**Mini – Project on Embeddings of Graphs**

Directed by Prof. Michael Elkin.

**Table of contents**

1. Presenting the chosen paper
2. The Algorithm
3. The Implementation of The Algorithm
4. Research question
5. The Experiments
6. The Implementation of The Experiments.
7. The Results of your experiments
8. The conclusion
9. directions for future work.

**Section 1: The Paper**

The algorithm implemented in this project is taken from the paper by Michael Elkin (2011), titled **"Streaming and Fully Dynamic Centralized Algorithms for Constructing and Maintaining Sparse Spanners"**, published in *ACM Transactions on Algorithms*, Vol. 7, No. 2, Article 20, March 2011.

**Full citation:**

*Michael Elkin. Streaming and Fully Dynamic Centralized Algorithms for Constructing and Maintaining Sparse Spanners. ACM Trans. Algor. 7, 2, Article 20 (March 2011), 17 pages.* [*https://doi*](https://doi)*.org/10.1145/1921659.1921666*

**Context and Contribution of the Paper:**

Graph spanners are fundamental structures in algorithmic graph theory, used to approximate distances in large graphs with significantly fewer edges. They are widely applied in areas such as distributed computing, network design, routing, and approximation algorithms.

Prior to this paper, most known algorithms for spanner construction were either too slow for large-scale applications or unsuitable for streaming or dynamic environments. Notably, the streaming algorithm by Feigenbaum et al. (2008) had a high processing cost per edge, making it inefficient in real-time or massive data scenarios.

Michael Elkin's 2011 paper presents a new algorithmic framework for constructing sparse spanners in the streaming model (edges arrive sequentially and cannot be stored in full) and extends it to a fully dynamic centralized setting, where edges can be added and removed incrementally.

**Main Contributions:**

This paper is important because it is the first to provide a streaming algorithm for constructing (2t−1)-spanners with **optimal per-edge processing time** O(1), while also achieving strong guarantees on spanner size and stretch.

It tackles the core challenge of how to construct and maintain a high-quality spanner for an unweighted graph in **streaming** or **fully dynamic** settings environments where edges arrive one-by-one or can be added and removed over time.

The solution uses **minimal memory**, requires only **a single label per vertex**, and avoids the overhead of maintaining complex data structures like clusters or histories of label changes. The result is a surprisingly simple and elegant algorithm that is efficient even for massive graphs, where storing the full edge set is infeasible and dynamic updates are frequent.

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.

**Main Results of the Paper:**

The paper proposes a randomized streaming algorithm that, with high probability, achieves the following:

* **Stretch Guarantee**:  
  The constructed spanner is a **(2t−1)-spanner**. 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).
* **Spanner Size**:  
  With **high probability**, the spanner contains Edges
* **Processing Time Per Edge**:  
  Each edge is processed in **worst-case**  time.
* **Space Complexity**:  
  The algorithm uses bits of **memory**.
* **Deletions** are handled in **expected time**  where m is the number of edges in the input graph.

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

Its elegant design avoids complex structures, showing that sparse spanners can be built and maintained efficiently with minimal state, even in large or evolving graphs.

**Section 2: The Algorithm**

The algorithm implemented in this project is the **streaming spanner construction algorithm**. It constructs a **sparse subgraph** (called 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.

**Key Ideas:**

1. Label Structure

Each vertex v is assigned a label P(v), which encodes two values:  
- A base identifier (initially, the vertex’s own ID)  
- 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: higher level wins; if equal, higher base ID wins.

1. Radius Sampling

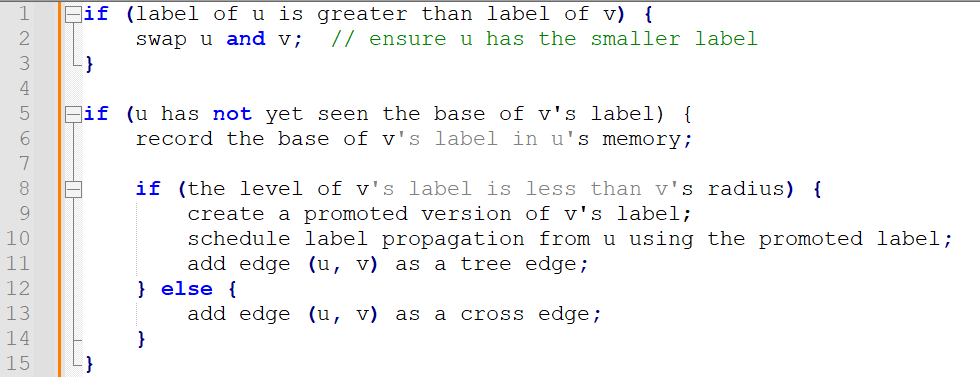
Each vertex independently draws a random radius r(v) from a truncated geometric distribution, which controls how far its label will propagate:

, 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 vs. 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).  
  
Assume:  
- A gets a high-priority label with radius 2  
- Label from A propagates to B and then to C  
- (C,D) doesn’t satisfy the propagation condition → becomes a cross edge  
  
Then the spanner includes:  
- Tree edges: (A,B), (B,C)  
- Cross edge: (C,D)  
- Possibly (D,E), depending on the labels and radius  
  
This keeps the spanner sparse but ensures no shortest path is stretched by more than (2t - 1) hops.

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

This section describes how the algorithm from Elkin’s paper was implemented in Python, with an emphasis on modularity, clarity, and alignment with the theoretical model.

**Project Structure and Files:**

The implementation is organized across multiple files, each with a clear 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 (readEdge, 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

**Representation of Key Elements:**

* **Vertices** (Vertex.py):  
  Each vertex object has:
  + An id as a unique identifier
  + A label (instance of the Label class)
  + A radius value drawn from a geometric distribution
  + Two edge sets: tree and cross edges
  + A table set to track seen label bases (M(v) in the paper)
* **Labels** (Label.py):
  + Stored as an integer: label = level \* n + base
  + Methods exist to promote() a label (increment level) and to extract base and level from integer form.
  + baseVertex links the label to its origin.
* **Edges** (Edge.py):
  + Simple object storing the two vertices it connects: first and second.

**Spanner Construction Flow:**

1. **Graph Generation:**  
   Graph.py uses networkx.erdos\_renyi\_graph() to randomly generate graphs, either weighted or unweighted.
2. **Vertex Initialization:**  
   Each graph node is wrapped with a Vertex object.
3. **Radius Sampling:**  
   Implemented in generateRadiusValue() (in Spanner.py) using a truncated geometric distribution as defined in the paper. Each vertex independently samples a radius
4. **Label Initialization:**  
   Each vertex starts with a label where base is the vertex ID and level is 0.
5. **Edge Processing and Label Propagation:**  
   Implemented in readEdge() (in Spanner.py). For each edge:
   * Vertices are compared by label
   * Label promotion happens if the radius allows it
   * Tree edges T(v) and cross edges X(v) are collected accordingly
6. **Spanner Extraction:**  
   The union of all tree and cross edges from all vertices is added to a new networkx.Graph object as the final spanner.

**Implementation Decisions:**

* **Use of networkx:**  
  Provides efficient graph data structures and algorithms, which simplifies graph generation and visualization.
* **Hash Set for M(v) (Vertex.table)**:  
  Using a Python set() allows constant-time checks and insertions for seen label bases, exactly matching the paper’s goal of minimal state per vertex.
* **Compact Label Encoding:**  
  Label encoding as a single integer simplifies label comparisons and fits well with the lexicographic comparison model in the original algorithm.
* **Separation of Tree and Cross Edges:**  
  Maintains clear alignment with the algorithm’s dual-edge structure, facilitating both label propagation and stretch guarantees.

**Section 4: Research question**

This section explores the empirical behavior of the streaming spanner algorithm under varying graph parameters. The research question is:

**How does the behaviour of the constructed spanner change when the underlying graph structure varies?**

Specifically, this question examines how the distribution of edge stretches in the resulting spanner is affected by two key parameters:

* 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 stretch behaves **on average** or **in distribution** when:

* The graph becomes **larger** (increased number of vertices)
* The graph becomes **denser or sparser** (by changing edge creation probability p)

The underlying goal is to better understand:

* How often edges in the spanner are **stretched** close to the worst-case bound
* Whether the **average stretch remains low** in practice
* 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, this research aims to uncover practical insights into the performance and scalability of the algorithm beyond its theoretical guarantees.

**Section 5: The Experiments**

Fixated 2 play with the third

Create graph and stats and stuff

And conclusions

* Alpha
* Edge prob
* Vertex count

Can also check compression ratio vs stretching factor (max, avg etc) tradeoff