Dynamizing Dijsktra

*Note: Sub-titles are not captured in Xplore and should not be used

Rija Hasan Abidi
dept. Computer Science
Habib University
Karachi, Pakistan
sa07424

Zuhair Abbas
dept. Computer Science
Habib University
Karachi, Pakistan
sr07889

Adina Adnan
dept. Computer Science
Habib University
Karachi, Pakistan
am08349

***Abstract*—The classic Dijkstra algorithm efficiently solves the single-source shortest path problem in static graphs but fails to efficiently adapt to dynamic changes such as edge updates or deletions without full recomputation. This paper presents an implementation of dynamization of Dijkstra's algorithm using a retroactive priority queue, taking from the work of Sunitaa et al., allowing for efficient updates localized to affected vertices. By leveraging retroactivity and red-black trees, our method updates only necessary parts of the graph, achieving a per-update time complexity of $O((E+V)\log V)$ in the worst case and $O((\log V)^2)$ in the average case. Extensive experiments demonstrate that our Dynamic Dijkstra algorithm significantly outperforms static recomputation, particularly in sparse graphs and scenarios with frequent, localized updates. The approach provides a promising direction for real-time navigation systems and dynamic network analysis.**

***Index Terms*—Dynamic shortest paths, Dijkstra's algorithm, Retroactive priority queue, Dynamic graph algorithms, Real-time pathfinding, Red-black trees, Algorithm optimization**

I. Introduction

Over the years, many efficient algorithms have been proposed to solve the shortest path problem for a single source, given a graph as 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 m vertices and n edges. Most importantly, only affected paths are updated, a cleaner and faster approach as opposed to static algorithms, which recompute every single path.

II. Background & Motivation

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 [1] [2] has been proven to be the best one in practice [3] [4]. 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 [5]. Demetrescu et al. enhanced this algorithm by introducing a mechanism to identify the set of affected nodes belonging to the queue, minimizing edge scans [4]. Sunitaa et al.'s approach 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 it from the unaffected graph. The number of computations in this case is fewer than those required for maintaining a nonretroactive priority queue because the algorithm targets the affected point in time and applies the update only to that segment of the graph. 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. [3]. Dynamic Dijkstra performs better in both memory and time than other algorithms as shown in Table I.

TABLE I
COMPARISONS WITH OTHER ALGORITHMS

Algorithm	**Time Taken (s)**	**Memory Usage ($\times 10^6$ Bytes)**
Dynamic Dijkstra	**0.0075**	**118**
Demetrescu et al. (Optimized)	0.0080	130
Ramalingam et al. (Optimized)	0.0100	120
Demetrescu et al.	0.0325	182
Ramalingam et al.	0.0400	140

While the difference is small after the heap optimization technique; however, it is evident that the given dynamic Dijkstra with RPQ exceeds expectations and must be tested by impartial entities for verification and new solutions to the problem.

III. IMPLEMENTATION METHODOLOGY

A. Algorithm Summary

We implemented the dynamic Dijkstra algorithm in Python based on the pseudocode provided in the paper by Sunitaa et al. The heart of the algorithm is the RPQ, which is implemented through a red-black tree. The RPQ maintains two trees, T_ins and T_{d_m} that maintains sets of $Insert$ and Del_Min. T_ins is indexed by tuples of distance from source and time added (to the shortest path), and T_{d_m} is indexed by time only. These trees are linked whereby the deleted element from T_{ins} is not actually deleted from the tree once the user calls $invoke\_delmin$, but only marked invalid and added into T_{d_m} with the time stamp. The paper is extremely ambiguous on when these functions are exactly and how exactly the algorithm "goes back" in time. Hence, from this point onwards, we implemented the algorithm solely on guesswork and creativity.

While the paper attempts to ambiguously describe an algorithm that works on insertions and deletions of vertices and edges, our contribution is more focused on the update edge function. Due to the constant look-up time provided by adjacency matrices, we represented our input graph as an adjacency matrix, and updates are given as (u, v, w) where (u, v) is an edge $u- > v$ and w is the new weight. The RPQ is implemented similarly to paper but may or may not align with the usage of the paper given that it not mentioned when the functions of RPQ are called. Our implementation relies on three of the four functions given in the paper: $invoke\_insert$, $invoke\_delmin$, and $revoke\_delmin$.

- $invoke\_insert(u, t, d, \pi)$: Inserts the element u into T_{ins} with t as time inserted, d as distance from source, π as predecessor or parent.
- $invoke\_delmin(t)$: Marks the minimum element (by distance and time) invalid in T_{ins} and inserts it into T_{d_m} with t as time deleted.
- $revoke\_delmin(t)$: Revokes deletion of element deleted at or just before t from T_{d_m}.

Important functions of RPQ are:

- $search(t)$: Searches T_{d_m} by time.
- $find\_valid\_min\_ins()$: Finds the valid vertex with the minimum distance from the source in T_{ins}.

In our algorithm, the vertices are added to T_{ins} in the initial distance calculation and when a predecessor of a vertex is updated in the edge update function. The crux of the algorithm lies in the $invoke\_delmin(t)$, $revoke\_delmin(t)$, and a processed times array, which is maintained for each vertex. Whenever the shortest distance of a vertex is calculated, the time is recorded against the node. This enables us to "traverse" in time to historical states of vertices in T_{d_m}. Note that we need t parameter to call $revoke\_del\_min$, which is why processed times are necessary. Whenever a path is processed, it is deleted from T_{ins} by

$invoke\_delmin(t)$ and added to T_{d_m}. Only the active vertices are active in RPQ. When an edge (u, v) is updated and if the weight decreases, we keep the predecessor as is because we assume that all vertices were already updated and had shortest distances calculated. If the previous edge weight was greater and u was v's predecessor, then with even shorter distance, it must also be v's predecessor. In this case, the edge weight change is propagated to all of v's ancestors and their immediate parents to find if any predecessor changes due to the new change. On the other hand, if edge weight increases, the historical states of v are evaluated for better predecessors and the shortest distance from the source in the past. In this case too, the edge weight change is propagated to all of If a better and valid distance and v's ancestors in the same way. Better predecessors are restored with the $revoke\_delmin(t)$ function. The pseudocode of our algorithm is given in Algorithm 1.

IV. Experiments & Results

A. Correctness Testing

Multiple tests were run to test the retroactive updates of the algorithm. These were divided into:

1) **Isolated Edges**: Update edge **not involved** in any other shortest path
2) **Involved Edges**: Update edge **involved** in the shortest path of ancestors
3) **Repeated Updates**: **Reupdating** nodes and going back and forth in time

We will be using the graph in Table 1 to demonstrate the correctness of our algorithm. The initial distances calculated from the static Dijkstra algorithm are described in Snippet 2.

Isolated Edges: We refer to edges that are not involved in any of the shortest paths as isolated edges.

- **Weight increases:** If the weight of such an edge increases, it should have no effect on any of the paths or predecessors recorded. This can be shown in Figure 3. The increase of edge $(1, 4)$ was not involved in the shortest path of other vertices. Hence, after the update, only the path to 4 has changed.

Algorithm 1 Dynamic Shortest Path with Predecessor Updates

Input: Graph (Adjacency matrix)
Output: Predecessor of each node and shortest distances from source
Initialization:
  Run static Dijkstra to find initial predecessors and shortest distances of all nodes from source
  Insert each update into RPQ

procedure UPDATE(u, v, w)
    Input: (u, v): edge, w: weight
  if $w <$ current weight of (u, v) **then**
    Keep u as v's predecessor
    Update v's distance from source
  else
    Find all past v predecessors and weights through v's states in RPQ
    if better predecessor for v found with least weight **then**
      Make this u as v's predecessor
      Update distance of v from source through u
    end if
  end if
  Propagate changes to v's ancestors
  Relax outgoing edges and update their predecessors if needed
end procedure

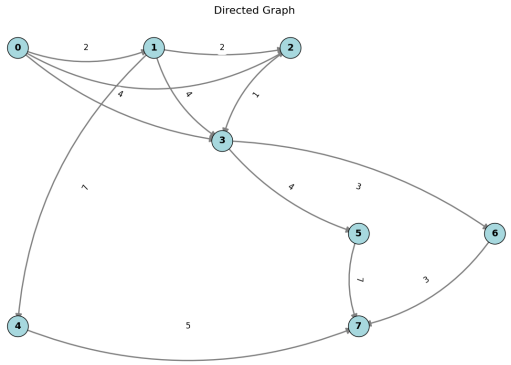

Fig. 1. Graph for testing

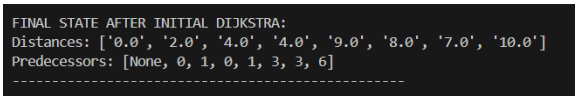

Fig. 2. Initial Distances

- **Weight decreases:** If the weight of such an edge decreases, a new shortest path may be present to its ancestors. In this case, the effect is cascading as seen in figure 4, the predecessor of 7 goes from 6 to 4.

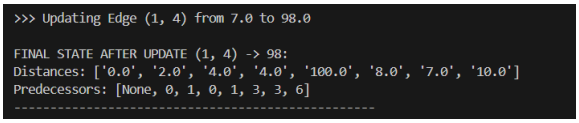

Fig. 3. Isolated Edge (1,4) - Weight Increases

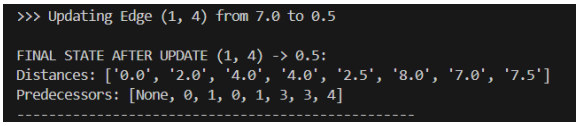

Fig. 4. Isolated Edge (1,4) - Weight Decreases

Involved Edge: Involved edges are the ones that have been involved in the shortest path calculated before, and updating these may or may not have an effect on the shortest paths to other vertices.

- **Weight increases:** If the weight of an involved edge increases, it may have an effect on the shortest path predecessors of other vertices. This can be shown in Figure 5. The increase of edge $(0, 3)$ to 100 changes the paths to 5 and 6 and as a result, the predecessor of 7 becomes 4 from 6. This is the **cascading effect** where updates to one edge can lead to many other updates.
- **Weight decreases:** If the weight of an involved edge decreases, all predecessors remain the same, but the distances are updated in a cascading manner. This is shown in figure 6 where the update of (0,3) to 1 now changes only the distances, not predecessors.

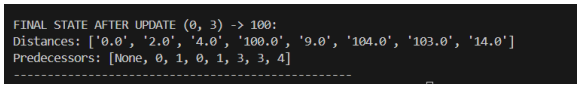

Fig. 5. Involved Edge (0,3) - Weight decreases

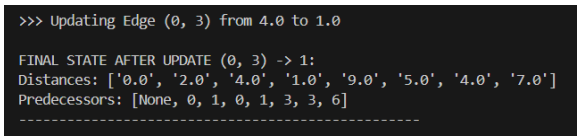

Fig. 6. Involved Edge (0,3) - Weight decreases

Repeated updates: Repeated updates entail going back in time to revoke deleted operations and propagate the changes to all vertices dependent on the update. This is shown in the figure in 7 where edge (0,3) increases to 100, decreases to 1, and then increases to 100 again. In this case, the algorithm finds the deletion time of the vertex affected, 7 in this case, and rolls back the state to the previous distance and predecessor. This ensures that paths are not calculated repeatedly, but fetched from the right moment of time in the past.

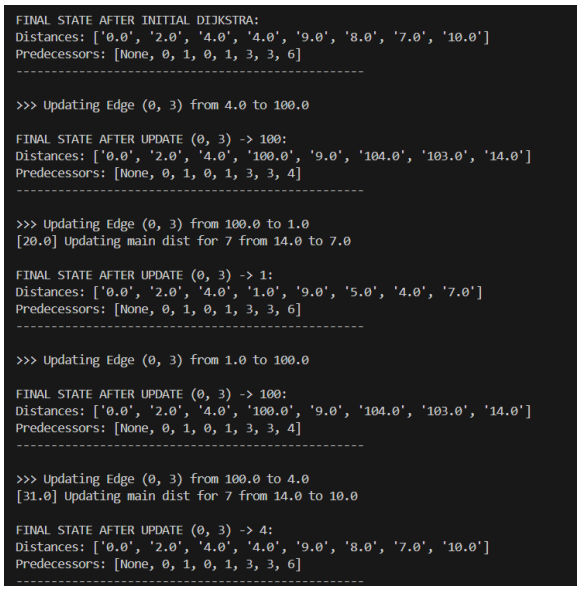

Fig. 7. Repeated Updates on node 3

B. Complexity Analysis

First, let's look at the RPQ. The RPQ operations (insert, delete-min, decrease-key) take $O(\log k)$, where $k \leq V + E$, so roughly $O(\log V)$.

In the worst case, an update (e.g., reducing a source-adjacent edge weight) impacts all V vertices, requiring V extract-mins and E decrease-keys. V extract-mins cost $O(V \log V)$ and E decrease-keys cost $O(E \log V)$. Summing these gives us:

$$O(V \log V) + O(E \log V) = \mathbf{O}((\mathbf{E} + \mathbf{V}) \log \mathbf{V})$$

What about the average case? Assume a sparse graph ($E \approx V$) and a balanced shortest path tree, where an update affects $a = O(\log V)$ vertices (e.g., a subtree in a binary-like tree, averaged over depths). Operations include $O(a)$ extract-mins and $O(a)$ decrease-keys, each $O(\log V)$, so we can freely consider both as the same. Our average-case complexity is

$$O(a \log V) = O(\log V \cdot \log V) = \mathbf{O}((\log \mathbf{V})^{\mathbf{2}})$$

.

C. Baseline Evaluation - Static Dijkstra

An obvious baseline is the standard or static Dijkstra algorithm. Traditionally, it takes $O(V^2)$; with a priority queue, it optimizes to $\mathbf{O}((\mathbf{V} + \mathbf{E}) \log \mathbf{V})$ for **any** update, not just the worst one.

From our prior analysis, While the worst-case complexity is the same as the optimized standard Dijkstra's algorithm, the dynamic Dijkstra shows its speedup in the average case of localized updates. So, in theory, dynamic Dijkstra is much more optimized for the average case than its static counterpart.

We have compared our algorithm with the baseline evaluation which is the approach that involves rerunning the Dijsktra algorithm after each update. The results are shown in the table below. The dynamic algorithm performs much better than the static version, especially when the graph becomes larger and the number of updates increase. The results are shown in the table below.

TABLE II
COMPARISON OF STATIC AND DYNAMIC DIJKSTRA'S ALGORITHM

Vertices (n)	Edges (m)	Updates	Static Dijkstra Time (s)	Dynamic Dijkstra Time (s)
10	15	10	0.0000	0.0000
20	30	20	0.0029	0.0000
50	100	50	0.0010	0.0000
100	200	150	0.0059	0.0010
200	500	200	0.0207	0.00102

D. Comparison Evaluation - Bellman Ford

We also compared our algorithm with the Bellman-Ford algorithm and found that the difference was immense. At 20,000 nodes, 20,000 edges, and 5,000 updates, our Dynamic Dijkstra algorithm completed in 18 seconds, whereas the Static Dijkstra algorithm took 182 seconds, and the Bellman-Ford algorithm took approximately 9 hours.

In another instance, standard Dijkstra took 180 seconds for a specific update set, while our Dynamic Dijkstra completed the same task in just 0.03 seconds. For comparison, Bellman-Ford took 180 seconds to solve a problem that Dynamic Dijkstra completed in 0.02 seconds.

These results clearly demonstrate the significant efficiency gains achieved by our approach compared to traditional methods, particularly under large-scale and frequent update scenarios.

TABLE III
COMPARISON OF BELLMAN-FORD AND DYNAMIC DIJKSTRA'S ALGORITHM

Vertices (n)	Edges (m)	Updates	Bellman-Ford Time (s)	Dynamic Dijkstra Time (s)
10	15	10	0.6926	0.0000
20	30	20	7.0035	0.0000
50	100	50	20.02	0.0000
100	200	150	180.12	0.0010
200	500	200	328.5	0.0010
1000	5000	500	2850.0	0.0025
2000	1000	1000	7639.0	5.0050
20000	20000	5000	32400	18.050

E. Comparitive Correctness

We compared the shortest distances and predecessors calculated by our dynamic algorithm with

the results from the recomputation of shortest distances by running static Dijkstra and Bellman-Ford algorithms. All of the results calculated by our algorithm were exactly similar to the ones calculated by the recomputation of these algorithms, although in a fraction of the time, as will be demonstrated in Tables II and III. Our testing code also compared the results from all algorithms as represented in the figure 8.

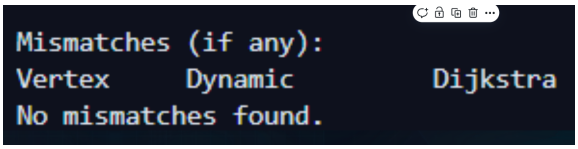

Fig. 8. Correctness of Dynamic Dijsktra

V. Conclusion

In this paper, we implemented a dynamic variant of Dijkstra's algorithm utilizing a retroactive priority queue to address the dynamic shortest path problem. Our approach selectively updates only the affected vertices and their ancestors rather than recomputing the entire graph, resulting in considerable time and memory savings. Through detailed correctness testing, complexity analysis, and baseline comparisons, we demonstrated that Dynamic Dijkstra achieves better performance than traditional static recomputation methods, especially in graphs subject to frequent local changes. Future work could explore extending the retroactive priority queue approach to multi-source shortest paths and weighted dynamic graphs with negative edges, further broadening the applicability of dynamic pathfinding solutions.

References

[1] G. Ramalingam and T. Reps, "On the computational complexity of dynamic graph problems," *Theoret. Comput. Sci.*, vol. 158, pp. 233–277, 1996.

[2] ——, "An incremental algorithm for a generalization of the shortest-path problem," *J. Algorith.*, vol. 21, no. 2, pp. 267–305, 1996.

[3] L. Buriol, M. Resende, and M. Thorup, "Speeding up dynamic shortest path algorithms," AT&T Labs Research, Tech. Rep. TR TD-5RJ8B, 2003.

[4] C. Demetrescu and G. F. Italiano, "Experimental analysis of dynamic all pairs shortest path algorithms," *ACM Trans. Algorith.*, vol. 2, no. 4, pp. 578–601, 2006.

[5] ——, "A new approach to dynamic all pairs shortest paths," in *Proceedings of the 35th Annual ACM Symposium on Theory of Computing (STOC'03)*, San Diego, CA, 2003, pp. 159–166.