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 Algorithm Overview

2.1 Incremental SSSP

- **Input**: Directed graph $G = (V, E)$, source node s, distance bound k, edge insertions.
- **Output**: Shortest path distances from s to all nodes up to distance k.
- **Main Idea**: Maintain a truncated shortest-path tree using iterative BFS, updating distances only when insertions improve paths within bound k.
- **Complexity**: Insertion time $\mathcal{O}(km)$, query time $\mathcal{O}(1)$.

2.2 Greedy Spanner Construction

- **Input**: Weighted undirected graph $G = (V, E)$, stretch factor k.
- **Output**: A $(2k - 1)$-spanner with reduced edges preserving approximate distances.

- **Main Idea**: Greedily add edges to a sparse subgraph if they significantly reduce path lengths, using incremental SSSP for distance computations.
- **Complexity**: Construction time $\mathcal{O}(nm^2)$, space $\mathcal{O}(n^2 + m)$.

2.3 Fully Dynamic APSP (Insertions Only)

- **Input**: Unweighted directed graph $G = (V, E)$, edge insertions.
- **Output**: Approximate shortest path distances between all node pairs.
- **Main Idea**: Use random sampling and BFS trees to maintain approximate distances, optimizing for insertion updates.
- **Complexity**: Amortized insertion time $\mathcal{O}\left(\frac{m \log n}{k^2}\right)$, query time $\mathcal{O}\left(\frac{\log n}{k} + k\right)$.

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 Evaluation

4.1 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.

4.2 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 $\mathcal{O}(nm^2)$.

4.3 Comparisons

- **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.

5 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.

6 Conclusion

- Summarize key findings: Successful implementation of SSSP, spanner, and partial APSP, verified correctness, and tested scalability.
- Highlight enhancements: Synthetic dataset, weighted support, and iterative BFS improved applicability and robustness.
- Discuss limitations: Spanner's high runtime, incomplete APSP (no deletions).
- Future work: Optimize spanner construction, implement edge deletions, test on real datasets.

References