Habib University

Rija Hasan, Zuhair Abbas, Adina Adnan

Dynamizing Dijkstra for Dynamic Shortest Paths

April 21, 2025

CS/CE 412/471 Algorithms: Design & Analysis
Implementation Report

Implementation Summary

We implemented the dynamic Dijkstra algorithm in Python based on the pseudocode provided in the paper. All of the algorithm is complete and tested, as we elaborate in the next section. It is worth mentioning that there were typos in the pseudocode (e.g: line 10 ”Update estimated distance ($d[e_t]$) for the vertex e_t in priority queue should have been to update e_h aka the current distance we have travelled) which we went out of our way to correct.

Let us begin with the main data structure for the algorithm: the retrospective priority queue (RPQ).

The RPQ is managed using red black trees, where nodes represent vertices and their associated information, such as distance, time of insertion, and predecessors. It is divided into two trees: one for insertions (T_ins) and one for deletions (T_{d_m}). The algorithm checks for conflicts between these two trees when edges are updated. When an edge's weight is updated, the algorithm isolates that specific edge update instead of recalculating the entire graph. This is done by updating only the affected edges and their ancestors. The cascading effect ensures that if an edge is part of the shortest path, its change propagates to all relevant ancestors, adjusting the distances accordingly.

After updating an edge, the algorithm propagates changes to the ancestors of the affected vertex, revisiting previously processed nodes and checking for better paths. Cascading updates ensure that the changes are applied incrementally, only affecting the nodes that are influenced by the updated edge. When an edge deletion causes a change in the shortest path, the relevant entries are marked as invalid, and the algorithm checks whether any of the previously invalidated entries need to be revived.

The heart of the algorithm's efficiency is through only updating the necessary parts of the graph. By focusing on specific edge updates rather than recalculating the entire graph, it minimizes computation.

Correctness Testing

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

1. Update edge **not involved** in any other shortest path
2. Update edge **involved** in the shortest path of ancestors
3. **Reupdating** nodes and going back and forth in time

We will be using the graph in 1 to demonstrate the correctness our 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.

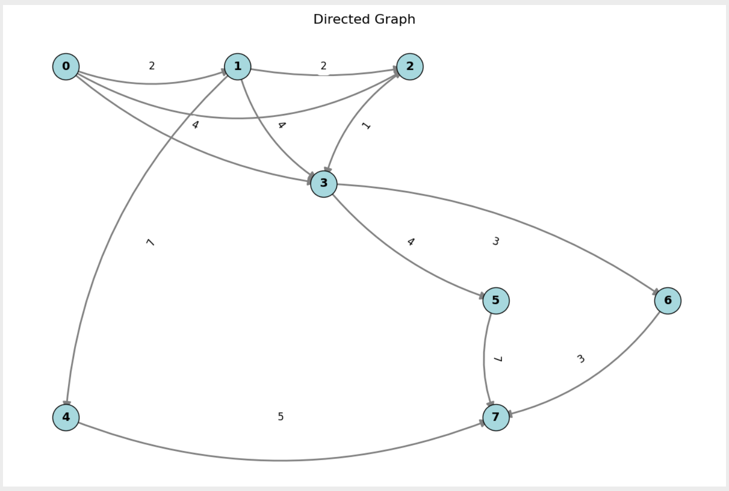
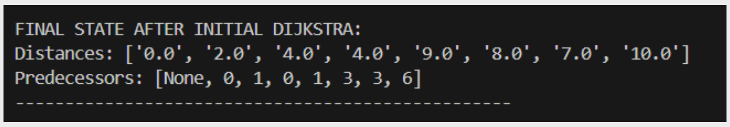


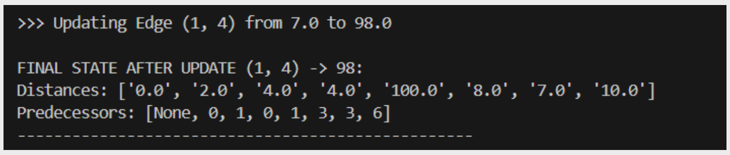
Figure 1: Graph for testing



```
FINAL STATE AFTER INITIAL DIJKSTRA:
Distances: ['0.0', '2.0', '4.0', '4.0', '9.0', '8.0', '7.0', '10.0']
Predecessors: [None, 0, 1, 0, 1, 3, 3, 6]
--------------------------------------------------
```

Figure 2: Initial Distances

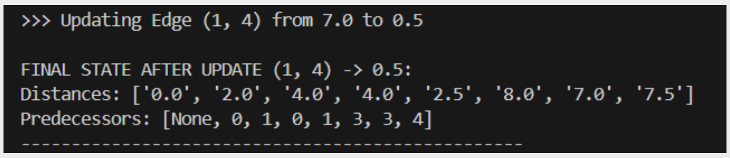
- **Weight decreases:** If the weight of such an edge decreases, a new shortest path may be present to its ancestors. In this case, the effect is cascading as seen in figure 4, the predecessor of 7 goes from 6 to 4.



```
>>> Updating Edge (1, 4) from 7.0 to 98.0

FINAL STATE AFTER UPDATE (1, 4) -> 98:
Distances: ['0.0', '2.0', '4.0', '4.0', '100.0', '8.0', '7.0', '10.0']
Predecessors: [None, 0, 1, 0, 1, 3, 3, 6]
--------------------------------------------------
```

Figure 3: Isolated Edge (1,4) - Weight Increases



```
>>> Updating Edge (1, 4) from 7.0 to 0.5

FINAL STATE AFTER UPDATE (1, 4) -> 0.5:
Distances: ['0.0', '2.0', '4.0', '4.0', '2.5', '8.0', '7.0', '7.5']
Predecessors: [None, 0, 1, 0, 1, 3, 3, 4]
--------------------------------------------------
```

Figure 4: Isolated Edge (1,4) - Weight Decreases

Involved Edge:

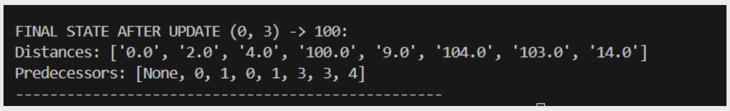
Involved edges are the ones that have been involved in the shortest path calculated before, and updating these may or may not have an effect on the shortest paths to other vertices.

- **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.

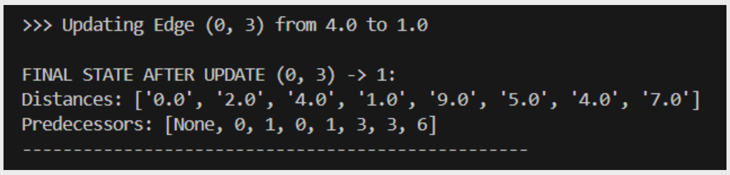
Repeated updates:

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.



```
FINAL STATE AFTER UPDATE (0, 3) -> 100:
Distances: ['0.0', '2.0', '4.0', '100.0', '9.0', '104.0', '103.0', '14.0']
Predecessors: [None, 0, 1, 0, 1, 3, 3, 4]
--------------------------------------------------
```

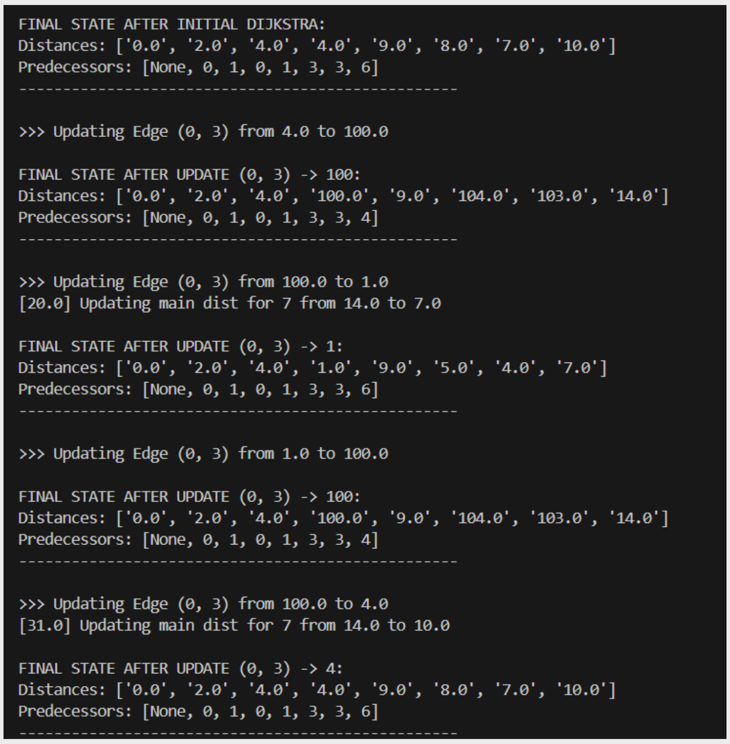
Figure 5: Involved Edge (0,3) - Weight decreases



```
>>> Updating Edge (0, 3) from 4.0 to 1.0

FINAL STATE AFTER UPDATE (0, 3) -> 1:
Distances: ['0.0', '2.0', '4.0', '1.0', '9.0', '5.0', '4.0', '7.0']
Predecessors: [None, 0, 1, 0, 1, 3, 3, 6]
--------------------------------------------------
```

Figure 6: Involved Edge (0,3) - Weight decreases



```
FINAL STATE AFTER INITIAL DIJKSTRA:
Distances: ['0.0', '2.0', '4.0', '4.0', '9.0', '8.0', '7.0', '10.0']
Predecessors: [None, 0, 1, 0, 1, 3, 3, 6]
--------------------------------------------------

>>> Updating Edge (0, 3) from 4.0 to 100.0

FINAL STATE AFTER UPDATE (0, 3) -> 100:
Distances: ['0.0', '2.0', '4.0', '100.0', '9.0', '104.0', '103.0', '14.0']
Predecessors: [None, 0, 1, 0, 1, 3, 3, 4]
--------------------------------------------------

>>> Updating Edge (0, 3) from 100.0 to 1.0
[20.0] Updating main dist for 7 from 14.0 to 7.0

FINAL STATE AFTER UPDATE (0, 3) -> 1:
Distances: ['0.0', '2.0', '4.0', '1.0', '9.0', '5.0', '4.0', '7.0']
Predecessors: [None, 0, 1, 0, 1, 3, 3, 6]
--------------------------------------------------

>>> Updating Edge (0, 3) from 1.0 to 100.0

FINAL STATE AFTER UPDATE (0, 3) -> 100:
Distances: ['0.0', '2.0', '4.0', '100.0', '9.0', '104.0', '103.0', '14.0']
Predecessors: [None, 0, 1, 0, 1, 3, 3, 4]
--------------------------------------------------

>>> Updating Edge (0, 3) from 100.0 to 4.0
[31.0] Updating main dist for 7 from 14.0 to 10.0

FINAL STATE AFTER UPDATE (0, 3) -> 4:
Distances: ['0.0', '2.0', '4.0', '4.0', '9.0', '8.0', '7.0', '10.0']
Predecessors: [None, 0, 1, 0, 1, 3, 3, 6]
--------------------------------------------------
```

Figure 7: Repeated Updates on node 3

Visualization:

Here is a visual example with the same graph as before. This was what was obtained after running an initial Dijkstra before updates.

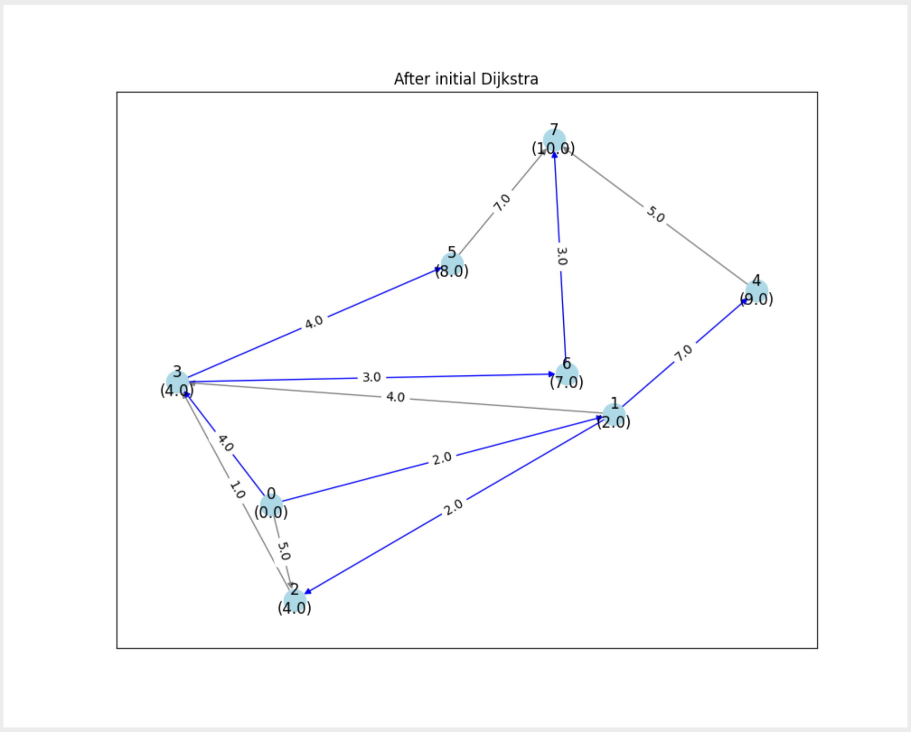


Figure 8: Test graph's initial distances.

Let's say one of the distances gets updated. Specifically, edge (5, 7) morphs from a distance of 7.0 to 1.0 instead. The algorithm processes and updates accordingly to give a new shortest path to node 7.

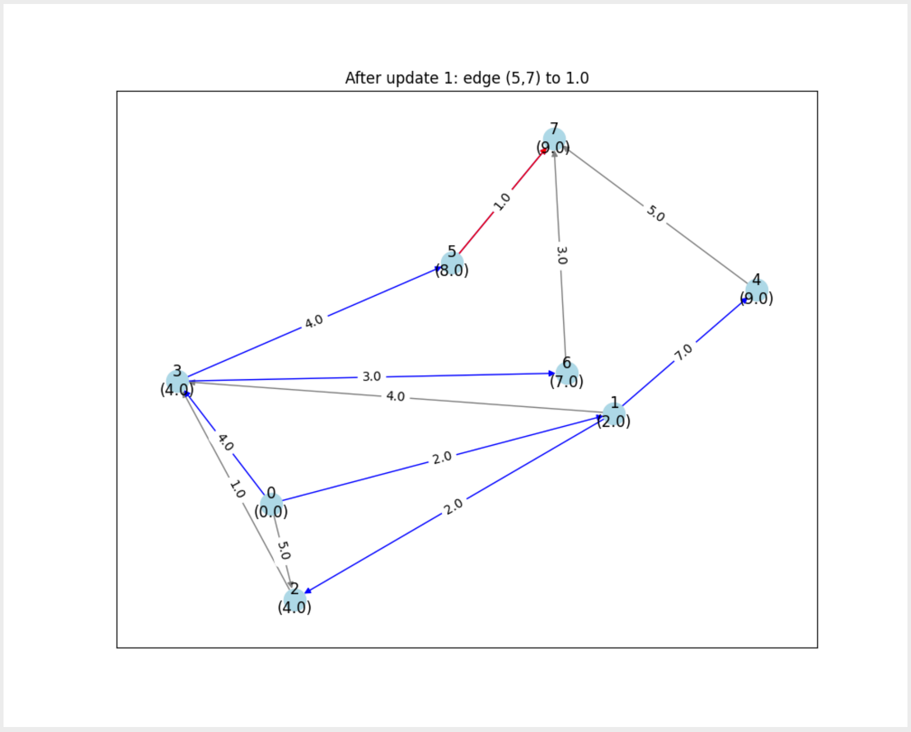


Figure 9: After updating (5, 7).

Similarly, we may update another edge (e.g: (3, 6)) from 3.0 to 1.5. This gives us a newer path to node 7.

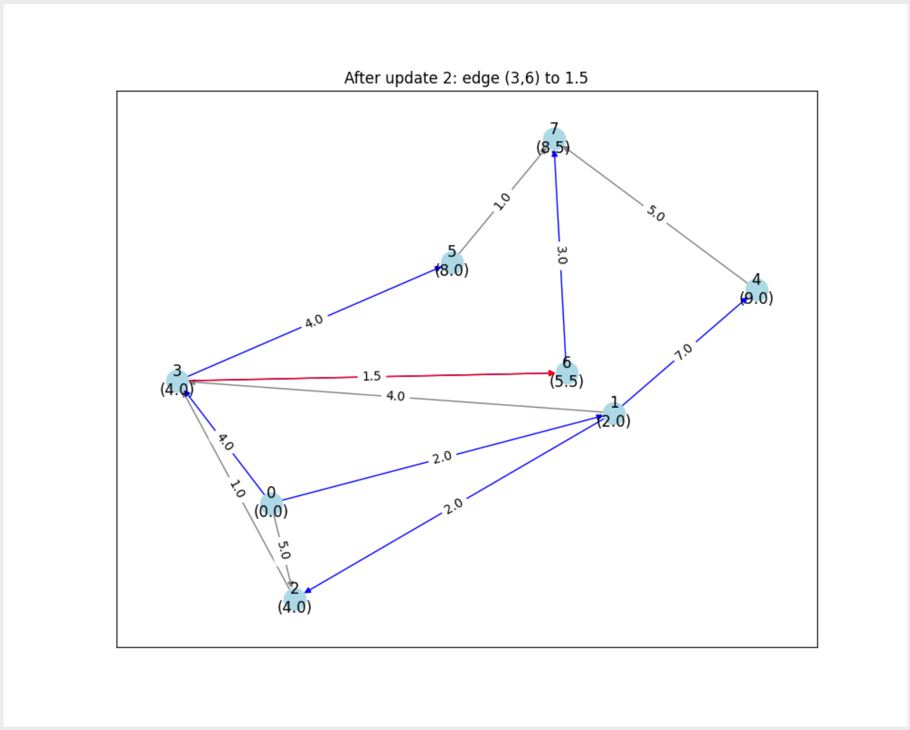


Figure 10: After updating (3, 6).

Complexity & Runtime 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

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

.

Baseline or Comparative Evaluation

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.

Vertices (n)	Edges (m)	Num of Updates	Time Taken by Static Dijkstra	Time Taken by Dynamic Dijkstra
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

Table 1: Comparison of Static and Dynamic Dijkstra's Algorithm

Challenges & Solutions

We have faced significant challenges in the implementation of this algorithm, whether it be due to the complexity of the concept or the scalability of the solution.

- **Paper's Description:** The pseudocodes present in the paper detail ambiguous functions that were extremely difficult to implement without any other sources on the material available. The figures in the charts had typos where entries changed, moving from one state to another without any logical progression. We dealt with this problem by discussing the approach amongst ourselves in coordinated sessions.
- **Complexity:** Our initial algorithm did not actually utilize the RPQ as well as it should have. Though we did not perform a theoretical analysis of it, runtime results showed time taken similar to Dijkstra for larger graphs, which we found inadequate. This required a reevaluation of the algorithm to better fit the paper.
- **Repeated Edge Updates:** Another challenge that was faced was managing repeated edge updates. When the weight of an edge is updated multiple times, it was causing inconsistencies in the graph's state. The primary concern was to handle these repeated updates without recalculating the the shortest paths for the entire graph. To resolve this, we isolated the updates to only affected edges and vertices, using RPQ to track changes and propagate them efficiently. This approach prevented unnecessary recalculations and ensured consistency, allowing the system to handle multiple edge updates without restarting the entire process.

Enhancements

- Our algorithm is not using the heap library to optimize the memory of RPQ. Once we implement that, the algorithm is expected to perform with much faster practical running time.
- The algorithm was corrected as compared to the pseudocode in the paper. Particularly:

Lines 6 and 10: Update estimated distance ($d[e_t]$) for the vertex e_t should be $d[e_h]$ for vertex e_h, as the distance to the head vertex, **not** the tail, is affected by the edge (e_t, e_h).

Line 14: $d[e`h] = 1$ should be $d[e`h] = \inf$ to invalidate the distance of a vertex whose predecessor is e_h.

- We also tested it out on a different, more ambitious dataset than the papers covered, as described before.