**Mini – Project on Embeddings of Graphs**

Directed by Prof. Michael Elkin.

**Table of contents**

1. Chosen paper
2. Algorithm
3. Implementation of The Algorithm
4. Research question
5. Experiments
6. Implementation of The Experiments
7. Experiments Results
8. Conclusion and directions for future work

# **Section 1: Chosen Paper**

The algorithm implemented in this project is from the paper by 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.

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).

Prior to this paper, a known algorithm is the streaming algorithm of Feigenbaum et al. (2008) which had a processing time per edge of O(t2·log n·n1/(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 per-edge processing time of O(1), while also achieving strong guarantees on spanner size and stretch. The algorithm itself uses bits of memory and with high probability the spanner contains edges.

Furthermore, the paper introduces the first fully dynamic algorithm to offer non-trivial 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 little time edge processing, support for streaming and dynamic models, and efficient space and update performance.

# **Section 2: 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.

1. Radius Sampling

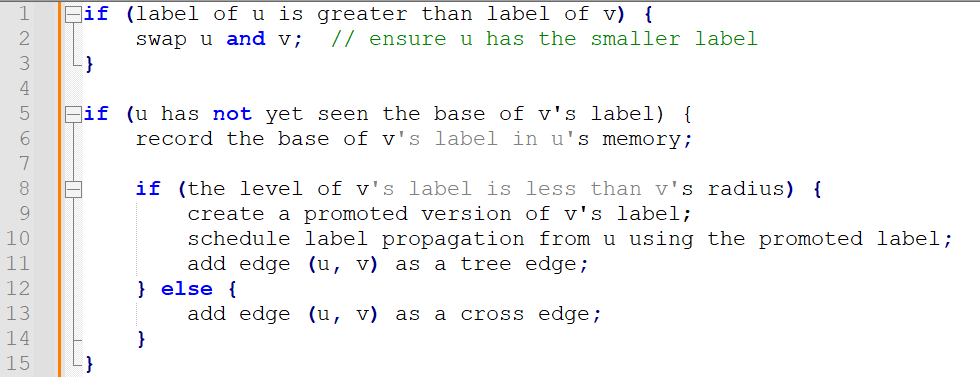
Each vertex independently draws a random radius r(v) from a truncated geometric distribution, which controls how far its label will propagate:

, for every , and

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

1. Streaming Edge Processing & Label Propagation

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



1. 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)

1. 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. The label from A propagates to B and then to C.  
(C,D) doesn’t satisfy the propagation condition so it becomes a cross edge.  
  
The spanner now includes:  
Tree edges: {(A,B), (B,C)}  
Cross edge: (C,D)  
and possibly (D,E), depending on the labels and radius.  
This keeps the spanner sparse while ensuring no shortest path is stretched by more than (2t - 1) hops.

# **Section 3: Implementation of The Algorithm**

This section describes how the algorithm from Elkin’s paper was implemented in Python.

The implementation is organized across multiple files, each with a different responsibility:

* Main.py – Entry point of the program - initializes the graph, assigns radii and labels, processes edges, and constructs the spanner.
* Graph.py – Generates and stores the graph structure using the networkx library.
* Vertex.py – Represents a vertex in the graph, along with its label, radius, edge sets, and memory table.
* Edge.py – Represents an edge between two vertices.
* Label.py – Encodes label behavior, including label promotion and extraction of base and level.
* Spanner.py – Contains the main algorithm logic: radius sampling, label comparisons, and spanner construction using functions readEdge and generateRadiusValue.
* config.py / config.json – Configuration files that define parameters such as graph size, edge probability, and stretch factor.

**Running the project:**

Todo, include requirements n stuff

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. Include 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 before where each vertex independently samples a radius .

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, label promotion happens if the radius allows it and 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 network 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

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.

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: The 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

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.

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 steps 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 spanner vs original.
   * Statistical Summary: average, median, mode, max, and standard deviation of stretch.
4. Output Files  
   Each run generated two outputs:
   * A CSV file summarizing the statistics.
   * 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 is to uncover trends and trade-offs between structural graph parameters and the performance of the streaming spanner algorithm. The following relationships are being 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.