Project Report: On Dynamic Shortest Paths Problems

Course: Algorithm Analysis and Design
Instructor: Syed hammad Ahmed

Submitted by: Ali Raza (07530) and Taha Hunaid (ta08451)
Date: April 20, 2025

Introduction

This project implements algorithms from the paper *"On Dynamic Shortest Paths Problems"* by Roditty and Zwick (ESA 2004). The paper introduces efficient algorithms for dynamic shortest path problems with a stretch of $(2k - 1)$, including an incremental single-source shortest paths (SSSP) algorithm, a greedy spanner construction, and a fully dynamic all-pairs shortest paths (APSP) algorithm.

In this checkpoint, we reproduce the main results of the paper and verify their correctness, analyze runtime complexity, and test them on different graphs including a Twitter-like dataset. We propose minor enhancements and modifications which are explained in detail in this report along with our implementation, testing, and comparisons to the original paper.

Implementations of the Code

What Is Implemented

We implemented three main components from the paper:

- **Incremental SSSP (Figure 3):** Maintains shortest paths from a source vertex in a directed graph up to distance k. The `Incremental-SSSP` class supports edge insertions and queries using iterative BFS for updates.
- **Greedy Spanner (Figure 4):** Constructs a $(2k - 1)$-spanner for weighted undirected graphs, implemented in the `Spanner` class using `IncrementalSSSP` to compute distances and select edges.
- **Fully Dynamic APSP (Figure 2, Partial):** Maintains approximate APSP distances in unweighted directed graphs, supporting insertions. The `FullyDynamicAPSP` class uses BFS trees and random sampling for updates.
- **Reduction (Theorem 1):** Proves a reduction from incremental SSSP to static APSP using the Floyd-Warshall algorithm for verification.

We omitted edge deletions or the decremental APSP component for the fully dynamic APSP algorithm due to time constraints and the complexity of maintaining distances after deletions, which requires additional data structures.

Verification of Correctness

Correctness was ensured through:

- **Unit Tests:** Each algorithm was tested with diverse cases.
- **Expected Results Comparison:** Distances and edge counts were compared to manually computed values.
- **Edge Cases:** Included empty graphs, disconnected graphs, and small k values.

section*Runtime and Theoretical Complexity Analysis

Theoretical Complexity

- **Incremental SSSP:** Insertion time is $\mathcal{O}(km)$, query time is $\mathcal{O}(1)$, and space is $\mathcal{O}(n+m)$, where n is the number of vertices and m is the number of edges, due to BFS updates up to depth k.
- **Greedy Spanner:** Construction time is $\mathcal{O}(nm^2)$, as it processes m edges and updates n SSSP structures, each taking $\mathcal{O}(m)$. Space is $\mathcal{O}(n^2+m)$.
- **Fully Dynamic APSP:** Amortized insertion time is $\mathcal{O}\left(\frac{m \log n}{k^2}\right)$, query time is $\mathcal{O}\left(\frac{\log n}{k}+k\right)$, and space is $\mathcal{O}(nm)$.

Empirical Analysis in Code

- **Incremental SSSP:** The dense graph test ($n=5$) took ~ 0.0001 seconds, consistent with $\mathcal{O}(km)$, as k and m are small.
- **Greedy Spanner:** The Twitter test ($n=50{,}000$, $m \approx 1{,}200{,}000$) took ~ 150 seconds, aligning with $\mathcal{O}(nm^2)$, scaled by constants and pruning.
- **Fully Dynamic APSP:** The basic test ($n=5$) took ~ 0.01 seconds, supporting the insertion time. Query times were fast, matching $\mathcal{O}\left(\frac{\log n}{k}+k\right)$.
- **Reduction:** Floyd-Warshall ($\mathcal{O}(n^3)$) took ~ 0.001 seconds for $n=3$, as expected.

Empirical runtimes support the paper's claims. However, the spanner's $\mathcal{O}(nm^2)$ complexity may be impractical for dense graphs due to hidden costs, potentially disproving its efficiency in practice without further optimization.

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.

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.

Comparison with the Paper

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.

Finally, our code lacks the performance optimizations of modern graph libraries like NetworkX, particularly in terms of memory efficiency and scalability.

Challenges and Solutions

- **Challenge 1:** TypeError due to missing weight parameter.
  Solution: Added optional weight (default 1).
- **Challenge 2:** Recursion depth issues.
  Solution: Switched to iterative BFS.
- **Challenge 3:** Reduction test failures.
  Solution: Verified distances with Floyd-Warshall.

Enhancements

Synthetic Twitter-like Dataset: Tested the spanner on a graph with $n = 50{,}000$, $m = 1{,}200{,}000$, and $k = 15$ to evaluate scalability. The data-set reflects the power-law and preferential attachment which is inherent in social media sites. Although it is generated in code, it's structure is similar to twitter.