Algorithms Analysis and Design

Instructor: Hammad Khan

Team Members: Ali Raza, Taha Hunaid

Summary and Breakdown

Checkpoint 2

April 07, 2025

Technical Summary: Dynamic Shortest Paths Problems

Paper Details

- **Title:** On Dynamic Shortest Paths Problems
- **Authors:** Liam Roditty, Uri Zwick
- **Conference:** ESA 2004
- **Year:** 2004
- **DOI:** 10.1007/978-3-540-30140-0$_5$2

Problem and Contribution

The paper addresses the challenge of efficiently maintaining the shortest paths in a dynamically changing graph, where edges can be inserted or deleted over time. We divide the key contributions as:

- **Main Contribution:**
  - **Randomized Fully-Dynamic APSP Algorithm:**
    A new randomized algorithm is introduced for the APSP problem in unweighted directed graphs. It is especially effective for sparse graphs, offering faster update times than previous approaches. This makes it valuable for real-time applications, such as dynamic network routing, where efficient updates are essential.
- **Other contributions**
  - **Theoretical Foundation:**
    The authors show that incremental and decremental single-source shortest-paths (SSSP) problems are at least as difficult as the static all-pairs shortest-paths (APSP) problem in weighted graphs. For unweighted graphs, these problems are as hard as Boolean matrix multiplication. This result highlights the complexity of dynamic SSSP and suggests that improving classical algorithms may require solving problems as hard as static APSP.
  - **Spanner Construction Algorithm:**
    The paper proposes a deterministic algorithm for building $(\log n)$-spanners with $O(n)$ edges in weighted undirected graphs. This supports quicker and more efficient construction of sparse graph representations, which are critical for tasks in network design and approximation algorithms.

Algorithmic Description

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

Comparison with Existing Approaches

Approach	Update Time	Query Time	Graph Type
Roditty & Zwick	$\tilde{O}(m\sqrt{n})$	$O(n^{3/4})$	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:

- Fully dynamic (handles both insertions and deletions)
- Better update time than static recomputation
- Practical for medium-sized graphs

Implementation Challenges (Simplified)

- **Graph Representation:** The graph needs to be stored in a way that makes it quick and easy to insert or delete edges.
- **Sampling:** The method involves multiple levels of sampling, which must be set up carefully to work correctly.
- **Concurrency:** If updates happen at the same time, they might interfere with each other, causing errors.
- **Error Handling:** Since the algorithm uses probability, it needs to make sure the results are still accurate enough.

Evaluation Metrics (Explained Simply)

- **Amortized Update Time:**It measures the average time it takes to update the graph over many operations.
- **Worst-Case Query Time:** Looks at the maximum time a query might take in the worst situation.
- **Memory:** How much memory the algorithm uses.
- **Empirical Stretch Factor (for spanners):** Checks how close the distances in the simplified graph (spanner) are to the actual distances in the original graph.