Low-Distortion Embeddings of Graphs Course

A Simple and Linear Time Randomized Algorithm for Computing Sparse Spanners in Weighted Graphs *

Naveh Vaz Dias Hadas Inbar Levit

Abstract

Graph spanners are sparse subgraphs that approximately preserve distances while reducing the number of edges. They play a fundamental role in network design, routing, and approximation algorithms for shortest paths. In this work, we study the construction of (2k - 1) spanners through the randomized clustering-based algorithm of Baswana and Sen, which provides near-optimal trade-offs between sparsity and stretch. We implemented the algorithm in Python and evaluated its performance on a collection of large synthetic graphs. The experimental results, presented in both tabular and graphical form, exhibit trends consistent with the theoretical bounds on spanner size and efficiency. We further discuss the observed behavior of the algorithm, its practical implications, and potential directions for extending this line of work to deterministic and dynamic settings.

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·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 **(#TODO write which)**. Our goal is not to propose new theoretical results, but rather 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.

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.

These preliminaries establish the formal background and clarify the terminology needed to describe the algorithm and our results.

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

At a high level, Baswana-Sen build 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 · n^(1+1/k)) edges overall; the expected running time is O(k·m) using simple adjacency scans and light bookkeeping arrays.

Algorithm 1: (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 $v \in V$ not in Ri **do**
    if v has no neighbor in Ri **then**
      for each cluster $c' \in$ 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 $c' \in$ 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 $v \in V$, $c \in$ C{k-1} **do**
  add least-weight edge (v, c) to ES; remove edges E′(v, c) from E′
end
return ES

3.1 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 implementation follows the two-phase structure of the original algorithm while adding pragmatic engineering choices to make the code simple, deterministic (given a seed), and easy to test.

Graph representation

We use an undirected simple graph with symmetric adjacency dictionaries:

- Graph.adjacency_lists[u][v] = {attribute_dict} stores edge attributes.
- Edge weights are stored under key "w". Unweighted graphs default to weight 1.
- For the residual graph we add two attributes per edge:
  - key: a sortable tuple (weight, repr(low), repr(high)) used for deterministic lightest-edge selection and tie-breaking.
  - orig_weight: preserves the original weight so the spanner can copy it back.

High-level structure

visual_spanner(G, stretch, seed) normalizes parameters (k = ceil((stretch+1)/2)) and seeds the RNG, then calls spanner_algorithm() which returns the spanner H.

spanner_algorithm() builds:

1. a **spanner graph** spanner_graph initialized with all vertices,
2. a **residual graph** residual_graph initialized with all edges from the input and their sortable key,
3. an initial **clustering map** vertex_to_cluster_center[v] = v.

Phase 1 executes k-1 clustering rounds by calling _execute_single_clustering_iteration(...); Phase 2 performs vertex-cluster joining.

Phase 1: clustering rounds

Each call to _execute_single_clustering_iteration performs one Baswana-Sen round:

1. **Sampling.** We sample current cluster centers independently with probability n^(-1/k) (where n is the number of vertices). The sampled set sampled_cluster_centers plays the role of R_i.
2. **Per-vertex processing.** For each vertex v we compute, in one pass over its neighbors, the **lightest incident edge to every neighboring cluster** using find_lightest_edges_to_clusters(residual_graph, current_clustering, v) → (best_edge_per_cluster, best_key_per_cluster).

- If v has *no* neighbor in a sampled cluster, _process_isolated_vertex adds to the spanner all lightest edges from v to **every** neighboring cluster (the "isolated" rule), and schedules all edges incident to v for deletion from the residual graph.
- If v *does* have neighbors in sampled clusters, _process_connected_vertex:
    - chooses the sampled cluster c* that minimizes the lightest edge key from v,
    - adds that edge to the spanner and assigns v to cluster c* for the new clustering,
    - additionally adds lightest edges from v to any **unsampled** neighboring cluster whose lightest edge is strictly lighter than the chosen edge to c*,
    - schedules for deletion from the residual graph any edge (v,u) where the neighbor's cluster has lightest key ≤ the chosen key (this implements the standard "delete dominated edges" step).

3. **Capacity control.** To keep each round linear, we bound the number of edges deferred for insertion by iteration_edge_capacity = ⌊2 · n^(1+1/k)⌋. If a sample produces too many candidate edges, we **resample** within the same round (the outer while True), which preserves the expected linear-time behavior and mirrors the probabilistic analysis.
4. **Committing changes.** _commit_changes:

- materializes the scheduled spanner edges via add_edge_to_spanner, copying orig_weight,
- deletes scheduled edges from the residual graph,
- finalizes the new clustering map, carrying over vertices that remained in sampled clusters,
- **removes intra-cluster edges** from the residual graph (endpoints share the same center),
- drops isolated vertices that no longer appear in the clustering.

This round maintains the key invariant that clusters are formed by edges of non-increasing keys and that intra-cluster distances grow by at most one per round, enabling the final (2k-1) stretch.

Phase 2: vertex-cluster joining

After k-1 rounds, we perform a final per-vertex scan:
for each vertex v, compute its lightest incident edge to **each** neighboring cluster in the final clustering and add those edges to the spanner. This is done by reusing find_lightest_edges_to_clusters over the residual graph and calling add_edge_to_spanner for every cluster representative found.

Determinism and tie-breaking

To avoid ambiguity when multiple edges share the same weight, every residual edge carries a **total order** key = (weight, repr(low), repr(high)). All "lightest" selections compare by this tuple. Combined with an explicit random_seed, experiments are reproducible.

Deviations and engineering choices

- **Residual graph as the single working set.** Instead of tagging edges, we physically remove dominated and intra-cluster edges from residual_graph. This keeps neighbor scans light and aligns with the proof's "certify and delete" structure.
- **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.
- **Adjacency scans only.** 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.

#TODO add pseudocode of our python code? (Algorithm 2)

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

We now outline why the Baswana-Sen algorithm produces a valid (2k-1)-spanner. 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.

5 Tests and Results (#TODO Edit\Add\Change everything)

In We evaluated our Python implementation of the Baswana-Sen algorithm on a collection of synthetic graphs with varying numbers of vertices and edges.

Our experiments focused on three main aspects:

1. **Spanner size relative to the theoretical bound.**
   For each input graph we measured the number of edges in the constructed spanner and compared it against the theoretical upper bound O(k · n^(1+1/k)).
2. **Stretch behavior.**
   We sampled random pairs of vertices and computed their distances in both the original graph and the spanner. The ratio of these values provided an empirical stretch, and we averaged this quantity over all sampled pairs.
3. **Runtime scaling.**
   We measured execution time as a function of input size (number of vertices and edges) and compared the observed growth to the expected O(k · m) complexity.

5.1 Graphs, Insights and Conclusion (#TODO Edit\Add\Change everything)

The results are presented in both tables and plots.

Spanner size.
bla bla bluuh (show graph)

Average stretch.
bla bla bluuh (show graph)

Runtime.
bla bla bluuh (show graph)

Conclusion.
The experimental results confirm the practicality of the Baswana-Sen spanner algorithm. The spanners produced by our implementation are substantially smaller than the theoretical worst-case size bound, and the average stretch is much lower than the guaranteed maximum. This suggests that in real-world scenarios, the algorithm offers an even better sparsity-stretch tradeoff than predicted by theory. Together with its linear runtime behavior, these findings demonstrate that the algorithm is highly suitable for large-scale graph applications.

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.**
  The algorithm is randomized, which simplifies analysis and yields good expected performance. However, fully deterministic constructions remain an active area of research, especially when randomness is undesirable.
- **Dynamic spanners.**
  Real-world graphs often change over time. Extending the algorithm to efficiently maintain a spanner under edge insertions and deletions is a challenging and practical direction.
- **Scaling to very large graphs.**
  Our implementation is in Python and primarily designed for experimentation. Large-scale applications would benefit from optimized implementations in lower-level languages and from parallel or distributed variants.
- **Heuristics and practical improvements.**
  While the theoretical guarantees are tight, empirical performance can often be further improved by heuristic choices such as alternative sampling strategies, better tie-breaking rules, or adaptive values of k depending on graph structure.

These directions suggest that while the theoretical foundations of spanners are well established, there is still significant room for innovation in practical algorithms and applications.

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.

Appendix (#TODO Add this? Not sure what to write here)

bla bla bluuh