Habib University

Syed Zuhair Abbas Rizvi, Syeda Rija Hasan Abidi,
Adina Adnan Mansoor

Dynamizing Dijkstra for Dynamic Shortest Paths

April 8, 2025

CS/CE 412/471 Algorithms: Design & Analysis
Technical Summary

Problem and Contribution

Over the years, many efficient algorithms have been proposed to solve the shortest path problem for a single source given a graph as an input. These fall apart when this problem becomes **dynamic**, where the input's topology—vertices, edges, or weights—morphs over time. The crux of the problem is efficiency; algorithms designed for the static version of the problem have to recompute paths entirely per update which is mediocre for applications such as real-time navigation.

Here is where this paper brings forward its contribution: a dynamization of Dijkstra's algorithm. Using primarily a **retroactive priority queue**, the new algorithm adjusts to updates with a time complexity per update of $O(n \log m)$ for a graph of n vertices and m edges. Most importantly, only affected paths are updated, a cleaner and faster approach as opposed to static algorithms which recompute every single path.

Algorithm Description

The core idea of the Dynamic Dijktsra algorithm is to adapt the traditional Dijkstra algorithm to handle dynamic graphs where the edge weights may change. Instead of recalculating the entire shortest path tree after each weight updates, the algorithm uses a retroactive priority queue to efficiently adjust operations in response to the weight change.

Inputs:

- A directed graph G = (V, E, w): where V is the set of Vertices, E is the set of edges and w, the function assigning weights to the edges.
- Source vertex: Starting vertex from where the shortest paths will be calculated
- Edge Weight Updates: A sequence of updates where the weight of a specific edge changes (increases or decreases)

Outputs:

- Updated shortest path distances from the source: after each edge weight update, the algorithm returns the new shortest path distances from the source to all the other vertex in the graph

High Level Logic

Initialization

Run the standard Dijkstra's algorithm to compute the initial shortest path tree, storing all distances and paths for all vertices. All the initial distances are stored in the RPQ at this point.

Update Handling

When an edge weight is updated, for example, an edge $e = (u, v)$ changes by a value α, we check if the head vertex u of the updated edge is in the RPQ.

- If u is not in the RPQ, then the update doesn't affect any of the shortest paths, and the algorithm continues with the current distances
- If u is active (it hasn't been finalized as part of any shortest path yet), and weight update reduces the edge weight, adjust the distance to v accordingly.
- If u was previously deleted (it was removed from the RPQ during the algorithm execution), then the algorithm averts to the state just before u was deleted and updates all distances for all affected vertices.

Propagation

The RPQ identifies which operations were affected by the update (such as "del-min" operations) and re-executes them. This ensures that only the necessary updates are made, and the shortest paths are recalculated for the affected vertices without recalculating the entire graph. Just like static Dijsktra, this algorithm provides the complexity of $O(n \log m)$ for the update time.

Comparison

The dynamic shortest path problem is an open problem with two overarching existing approaches. One involves reconstructing the graph using a shortest path algorithm on every change, while the other focuses on considering only the nodes and vertices affected by the change. All the modern approaches focus on the latter because of its efficiency. It has been implemented in many variants, but Ramalingam and Reps' algorithm [5][4] has been proven to be the best one in practice [1][3]. This algorithm works by identifying the set of affected nodes, which are divided into two sets: one queue of the affected nodes whose shortest path cannot be calculated using the current topology of the shortest path tree and one heap of the affected nodes whose updated shortest path can be calculated from the current topology of the shortest path tree. Demetrescu et al. enhanced this algorithm by introducing a mechanism to identify the set of affected nodes belonging to the queue, minimizing edge scans [2].

Our paper introduces an entirely novel approach to the dynamic shortest path problem by using a retroactive priority queue. Retroactivity helps maintain the historical sequence of events and updates the shortest path when the change occurred. The priority queue enables faster updates of the affected vertex by isolation from the unaffected graph. The number of computations in this case is fewer than those required for maintaining a non-retroactive priority queue because the algorithm targets the affected point in time and applies the update only to that portion. The paper compares the running time of our algorithm with that of Ramalingam et al. and Demetrescu et al.'s algorithms, both the original and those improved by the heap reduction technique proposed by Buriol et al. [1]. Dynamic Dijkstra performs better in both memory and time than other algorithms as shown in Table .

Algorithm	Time Taken (Seconds)	Memory Usage (10^6 Bytes)
Dynamic Dijkstra	**0.0075**	**118**
Demetrescu et al Optimized	0.008	130
Ramalingam et al Optimized	0.01	120
Demetrescu	0.0325	182
Ramalingam et al	0.04	140

Table 1: Comparisons with other Algorithms

Data Structures and Techniques

- **Retroactive Priority Queue (RPQ),** a priority queue that retroactively adjusts past operations to handle dynamic graph updates. This is the main tool for the algorithm.
- **Height-Balanced Trees**, balanced binary trees (T_ins for inserts, T_d_m for delete-min), meant to ensure $O(\log n)$ operations for efficient updates and queries. Two of them are used in the RPQ for this very purpose.
- **Graph**, a structure $G = (V, E, w)$ representing vertices, edges, and weights. Our input problem, and evolves over time.
- **Predecessor Array (pred)**, an array tracking each vertex's predecessor in the shortest path tree. This checks if updates affect paths.
- **Distance Array (d)**, an array storing estimated shortest path distances from the source. It updates and compares distances during execution.

Implementation Outlook and Challenges

- **Complexity of Data Structures:** Implementing the RPQ efficiently with red-black trees can be complex, especially when handling frequent updates or large input size
- **Handling large graphs (large input sizes):** As the number of vertices and edges increase, the time and memory needed to process updates also increase. Managing changes efficiently becomes more and more tricky, especially with more complex data structures like the retroactive priority queue
- **Numerical Precision:** When the edge weights are floating points, even tiny rounding errors can mess up the distance calculations. So, we need to be extra careful with how we handle these numbers to avoid any mistakes in the final paths.
- **Dynamic updates:** Frequent real-time updates (e.g., in traffic networks) demand efficient RPQ operations and graph modifications, possibly requiring parallel processing or optimized tree structures.

References

[1] L.S. Buriol, M.G. Resende, and M. Thorup. Speeding up dynamic shortest path algorithms. Technical Report TR TD-5RJ8B, AT&T Labs Research, 2003.

[2] C. Demetrescu and G. F. Italiano. A new approach to dynamic all pairs shortest paths. In *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC'03)*, pages 159–166, San Diego, CA, 2003.

[3] C. Demetrescu and G. F. Italiano. Experimental analysis of dynamic all pairs shortest path algorithms. *ACM Trans. Algorith.*, 2(4):578–601, 2006.

[4] G. Ramalingam and T.W. Reps. An incremental algorithm for a generalization of the shortest-path problem. *J. Algorith.*, 21(2):267–305, 1996.

[5] G. Ramalingam and T.W. Reps. On the computational complexity of dynamic graph problems. *Theoret. Comput. Sci.*, 158:233–277, 1996.