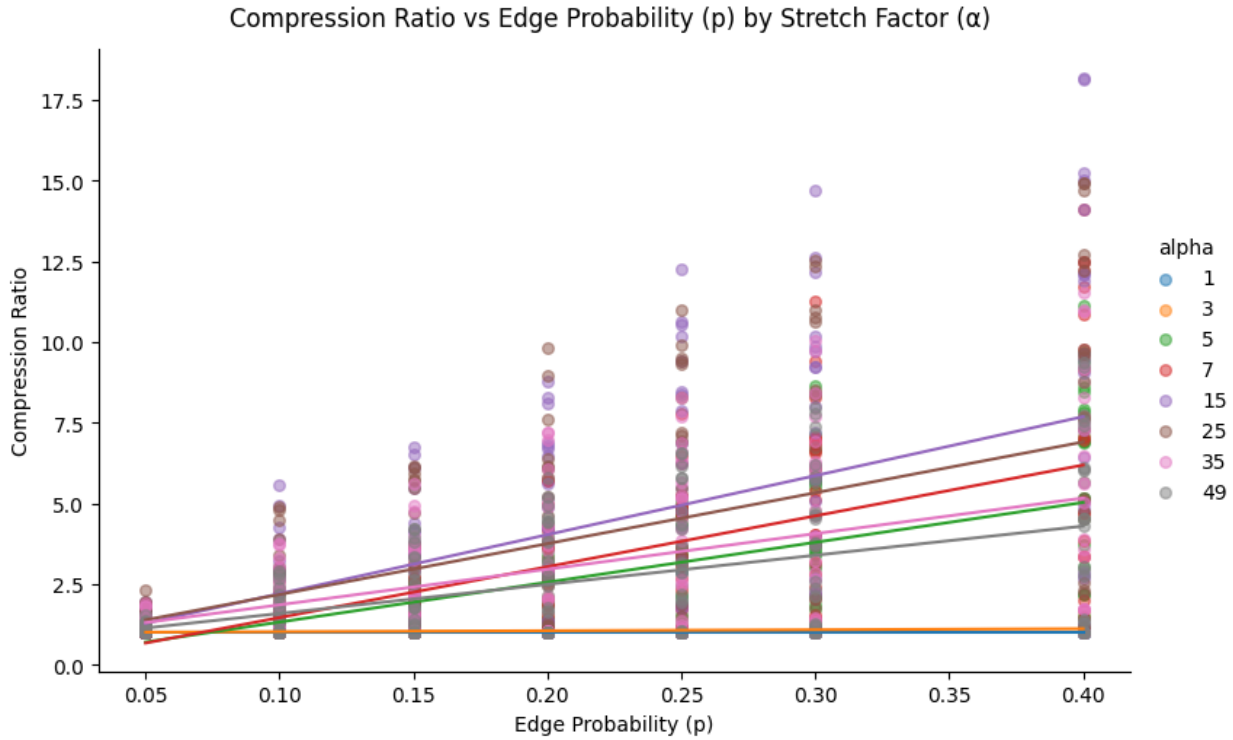
At first glance, this might seem counterintuitive, denser graphs might be expected to allow for shorter paths, thus lowering stretch. However, the spanner aggressively prunes edges in denser graphs to maximize compression, which can lead to slightly increased path lengths.

This tradeoff is a key design feature of spanners, and the relatively modest increase in stretch confirms the algorithm maintains good path quality even when heavily compressing dense graphs.

Stretch Average vs. Number of Vertices (n) by Stretch Factor (α)



Compression Ratio vs. Edge Probability (p) by Stretch Factor (α)



This pair of plots explores how compression ratio evolves with structural graph properties, namely, the number of vertices (n) and edge probability (p), while controlling for the stretch factor (α). Each line represents a fixed α value, making it easy to observe how different α levels modulate compression outcomes.

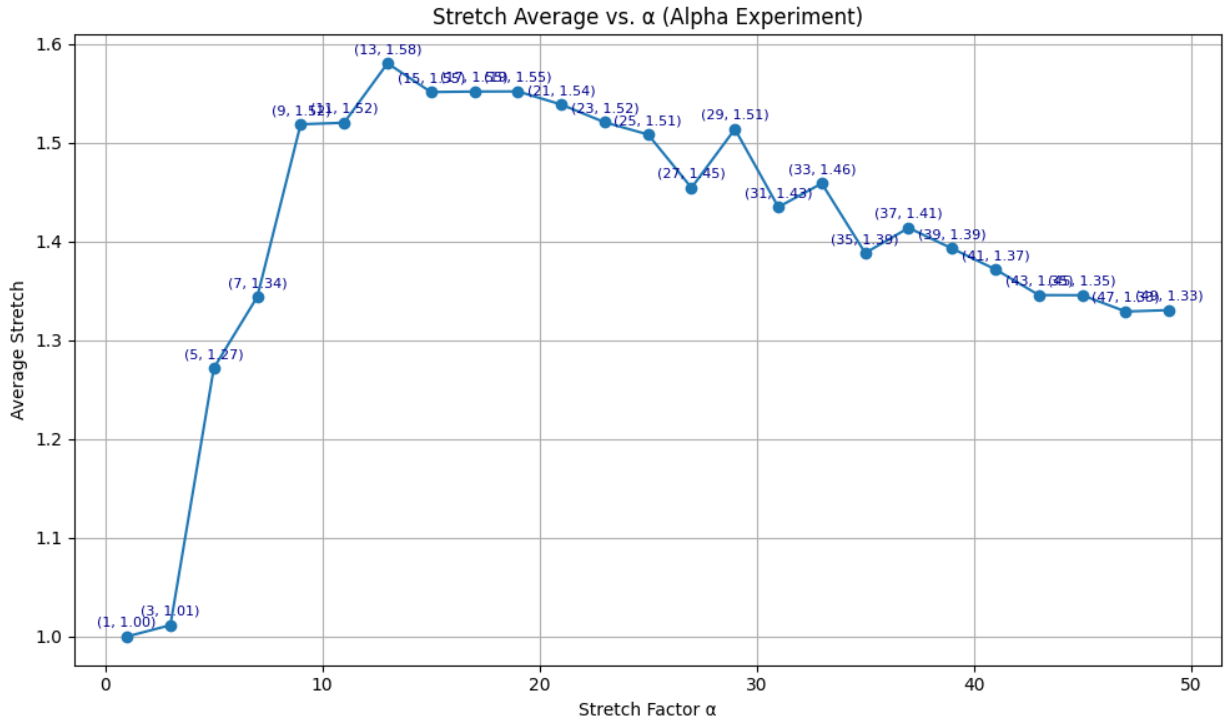
As expected, for small α values (e.g., 1 or 3), the compression ratio remains consistently low across all structural configurations, reflecting tight spanner constraints. As α increases, the compression ratio also grows, particularly between α = 5 and α = 15, where the spanner gains flexibility to discard more edges without violating the stretch bound.

However, a turning point appears beyond α ≈ 20, where $t \geq \log(n)$, the compression ratio no longer improves and, in some cases, begins to plateau or decline. This non-monotonic behaviour support the claim from the paper where a very high α values lead to sparser graphs that are no longer able to capture useful structural shortcuts, resulting in diminishing compression returns.

Planned Follow-Up Experiment (i.e. Alpha Experiment)

To test the claim that beyond $t \geq \log(n)$ the spanner's qualities decline and to better understand this critical α region, we plan a focused experiment where the graph size (n) and edge probability (p) are held constant while α is varied. This will allow us to pinpoint the stretch factor that maximizes compression ratio, validating whether $t = \log(n)$ represents an upper bound for the range of useful stretch factor.

Stretch Average vs. α (Alpha Experiment)



This plot explores how the average stretch evolves as the stretch factor α increases, using a fixed graph configuration with 200 vertices and an edge probability of 0.15. Each data point represents the average stretch over five randomized runs per α value, ensuring a robust signal across stochastic variability.

As expected, for small values of α (1 to 3), the stretch remains tightly constrained, with values near the theoretical minimum (just above 1.0). This reflects the strict path length preservation requirements at low α levels. As α increases beyond 5, the spanner is permitted to discard more edges, which naturally increases the stretch, the average stretch climbs sharply, peaking around α = 13 to 15 with an average stretch of ~1.58.

This growth is not unbounded. Beyond α ≈ 15, the curve begins to flatten and eventually decline. By the time α reaches 35 to 49, the average stretch falls back toward ~1.33.

This finding confirms claim from the paper that beyond $t \geq \log(n)$, the spanner's qualities worsen.

Section 7: Conclusion and directions for future work

In the project we implemented and analyzed a streaming spanner construction algorithm providing the theoretical background and practical insights shown by empirical data.

We developed the algorithm in Python using key elements from the chosen paper, keeping O(1) processing time per edge and including efficient data structures. That implementation was used to gather data through changing the graph size, density and stretch factor needed which revealed some important behaviours.

As expected, there were tradeoffs between compression and average stretch, denser graphs achieved higher compression ratios as well as larger graphs showed increased compression potential. These show that the algorithm finds and exploits the ways to compression and gets close to the potential even though it works as a streaming algorithm.

These findings show that the algorithm stays consistent in it's performance across different graphs and alphas with low variance across the runs.

Future work could prioritize extending the analysis into bigger graphs to confirm the findings and show more accurate findings as well as implementing and analyzing the algorithm for weighted graphs asking if the findings stay similar.

The algorithm itself could be improved to be more useful for practical needs by extending it to find optimal alpha based on the user's needs and optimized according to the metrics investigated like stretch or average edge length and extending the algorithm for weighted graphs.

The data analysis on the given algorithm reveals new behaviour outside of the theoretical bounds, showing trade-offs and expected behaviour for a range of variables allowing us to understand the implementation and construction of spanners done by it. Future works could allow us to understand these behaviors to a larger extent and find out new behaviours.

Bibliography

[1] Michael Elkin. 2011. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. ACM Trans. Algorithms 7, 2, Article 20 (March 2011).

[2] Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., and Zhang, J. 2008. Graph distances in the streaming model: The value of space. SIAM J. Comput. 38, 1700–1727.