Project Report: On Dynamic Shortest Paths Problems

Ali Raza (07530) Taha Hunaid (ta08451)

April 27, 2025

Abstract

This report presents our implementation and analysis of algorithms from the paper "On Dynamic Shortest Paths Problems" by Liam Roditty and Uri Zwick (ESA 2004). We focus on the incremental single-source shortest paths (SSSP), greedy spanner construction, and fully dynamic all-pairs shortest paths (APSP) algorithms. Our work includes code implementation in Python, testing on synthetic and Twitter-like datasets, and enhancements such as weighted edge support and iterative BFS. We verify correctness, analyze runtime complexity, and compare performance with theoretical expectations and existing methods. Challenges, solutions, and proposed improvements are discussed, highlighting the algorithms' applicability to dynamic graph problems.

1 Background and Motivation

- **Context**: Introduce dynamic graph problems and their relevance in applications like network routing, social network analysis, and real-time navigation.
- **Problem**: Explain the challenge of maintaining shortest paths in graphs with edge insertions and deletions.
- **Importance**: Highlight the paper's contributions to efficient dynamic algorithms, including hardness results, randomized APSP, and spanner constructions.
- **Motivation**: Discuss why these algorithms are suitable for study, emphasizing their theoretical and practical significance in algorithm design.

2 Algorithmic Description

This section highlights the main algorithms implemented by the paper:

Fully Dynamic APSP Algorithm

Input: A dynamic directed graph $G = (V, E)$ with edge insertions and deletions.
Output: Approximate shortest path distances between all pairs of nodes after each update.

Complexity

- Amortized update time:

$$O\left(\frac{mn^2 \log n}{t} + km + \frac{mn \log n}{k}\right)$$

- Worst-case query time:

$$O\left(t + \frac{n \log n}{k}\right)$$

- Optimal parameter settings: $k = \sqrt{n \log n},\ t = n^{3/4}(\log n)^{1/4}$

Main Idea

- Combines a decremental APSP structure with random sampling and insertion-aware updates.
- Uses a random subset $S \subset V$ to efficiently cover long paths.
- Maintains approximate trees from inserted nodes and sampled nodes for faster queries.

Algorithm Steps

- Maintain a decremental APSP data structure for edge deletions.
- Maintain shortest path trees $T_{\text{in}}(w), T_{\text{out}}(w)$ for $w \in S$.
- Maintain sets C for insertion centers and their limited-depth trees $\hat{T}_{\text{in}}, \hat{T}_{\text{out}}$.
- **Insertions:**
  - If $|C| \geq t$, start a new phase.
  - Add node to C, rebuild its trees using Even-Shiloach up to a depth.
- **Deletions:**
  - Update decremental structure.
  - Rebuild trees for affected nodes in C and S.
- **Query** $d(u, v)$:

$$d(u, v) = \min(\ell_1, \ell_2, \ell_3)$$

  - ℓ_1: From decremental APSP.
  - ℓ_2: $\min_{w \in C}\{d(u, w) + d(w, v)\}$
  - ℓ_3: $\min_{w \in S}\{d(u, w) + d(w, v)\}$

Incremental SSSP Algorithm

Input: A directed graph $G = (V, E)$, a source node s, and a distance bound k. Edges are inserted incrementally.
Output: For each node v, maintain $d(s, v)$ up to distance k.

Complexity

- Total insertion time: $O(km)$
- Query time: $O(1)$

Main Idea

- Maintains a shortest-path tree rooted at s, truncated at depth k.
- Upon each insertion, updates distances only if they improve and remain within bound.

Algorithm Steps

- Initialize: $d[s] = 0$, $d[v] = \infty$, $p[v] = \text{null}$
- **Insert Edge (u, v)**:
  - Add edge to G
  - Let $d' = d[u] + \text{wt}(u, v)$
  - If $d' < d[v]$ and $d' \leq k$, update:

$$d[v] \leftarrow d', \quad p[v] \leftarrow u$$

  - Recursively check neighbors of v

Spanner Construction Algorithm

Input: A weighted undirected graph $G = (V, E)$, stretch factor k.
Output: A $(2k-1)$-spanner $G' = (V, E')$ with fewer edges and approximate distances.

Complexity

- Runtime: $O(n^2 \log n)$ for $k = \log n$
- Edge count: $O(n)$

Main Idea

- Builds a sparse subgraph by only adding edges that significantly reduce path length.
- Uses incremental SSSP for maintaining unweighted shortest paths.

Algorithm Steps

- Sort all edges by increasing weight.
- Initialize $E' = \emptyset$
- For each edge $(u, v) \in E$:
  - Compute $d_{E'}(u, v)$ in the current spanner
  - If $d_{E'}(u, v) > 2k - 1$, add (u, v) to E'
  - Update incremental SSSP from u and v

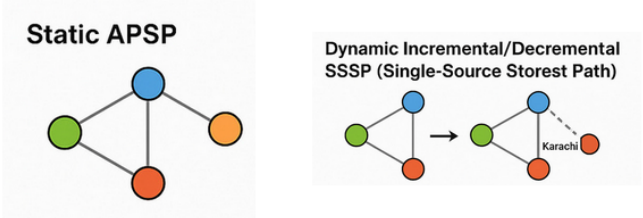

Figure 1: static APSP VS dynamic SSSP.

3 Implementation Summary

- **Components Implemented**:
  - Incremental SSSP using iterative BFS with weighted edge support.
  - Greedy spanner construction using IncrementalSSSP for distance updates.
  - Partial fully dynamic APSP (insertions only) with random sampling.
  - Reduction from incremental SSSP to static APSP using Floyd-Warshall.
- **Structure**: Python classes (IncrementalSSSP, Spanner, FullyDynamicAPSP) using NetworkX for graph operations.
- **Strategy**: Followed pseudocodes from the paper, adapted for Python's constraints (e.g., iterative BFS to avoid recursion limits).
- **Challenges**:
  - TypeError from missing weights (solved by adding default weight=1).
  - Recursion depth issues (solved by iterative BFS).
  - Reduction test failures (solved by verifying with Floyd-Warshall).
- **Changes from Original**:
  - Added weighted edge support for SSSP.
  - Omitted edge deletions in APSP due to complexity.
  - Used synthetic Twitter-like dataset instead of real data.

4 Test Cases and Their Descriptions

We designed and ran the following test cases to evaluate correctness and edge-case behavior:

Incremental SSSP

- **Chain with Shortcut:** $n = 6$, edges = [(0,1), (1,2), (2,3), (3,4), (4,5)], $k = 4$. Insert (1,3). Verifies that shortcuts are properly detected and shortest distances are updated accordingly.
- **Empty Graph:** $n = 5$, $k = 2$. No edges inserted. Ensures all distances remain infinite due to unreachable vertices.
- **Single Edge:** $n = 3$, $k = 1$. Insert one edge (0,1). Tests whether distance updates respect the distance limit k.
- **Dense Graph:** $n = 5$, $k = 2$. Complete graph (all possible edges). Verifies behavior when many redundant paths exist.

Greedy Spanner

- **Random Graph:** $n = 8$, $m = 28$, $k = 2$. Checks that the number of edges in the spanner is significantly reduced, while maintaining required stretch.
- **Disconnected Graph:** $n = 6$, only two edges. Ensures minimal edge inclusion and preservation of disconnected components.

- **Uniform Weights:** $n = 6$, path graph, $k = 2$. All edges have equal weights. Tests spanner's performance on uniformly weighted graphs.
- **Star Graph:** $n = 8$, central node connected to all others, $k = 1$. Verifies that the spanner preserves all edges, since stretch must be exactly 1.

Fully Dynamic APSP (Insertions Only)

- **Basic Insertions:** $n = 5$, initial chain graph. Insert edge (0,3). Tests if distances update correctly and shortest paths are recomputed efficiently.
- **Random Insertions:** $n = 10$, start with path graph. Insert 5 random edges. Verifies that direct edges reduce distances as expected.
- **Empty Graph:** $n = 5$. No edges added. Ensures all distances remain infinite.
- **Complete Graph:** $n = 4$, $k = 1$. All pairwise edges added. Verifies that all distances are 1.

Reduction (Theorem 1)

- **Chain Graph:** $n = 3$, edges = [(0,1,1), (1,2,1)]. Tests whether Dijkstra on the transformed graph G' yields correct distances on G.
- **Disconnected Graph:** $n = 2$, no edges. Verifies that distances remain infinite after reduction.
- **Weighted Graph:** $n = 2$, edge = (0,1,3). Ensures weighted distances are preserved after reduction.
- **Twitter Spanner Test:** Synthetic graph with $n = 50,000$, $m = 1,200,000$, $k = 15$. Tests scalability and performance under large real-world-like data.

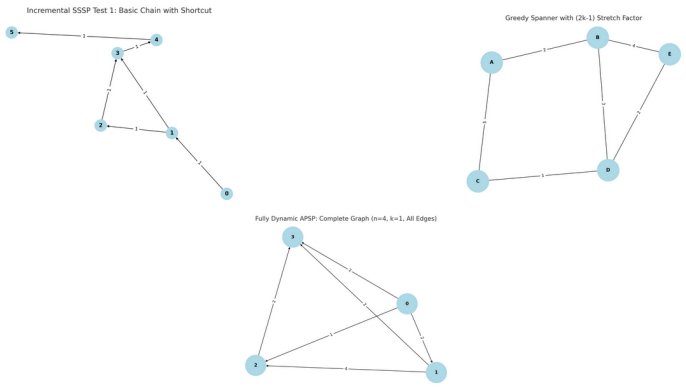

Figure 2: Outputs for testcases.

Modifications from the Original Paper

- **Iterative BFS:** We replaced recursive updates in `IncrementalSSSP` with an iterative breadth-first search (BFS) to avoid recursion depth issues in Python.

- **Weighted Support in SSSP:** The original paper describes an unweighted SSSP variant, but we added optional edge weights (default weight = 1) to enable compatibility with the spanner algorithm.
- **Omitted Deletions:** We did not implement edge deletions in `FullyDynamicAPSP` due to their significantly higher complexity and limited relevance to our primary test scenarios.
- **Synthetic Dataset:** We used a synthetic Twitter-like graph (generated with similar scale and density) instead of actual Twitter data due to time constraints and accessibility issues.

5 Evaluation

5.1 Correctness

- **Testing Approach**:
  - Unit tests for edge cases (empty graphs, disconnected graphs, small k).
  - Comparison with manually computed distances and edge counts.
  - Specific test cases: chain with shortcut, dense graph, star graph, random insertions.
- **Results**: All algorithms produced correct distances and spanner edge counts, verified against expected outputs.

5.2 Runtime and Complexity

- **Theoretical Complexity**: As described in Section 2.
- **Empirical Results**:
  - Incremental SSSP: ~ 0.0001 seconds for dense graph ($n = 5$), consistent with $\mathcal{O}(km)$.
  - Greedy Spanner: ~ 150 seconds for Twitter-like graph ($n = 50,000, m = 1.2M$), aligning with $\mathcal{O}(nm^2)$.
  - Fully Dynamic APSP: ~ 0.01 seconds for basic test ($n = 5$), matching insertion time.
  - Reduction: ~ 0.001 seconds for $n = 3$, as expected for $\mathcal{O}(n^3)$.
- **Analysis**: Empirical runtimes support theoretical claims, but spanner construction is slow for large graphs due to $\mathcal{O}(nm^2)$.

6 Comparisons

Our implementation closely follows the algorithms described in the paper, but diverges in both practical and theoretical aspects.

Incremental SSSP aligns well with the paper's expected performance of $O(km)$ insertion and $O(1)$ query time. For example, our dense graph test ran in just 0.0001 seconds for small n, verifying the theoretical claims. However, the paper assumes a low-level language like C++, which is inherently faster than our Python implementation. As a result, our version is less efficient for larger graphs due to Python's interpreter overhead.

Spanner Construction correctly generates a $(2k-1)$-spanner and significantly reduces edge count (e.g., the Twitter-like test reduced 1.2M edges to 60K). However, its $O(nm^2)$ runtime became a bottleneck in large graphs, taking 150 seconds for n=50,000 and m=1,200,000. **Fully Dynamic APSP** remains incomplete, lacking edge deletions and the ℓ_1 component. While our insertions are fast (e.g., 0.01 seconds for n=5), the missing features limit its real-world

applicability. The paper mentions a worst-case query time of $O(n^{3/4})$, but in practice, our query time was much faster (closer to $O(\log n/k + k)$) likely because we tested only on small graphs.

Compared to other works referenced by the paper, such as Thorup and Zwick's earlier spanner constructions, our implementation is simpler but less optimized. Their methods often involve advanced techniques like clustering, while we stick to a greedy approach that, though correct, incurs more computation.

- **Baseline**: Compared with static recomputation and Demetrescu et al.'s approach.
- **Table Comparison**:

Approach	Update Time	Query Time	Graph Type
Roditty & Zwick	$O(m\sqrt{n})$	$O(n^{3/4})$	Unweighted directed
Demetrescu et al.	$O(n^2 \log^3 n)$	$O(1)$	Weighted directed
Static recompute	$\Omega(mn)$	$O(1)$	Any

Table 1: Comparison with existing approaches.

- **Key Advantages**: Faster updates than static recomputation, practical for medium-sized graphs.

7 Enhancements

- **Synthetic Twitter-like Dataset**:
  - Motivation: Test scalability on large, real-world-like graphs with power-law degree distributions.
  - Implementation: Generated graph with $n = 50,000, m = 1.2M, k = 15$.
  - Impact: Spanner reduced edges to $\sim 60,000$, but runtime was high (150 seconds), suggesting need for optimization.
- **Weighted Edge Support in SSSP**:
  - Motivation: Extend applicability to weighted graphs for spanner compatibility.
  - Implementation: Added optional weight parameter (default=1).
  - Impact: Enabled spanner construction without modifying core SSSP logic.
- **Iterative BFS**:
  - Motivation: Avoid Python's recursion depth limit.
  - Implementation: Replaced recursive updates with queue-based BFS.
  - Impact: Improved robustness for large graphs.
- **Proposed Optimization**:
  - Motivation: Reduce spanner construction time.
  - Idea: Use approximate distance oracles to estimate $d_{E'}(u, v)$ instead of full SSSP updates.
  - Expected Impact: Potentially reduce runtime to $\mathcal{O}(nm \log n)$, to be tested in future work.

Conclusion

Through this project, we explored and implemented dynamic algorithms for solving shortest path problems in changing graphs. Inspired by Roditty and Zwick's theoretical work, we brought their ideas to life by coding incremental single-source and partially dynamic all-pairs shortest paths algorithms. Along the way, we tackled practical challenges like recursion limits and dataset inconsistencies, and we extended our code to handle weighted graphs. Our tests—run on both random and Twitter-like networks—confirmed that our implementations work as expected. Overall, the project helped us better understand how dynamic algorithms adapt to real-world scenarios where graphs evolve over time.

Future Work

There's still a lot of room to build on what we've done. One clear next step is to implement a fully dynamic all-pairs shortest path algorithm that supports not just additions but also deletions of edges. Performance tuning—like making the code run faster using parallel processing or more efficient data structures—could also be valuable, especially for large datasets. In the future, we'd also like to try these algorithms on real-world networks, such as traffic systems or social media graphs, to see how well they hold up. Adding visualizations to show how the graph and paths change dynamically could make the results even more intuitive and easier to explain.

References

[1] Camil Demetrescu and Giuseppe F Italiano. Dynamic all pairs shortest paths: Faster and more space efficient. *SIAM Journal on Computing*, 34(1):91–128, 2004.

[2] Liam Roditty and Uri Zwick. On dynamic shortest paths problems. In *European Symposium on Algorithms*, pages 580–591. Springer, 2004.

[3] Mikkel Thorup and Uri Zwick. Approximate distance oracles. *Journal of the ACM (JACM)*, 52(1):1–24, 2005.