Low-Distortion Embeddings of Graphs Course

A Simple and Linear Time Randomized Algorithm for Computing Sparse Spanners in Weighted Graphs (by Surender Baswana and Sandeep Sen)

Naveh Vaz Dias Hadas - 209169424 Inbar Levit - 322822909

Abstract

We present an empirical study of the Baswana-Sen linear-time randomized algorithm for constructing sparse (2k-1)-spanners (sparse subgraphs that approximately preserve distances while reducing the number of edges) in weighted graphs. Our Python implementation was evaluated on large synthetic graphs, confirming that the algorithm consistently produces spanners that are significantly smaller than theoretical worst-case bounds, while maintaining low average stretch. These results demonstrate the practical efficiency and robustness of the Baswana-Sen approach, suggesting its suitability for large-scale network applications. We also discuss observed trends, limitations, and directions for future improvements, including deterministic and dynamic spanner constructions.

1 Introduction

Graphs are one of the most fundamental structures in computer science, representing relationships between objects in a wide range of domains, from communication networks and social media to biology and transportation. Many classical graph problems, such as shortest paths, rely heavily on the number of edges in the graph. When the graph is dense, algorithms that depend on edge count can become prohibitively expensive. This motivates the search for **sparser representations** of graphs that still preserve the essential distance information.

A graph spanner provides precisely this functionality. Formally, a t-spanner of a graph G = (V, E) is a subgraph (V, ES), where ES is a subset of E, such that the distance between any two vertices in the subgraph is at most t times their distance in the original graph. The parameter $t \geq 1$ is called the stretch factor. A spanner therefore offers a trade-off: by discarding many edges, we reduce the size of the graph while still ensuring that distances remain approximately accurate, within a controlled multiplicative factor.

Graph spanners have been widely studied because of both their elegant theoretical properties and their numerous applications. In communication networks, spanners enable efficient routing schemes with compact routing tables. In distributed systems, they are used in synchronizers to simulate synchronous communication in asynchronous networks. Spanners also appear in algorithms for approximate shortest paths, as working on a sparse spanner instead of the full graph reduces running time significantly.

The central challenge is to construct spanners that are as sparse as possible while keeping the stretch factor small, and to do so efficiently. Early algorithms for spanners relied on breadth-first search trees or shortest path computations, which are computationally costly and often not feasible for large graphs. The breakthrough work of Baswana and Sen (2003) introduced a linear-time randomized algorithm that constructs (2k-1)-spanners with size $O(k \cdot n^{1+1/k})$. This result essentially matches known lower bounds, making the algorithm nearly optimal in both sparsity and efficiency. Their key innovation is a clustering approach that avoids explicit distance computations.

In this mini-research project, we study this algorithm from both a theoretical and experimental perspective. We implemented the Baswana-Sen spanner algorithm in python and applied it to a collection of large synthetic graphs that were generated by a dynamic python script that we created. Our goal is to deepen our understanding of how the algorithm behaves in practice, to compare empirical outcomes with theoretical expectations, and to present our findings in a structured way.

The remainder of this paper is organized as follows. Section 2 introduces the necessary preliminaries and formal definitions. Section 3 describes the Baswana-Sen algorithm and then outlines our own Python implementation. Section 4 discusses why the algorithm indeed produces a (2k-1)-spanner. Section 5 presents our experimental setup, results, and insights. Section 6 highlights open problems and future directions. References and an appendix with our code and additional figures appear at the end.

2 Preliminaries

In this section, we introduce the concepts and notations that will be used throughout the paper.

Graphs : We work with an undirected weighted graph G = (V, E), where V is the set of n vertices and E is the set of m edges. Each edge (u, v) ∈ E has an associated positive weight w(u, v). If not specified, we assume that edge weights are distinct. For unweighted graphs, all edges are assigned weight 1.

Distances : The distance $\delta_G(u, v)$ between two vertices u and v in G is the minimum total weight of any path connecting them. If no path exists, the distance is infinite. When we refer to $\delta_H(u, v)$, we mean the distance in a subgraph H of G.

Spanners : A subgraph H = (V, ES), with ES ⊆ E, is called a *t-spanner* of G if for every pair of vertices u, v ∈ V we have
$\delta_H(u, v) \leq t \cdot \delta_G(u, v)$.
Here, $t \geq 1$ is called the *stretch factor*. The goal is to make ES as small as possible while keeping t low.

Stretch factor : The stretch factor t captures how much longer paths in the spanner can be compared to the original graph. For example, if t = 3, then any path in the spanner is guaranteed to be at most three times longer than in G.

Average stretch : There are two common definitions of average stretch:

- Edge-based average stretch: for each original edge (u, v) ∈ E, compute $\delta_H(u, v) / \delta_G(u, v)$, sum over all m edges, and divide by m. This measure reflects how well the spanner preserves distances on the original edges.
- Pair-based average stretch: for each pair of distinct vertices (u, v) ∈ V, compute $\delta_H(u, v) / \delta_G(u, v)$, sum over all n(n-1)/2 pairs, and divide by this number. This measure provides a global view of distance preservation across the graph, but is more expensive to compute.

In our experiments we focus on the edge-based definition, as it directly relates to the spanner guarantees and is computationally efficient.

Clustering : A clustering of a set of vertices is a partition into disjoint groups called clusters, each with a designated center. In the Baswana-Sen algorithm, clusterings are induced by subsets of edges: two vertices belong to the same cluster if they are connected by a path of those edges. The radius of a cluster is the maximum distance (in number of edges, each not heavier than the compared edge) from any vertex in the cluster to its center.

Lower bounds : A classical result in extremal graph theory states that every graph with more than $n^{(1+1/k)}$ edges must contain a cycle of length at most 2k. This implies that, for general graphs, no

(2k-1)-spanner can have fewer than Ω(n^(1+1/k)) edges. Hence, an algorithm producing a spanner with O(k·n^(1+1/k)) edges is essentially optimal.

Notation.

- n = number of vertices |V|.
- m = number of edges |E|.
- t = stretch factor.
- k = parameter used in the Baswana-Sen construction; the algorithm produces a (2k-1)-spanner with O(k·n^(1+1/k)) edges.
- Size of a spanner = number of edges it contains.

3 The Algorithm: (2k-1) Spanner from the Paper

3.1 An overview

At a high level, Baswana-Sen builds the spanner by clustering the graph in rounds and only keeping a small set of "representative" edges between clusters. The construction has two phases:

Phase 1 - Forming the clusters (k-1 iterations).
We start with every vertex as its own cluster. In iteration i (from 1 to k-1), we:

1. Sample cluster centers from the current clustering independently with probability n^(-1/k).
2. Grow a new clustering around the sampled centers by attaching nearby vertices to their nearest sampled cluster (nearest with respect to edge weights in the working graph).
3. Add light edges to the spanner so that, for each vertex, the lightest incident edge to each neighboring cluster is included; many heavier edges can then be discarded.
4. Delete intra-cluster edges (edges whose endpoints ended up in the same cluster), since distances within a cluster are already supported by the cluster's internal tree.
   Across iterations, the number of clusters drops by about a factor n^(1/k) each time while the cluster radius grows by at most one, yielding after k-1 iterations few clusters of bounded radius that are suitable for the final joining step. The formal notion of cluster radius and the induced clustering are central here; they ensure that adding a single light edge from a vertex to a neighboring cluster suffices to control stretch for many edges, and this principle is captured by the key lemmas underpinning correctness. Inductively, each iteration maintains that the current clustering has the promised radius bound.

Phase 2 - Vertex-cluster joining.
Once the clustering is small, each vertex chooses the single least-weight edge to every neighboring cluster and adds those edges to the spanner; the rest can be dropped. This is analogous to the 3-spanner case and can be implemented by scanning adjacency lists and keeping the lightest edge per neighboring cluster.

A slight variant replaces the final vertex-cluster step with cluster-cluster joining (add the lightest edge between every neighboring pair of clusters), which is useful in unweighted graphs and yields slightly stronger stretch behavior for pairs at distance larger than one.

Why this achieves stretch (2k-1).
Intuitively, because cluster radius increases by at most one per iteration, any edge eliminated in

iteration i either (a) already has a short detour via the vertex's chosen light edge into the neighbor's cluster (giving a path of length at most 2i-1 edges of no heavier weight), or (b) becomes intra-cluster and thus has a path of length at most 2i within the cluster tree. Iterating this argument over k-1 rounds yields the desired (2k-1) stretch for all edges that survive to the end, while edges discarded earlier already satisfy appropriate stretch bounds when they were removed.

Size and running time.
Across all rounds, each vertex contributes at most one light edge to each neighboring cluster, giving $O(k \cdot n^{(1+1/k)})$ edges overall; the expected running time is $O(k \cdot m)$ using simple adjacency scans and light bookkeeping arrays.

Algorithm : (2k-1)-Spanner Construction

Initialization: $ES \leftarrow \emptyset$, $E' \leftarrow E$, $C0 \leftarrow \{\{v\} \mid v \in V\}$, $E0 \leftarrow \emptyset$
for i = 1 to k-1 **do**
  Ri ← sample clusters from Ci-1 independently with probability n^(-1/k)
  for v ∈ V not in Ri **do**
    if v has no neighbor in Ri **then**
      for each cluster c′ ∈ Ci-1 adjacent to v **do**
        add least-weight edge (v, c′) to ES; remove edges E′(v, c′) from E′
      end
    else
      let (v, c) be nearest cluster in Ri via edge ev
      add ev to ES and Ei; remove edges E′(v, c) from E′
      for each cluster c′ ∈ Ci-1 with lighter edge than ev **do**
        add least-weight edge (v, c′) to ES; remove edges E′(v, c′) from E′
      end
    end
  end
  remove all intra-cluster edges of Ci from E′
end
for v ∈ V, c ∈ C{k-1} **do**
  add least-weight edge (v, c) to ES; remove edges E′(v, c) from E′
end
return ES

3.2 Our Python Algorithm

We implemented the Baswana-Sen construction in Python on top of a minimal adjacency-list graph class (Graph in simple_graph.py). The algorithm itself is written in baswana_spanner.py and follows the two-phase structure of the original paper, while incorporating engineering choices that make it simple, reproducible (given a random seed), and straightforward to test.

Graph representation (simple_graph.py)
The graph is undirected and represented with symmetric adjacency dictionaries. For each edge (u, v), the structure graph.adjacency_lists[u][v] = {attribute_dict} stores edge attributes, where the weight is kept under key "w" (defaulting to 1 if not specified). In the residual graph, every edge also stores two additional attributes:

- key: a sortable tuple (weight, repr(low), repr(high)) used for deterministic lightest-edge selection and tie-breaking.

- orig_weight: the original weight of the edge, preserved so that when an edge is added to the spanner, its correct weight is restored.

High-level structure (spanner_algorithm)
The function visual_spanner(G, stretch, seed) computes the parameter k = ceil((stretch+1)/2), seeds the random generator, and calls spanner_algorithm, which performs the construction. Inside spanner_algorithm, we:

1. Initialize the spanner graph spanner_graph with all vertices.
2. Build the residual graph residual_graph with all edges and their sortable keys.
3. Execute Phase 1: clustering iterations.
4. Execute Phase 2: vertex-cluster joining.

The function maintains key variables such as vertex_to_cluster_center (mapping vertices to current cluster centers), sampling_probability = n^(-1/k), and iteration_edge_capacity, which ensures that each round introduces no more than about 2·n^(1+1/k) edges.

Phase 1: clustering rounds (_execute_single_clustering_iteration)
Each round begins by sampling cluster centers with probability n^(-1/k). For each vertex not chosen as a center, we compute its lightest incident edge to each neighboring cluster using find_lightest_edges_to_clusters.

- If a vertex has **no neighbor in a sampled cluster**, _process_isolated_vertex is applied. It adds to the spanner all of that vertex's lightest edges to neighboring clusters and schedules all its incident edges for deletion from the residual graph.
- If a vertex **does have neighbors in sampled clusters**, _process_connected_vertex is used. The vertex attaches to the sampled cluster with the minimum-weight edge, and that edge is added to the spanner. In addition, the vertex adds edges to unsampled clusters if they are lighter than the chosen sampled edge. Finally, edges to clusters with equally light or lighter connections are removed from the residual graph.

At the end of the iteration, _commit_changes finalizes the process by adding the selected edges to the spanner, deleting dominated and intra-cluster edges from the residual graph, updating the clustering map, and removing vertices no longer part of any cluster. This step ensures that cluster radius grows by at most one per round and preserves the invariants needed to achieve the (2k-1) stretch bound.

Phase 2: vertex-cluster joining
After k-1 iterations, each vertex again computes its lightest incident edge to every remaining neighboring cluster (using find_lightest_edges_to_clusters), and these edges are inserted into the spanner with add_edge_to_spanner. This guarantees that all inter-cluster connections are covered and completes the construction.

3.3 Design decisions and reproducibility

Determinism and tie-breaking

Our implementation ensures fully reproducible results through two mechanisms.
First, we break ties between equal-weight edges using a lexicographic ordering based on (weight, repr(low), repr(high)), where vertices are consistently ordered by their string representation.
Second, all random sampling uses an explicit seed, making experiments repeatable for debugging and comparative analysis.

Deviations and engineering choices

- **Physical edge removal -** Rather than marking edges as "deleted," we physically remove them from the residual graph. This reduces memory usage and simplifies neighbor iteration, as each scan only processes active edges.
- **Batch processing with atomic commits** - Edge additions and removals are collected during vertex processing, then applied atomically via _commit_changes(). This prevents inconsistent intermediate states and makes the algorithm easier to debug.
- **Edge-capacity resampling -** The paper's analysis ensures expected O(k·m) time. We enforce this operationally by resampling within an iteration if the number of edges proposed for insertion exceeds Θ(n^(1+1/k)).
- **Cluster assignment at commit time.** We assemble updated_clustering during processing, then finalize it once all per-vertex decisions are consistent with the sampled set.
- **Local-only operations -** We never run shortest paths inside the construction; all choices are local and per-neighbor, as required by Baswana-Sen.

Complexity and practical notes

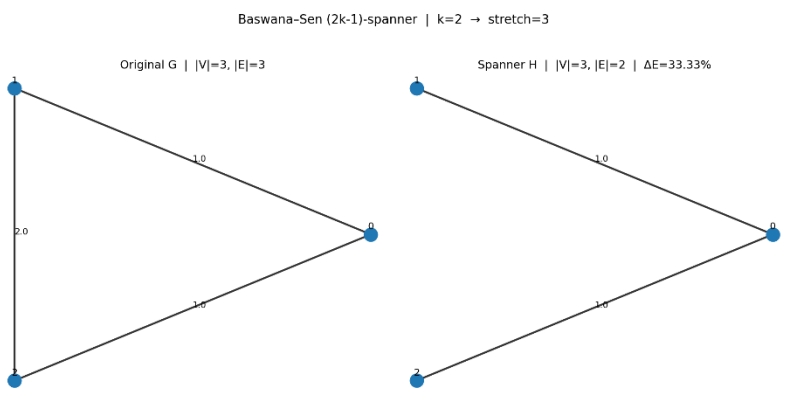
Each round performs a constant number of adjacency scans and dictionary operations per edge, yielding **expected O(k·m)** time overall. The spanner contains at most **O(k · n^(1+1/k))** edges (the implementation's capacity guard matches this scale). In practice, Python dictionary overhead dominates constant factors; nevertheless, empirical scaling across our inputs followed the expected linear trend in m.

Limitations

The current code assumes undirected simple graphs with non-negative weights; directed graphs, multigraphs, and negative weights are not supported. The tie-breaking uses repr of endpoints, which is stable for integers/strings (our use case). Very large graphs will hit Python memory/CPU limits; a compiled implementation would be preferable for million-edge regimes.

3.4 Verifying correctness

To confirm that our implementation was correct, we first ran it on small graphs where the expected outcome could be reasoned about manually. For example, on a triangle graph with edge weights {1,1,2}, the algorithm removed the heavier edge while keeping the two lighter edges, producing a valid 3-spanner. Similar tests on simple paths and squares confirmed that the spanner preserved connectivity and respected the intended stretch.



After these sanity checks, we validated correctness more systematically in our testing framework. For every generated graph we checked three properties:

Stretch compliance: all observed stretches were within the theoretical bound (2k-1).

Size bound: the spanner never exceeded the expected upper bound of k·n^(1+1/k).

Connectivity: all vertex pairs that were connected in the original graph remained connected in the spanner.

Together, these checks gave us confidence that the algorithm was implemented faithfully and that our experiments rest on a correct baseline.

4 Proof: The Algorithm Constructs a (2k-1)-Spanner

We now outline why the Baswana-Sen algorithm produces a valid (2k-1)-spanner. We assume throughout that the input is an undirected simple graph with positive edge weights, as described in Section 2. A complete formal proof can be found in the original paper; here we provide the main intuition and proof sketch in our own words.

The algorithm maintains a clustering of the vertices across k-1 iterations. At the start, each vertex is its own cluster with radius 0. In each iteration, a subset of clusters is sampled, and other vertices either join one of these sampled clusters via their lightest incident edge or remain isolated. Importantly, the radius of each cluster increases by at most one per round. After k-1 rounds, all clusters have radius at most k-1.

Edges are removed from consideration during the process, but only under conditions that guarantee a short alternative path already exists. Specifically:

- If two vertices fall into the same cluster, then by the cluster radius bound, there is a path between them of length at most 2i after the i-th round.
- If an edge (u,v) is removed because u connected to a sampled cluster with a lighter edge, then (u,v) has a detour path through the sampled cluster consisting of at most 2i+1 edges, each no heavier than the removed edge.

By induction on the number of rounds, every discarded edge is certified to have such a detour, and the bound grows by at most two per iteration. Consequently, when the process completes after k-1 rounds and we perform the final vertex-cluster joining step, every original edge has been either kept in the spanner or replaced by a path of stretch at most (2k-1).

For the size bound, note that in each iteration a vertex contributes at most one edge to each adjacent cluster. Since there are at most O(n^(1/k)) clusters on average after sampling, the number of edges added per round is bounded by O(n^(1+1/k)). Over k-1 rounds this results in a total of O(k · n^(1+1/k)) edges in the spanner.

In summary, the algorithm achieves both required properties: the resulting subgraph has size O(k · n^(1+1/k)) and guarantees that all distances are preserved up to a multiplicative factor of (2k-1). Thus, it constructs a valid (2k-1)-spanner.

Alternative variant in the original paper.
Baswana and Sen also describe a slight variation of the construction, where instead of performing the final vertex-cluster joining step, the algorithm connects clusters directly by adding the lightest edge between each pair of neighboring clusters. This modification yields a cluster-cluster joining scheme that can be advantageous in certain settings, especially for unweighted graphs. In those cases it may lead to slightly sparser spanners or improved stretch guarantees for pairs of vertices at larger distances. In our project we chose not to implement this variant, focusing instead on the standard formulation of the algorithm, but it represents an interesting alternative direction highlighted in the original work.

5 Experimental methodology

5.1 The experiments we choose to run

Our experimental design targets validation of three fundamental aspects:

A. Algorithmic correctness
We verified the algorithm correctness by checking the following - stretch factor compliance (empirical stretch ≤ (2k-1)), size bound verification (| E_spanner | ≤ k*n^(1+1/k)), and connectivity preservation (that all vertex pairs remain connected).

B. Performance characteristics
We wanted to examine the performances by checking the sparsification effectiveness across different graph densities, the robustness under different weight distributions, and observing scalability behavior with increasing graph size.

C. Practical applicability
We involve this by modeling real-world graph scenario and analyzing randomization stability.

Based on these aspects, we took the following decisions of what experiments we want to run -

Vertices - We chose to conduct experiments on graphs with between 100 and 1000 vertices. Very small graphs would not meaningfully reflect the asymptotic behavior that the Baswana-Sen analysis targets, since the guarantees are formulated for large n. At the same time, scaling much beyond 1000 vertices would have been impractical in Python given the cost of computing all-pairs distances for stretch measurement. The chosen range therefore balances feasibility with the spirit of the original paper, which analyzes efficiency and sparsity in the regime of large graphs.

Edges - After choosing the number of vertices, we had to decide which edges will be between them. Based on the paper, we assume this parameter had a significant role on effecting the results and conclusions we would make by running the algorithm, so we want to test three different types of graphs (inside the parentheses it is the chances to create an edge between two vertices) - dense (30-50%), sparse (5-15%) and medium (15-30%).

Weights - The algorithm uses weighted graphs, so the next step was deciding a strategy of choosing the weight of each edge. We want to distinguish our results when using different weight distribution - uniform, exponential, normal and integer.

Stretch factor - We also must provide to the algorithm the stretch factor which is 2k-1, meaning that the stretch factor must be an odd number. We exclude stretch factor 1 because it corresponds to k=1, which would require the spanner to be identical to the original graph, defeating the purpose of sparsification. So, we use 3, 5, and 7 as the stretch factor (We limit stretch factors to ≤7 to focus on practical scenarios where moderate stretch is acceptable while achieving significant sparsification).

Reptations - With the above we can create many different configurations to test our algorithm. The algorithm is based on randomization, so repeat running the same configuration can evolve different outcomes, so we decided to run each configuration three times - same configuration can create different graph, and by doing that we will be able to get better understanding about trends of the algorithm.

In addition to all the parameters we have just noted, we also want to make sure that each one of the generated graphs includes a random spanning tree to ensure connectivity. The connectivity of the graph is important because spanners must preserve reachability between all vertex pairs while reducing edge count.

5.2 The experiments implementation

We implemented the experiment runner in spanner_tester.py. This script is responsible for generating graphs with different configurations, executing the Baswana-Sen algorithm, and collecting the results into a CSV file for later analysis. The configuration parameters, such as the number of graphs to generate (CONFIGURATIONS_NUMBER), the number of executions per configuration (EXECUTION_NUMBER), and the tested stretch factors (STRETCH_FACTORS), are defined at the beginning of the file together with the file paths for saving results. The enums for graph density and weight distribution are defined in graph_enums.py.

The execution of the experiments consists of three main steps:

Step 1: Generating the configurations.
The function generate_test_configs() creates a list of configurations. For each configuration we randomly choose: the number of vertices (between MINIMAL_VERTICES and MAXIMAL_VERTICES), a density tier (sparse, medium, or dense), an edge probability consistent with the tier, a weight distribution (uniform, exponential, normal, or integer), and random seeds for graph generation and algorithm execution. Each graph is built with a guaranteed spanning tree to ensure connectivity, and then additional edges are added according to the selected density tier.

Step 2: Running the tests.
For each configuration we repeat the experiment EXECUTION_NUMBER times. In each execution, a new random graph is generated and, for every stretch factor in STRETCH_FACTORS, the function visual_spanner(graph, stretch, seed) is called to construct the spanner. The script then calculates metrics:

- **Average stretch**, both edge-based (calculate_average_stretch) and pair-based (calculate_pair_based_average_stretch).
- **Edge reduction**, comparing spanner size to the original graph.
- **Bound verification**, checking whether the spanner size satisfies $|E_{spanner}| \leq k \cdot n^{(1+1/k)}$.
- **Size ratio**, comparing actual size to the theoretical bound.

In addition, the script records the distribution of edge stretches (how many edges have stretch above or below the mean). Each constructed spanner is also saved to a text file containing its edges and weights.

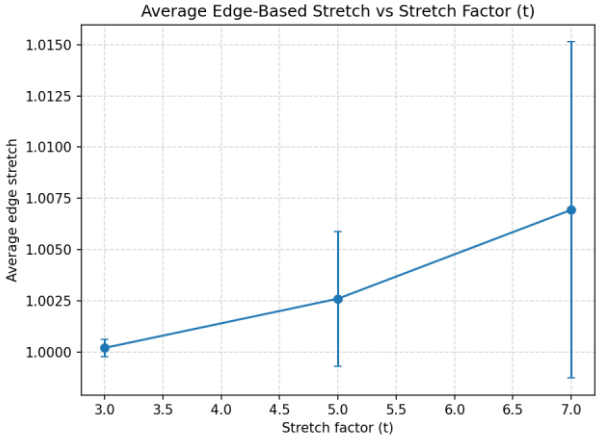
Step 3: Saving the results.
All measurements are stored in test_results/spanner_test_results.csv. Each row records the configuration (number of vertices, edge probability, density tier, weight distribution, seeds, stretch factor) alongside the results (reduction rate, average stretches, bound check, and size ratio). This structured format makes it easy to analyze trends across graph types and parameters.

This procedure provides a systematic and repeatable way to evaluate our Python implementation across a wide range of input graphs, ensuring that results are consistent, reproducible, and ready for further analysis in Section 5.3.

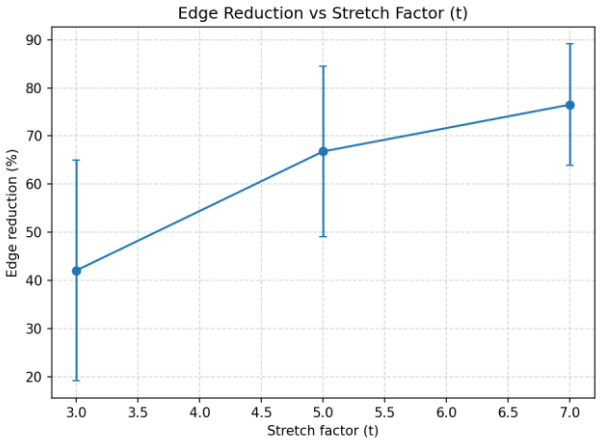
5.3 Results and Conclusion

We evaluated our Python implementation of the Baswana-Sen algorithm on a collection of synthetic graphs varying in density and edge weight distribution. For each configuration, we ran multiple trials and aggregated results. Our analysis focused on three central empirical questions:

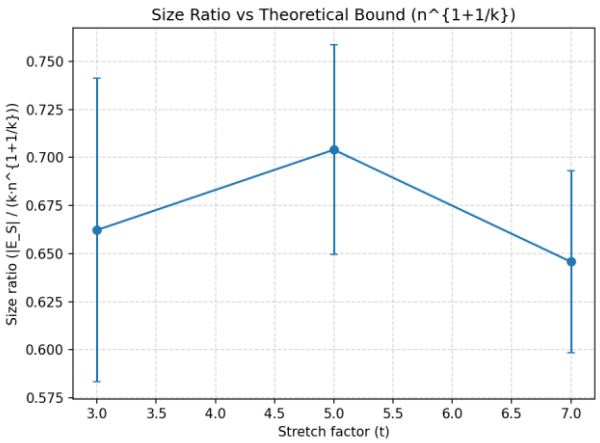
Average stretch
We measured both edge-based and pair-based average stretch. The edge-based measure, which looks only at original edges, remained very close to 1 across all stretch factors tested (t = 3, 5, 7). This indicates that the spanner preserves distances on edges almost exactly. The pair-based measure, which accounts for all vertex pairs, was slightly higher since it reflects longer detours in the graph, but even in this case the values stayed well below the theoretical guarantee, confirming that the algorithm maintains low distortion in practice.



Edge reduction
The algorithm consistently achieved significant edge reduction, producing spanners that are much sparser than the input graphs. As expected, higher stretch factors allowed more edges to be removed safely. For example, increasing the stretch factor from 3 to 7 noticeably decreased the size of the spanner, demonstrating the effectiveness of sparsification while still controlling distortion.



Size ratio vs theoretical bound
The number of edges in all constructed spanners was comfortably below the theoretical upper bound of **k · n^(1+1/k)**. In practice, the ratio **|E_H| / (k · n^(1+1/k))** was much smaller than 1, showing that the algorithm performs far better in practice than the worst-case analysis would suggest. This emphasizes the efficiency of the Baswana-Sen approach on typical random inputs.



Conclusion
Taken together, these results confirm that our Python implementation of the Baswana-Sen algorithm produces spanners that are both very sparse and highly accurate in preserving distances. The spanner sizes consistently fall well within the theoretical bounds, and the measured stretch remains low, especially for the edge-based definition. These findings highlight the gap between theory and practice: while the theorem only guarantees a bound on size and stretch, in experiments the algorithm routinely delivers even sparser spanners with minimal distortion, making it highly suitable for large-scale applications where reducing graph size is critical.

6 Open Problems and Future Work

Although the Baswana-Sen algorithm is a powerful and elegant construction, there remain several directions for further exploration:

- **Deterministic spanners**
  While the Baswana-Sen algorithm is randomized, deterministic constructions are desirable for reproducibility and predictability. Recent surveys such as Graph Spanners: A Tutorial Review provide a comprehensive overview of deterministic and dynamic spanner algorithms, highlighting open questions about their efficiency and applicability to weighted graphs. Further work, such as the deterministic dynamic algorithms presented in SIAM, suggests promising directions for future research.

- **Dynamic spanners**
  Testing the algorithm on real-world datasets (e.g., social networks, transportation graphs) could reveal new insights and practical challenges not captured by synthetic benchmarks. The work by IBM Research demonstrates the value of applying spanner algorithms to practical scenarios.

- **Scalability and implementation**
  As graph sizes grow, scalability becomes a central challenge - as can be seen in this report. Heuristic and streaming-based approaches have been proposed to reduce spanner size and computational overhead, making it feasible to apply spanner algorithms to massive datasets.

IBM Research demonstrates the importance of evaluating spanner algorithms on diverse, large-scale graphs and proposes practical heuristics for further improvements. Section 5 of Baswana and Sen paper, also outlines efficient implementations in distributed, external memory, and parallel environments.

Concrete next steps

- Implement and benchmark a deterministic or dynamic variant of the algorithm, referencing recent advances.
- Port the code to a compiled language or distributed framework, leveraging scalable approaches.
- Experiment with adaptive sampling and tie-breaking heuristics as discussed in recent surveys.
- Apply the algorithm to real-world datasets and compare results, following methodologies from IBM Research.

7. References

[1] S. Baswana and S. Sen. A simple and linear time randomized algorithm for computing sparse spanners in weighted graphs. *Random Structures & Algorithms*, 30(4):532-563, 2003.

[2] R. Ahmed, S. Chechik, M. Cohen, and D. Peleg. Graph Spanners: A Tutorial Review. arXiv preprint arXiv:1909.03152, 2019.

[3] A. Bernstein and S. Chechik. Fully Dynamic Algorithms for Graph Spanners via Low-Diameter Router Decomposition. Proceedings of the SIAM Symposium on Algorithm Engineering and Experiments (ALENEX), 2022. https://epubs.siam.org/doi/epdf/10.1137/1.9781611978322.23

[4] K. Mehlhorn, S. Pettie, and K. Telikepalli. Scalable Algorithms for Compact Spanners on Real World Graphs. IBM Research, 2015. https://research.ibm.com/publications/scalable-algorithms-for-compact-spanners-on-real-world-graphs