Algorithms: Design and Analysis
Checkpoint 3

Team 53
Qurba Mushtaq 08232, Hiba Shahid 08036

Reviewed research paper title:
Bidirectional Dijkstra's Algorithm is Instance-Optimal
by Bernhad Haeupler, Richard Hladik, Vaclav Rozhon, Robert E. Tarjan

1 Introduction

Bidirectional Dijkstra's algorithm is shown to be instance-optimal for finding the shortest path between two nodes in a graph. This optimality is proven for weighted multigraphs, both directed and undirected, with the guarantee measured in terms of the number of edges explored. It is important to note that this result holds only for graphs with strictly positive weights. If zero or negative weights are allowed, the assumptions required for instance-optimality break down. This report explores how bidirectional Dijkstra compares to other shortest path algorithms and under what theoretical and practical conditions it achieves optimality.

2 Aim of the Study

The paper establishes that no correct algorithm can outperform the proposed version of bidirectional Dijkstra on any single instance by more than a constant factor. This result is particularly significant in the adjacency list model, where access to the graph is limited to querying nodes and their neighbors. The optimality guarantee extends even to unweighted graphs, where the algorithm remains instance-optimal up to a factor of $O(\Delta)$, where Δ denotes the maximum degree of the graph. Theorem 1.1 in the paper formalizes this result.

3 Algorithm Overview and Motivation

While many algorithms can compute shortest paths, few are instance-optimal. In large graphs, exploring all paths is inefficient. Bidirectional Dijkstra improves this by exploring simultaneously from both the source and target and stopping early when the shortest path has been identified. Earlier variants of this idea did not guarantee optimality, often halting too early or traversing more than necessary. This paper introduces a variant with proven optimal stopping and exploration rules.

4 Algorithm Design, Correctness, and Optimality

4.1 Unidirectional vs. Bidirectional Dijkstra

The paper proves that unidirectional Dijkstra is instance-optimal under a restricted setting where only out-neighbor access is allowed. In such a model, even randomized algorithms cannot outperform it.

The proposed bidirectional algorithm runs two Dijkstra searches—one from the source (s) and one from the target (t)—alternating between them. At each step, it relaxes one edge from the active frontier. The search stops when:

$$\hat{d}(s, u_s) + \hat{d}(u_t, t) > \mu$$

where μ is the shortest path length found so far. This ensures no better path remains undiscovered. This is explained in further detail in our implementation in section 7 7.

4.2 Correctness and Comparison

Once the forward and backward searches meet at a common vertex or edge, the path is built by merging the two. If any shorter path existed, it would have been found before the stopping condition was satisfied. This guarantees correctness.

While Dijkstra explores more edges than necessary and A* requires extra heuristic information, the bidirectional version in this paper achieves strong performance without such assumptions. Even in large graphs, it explores fewer edges and runs faster in practice.

4.3 Instance Optimality

To prove that no correct algorithm can consistently outperform it, the authors construct two graph versions G and G'. An algorithm that skips a critical edge in G will return an incorrect result in G', where that edge is essential. This contradiction shows that the bidirectional algorithm explores the fewest possible edges without compromising correctness.

5 Remarks and Theoretical Limits

The algorithm is only instance-optimal when edge weights are **strictly positive**. If weights are zero or negative, alternative algorithms can outperform it. The paper also proves that even in unweighted graphs, no algorithm can do better than a factor of $O(\Delta)$, where Δ is the maximum degree of the provided graph.

6 Implementation Summary

We implemented Bidirectional Dijkstra's algorithm in Python using adjacency lists. The algorithm maintains two priority queues: one for forward search from the source, and one for backward search from the target. These searches alternate at each step, exploring one edge at a time. The stopping condition ensures that the shortest path has been found when no better path can exist.

The full implementation includes custom graph construction, forward and reverse edge tracking, edge relaxation, and detailed logging of traversal metrics. We also implemented baseline algorithms: unidirectional Dijkstra and NetworkX's built-in method for performance comparison.

7 How It Works

The bidirectional Dijkstra algorithm works by running two simultaneous searches—one from the source node s and one from the target node t. Each search maintains its own set of distances and predecessors.

- **Forward Search:** Begins at node s, using a priority queue to explore vertices. It updates shortest distances (`d_forward`) and predecessors (`pred_forward`) as it goes.
- **Backward Search:** Begins at node t, exploring the graph in reverse (i.e., following incoming edges in directed graphs). It updates `d_backward` and `pred_backward`.
- **Meeting Point:** The forward and backward searches eventually meet at a common vertex or edge. This meeting point is used to construct a complete path from s to t.
- **Stopping Condition:** The algorithm stops when the combined current distances from both searches exceed the best known path length (μ). This ensures that continuing would not find a better path.
- **Path Construction:** Once the searches meet, the path is built by combining the forward path to the meeting point with the backward path from that point to t. The final result is the shortest path from s to t.

8 Correctness Testing

To ensure the implementation was correct, we tested it on graphs of varying sizes and edge densities. Sample test cases included:

- Small graphs where shortest paths could be calculated manually.
- Graphs with multiple shortest paths of the same length.
- Disconnected graphs to ensure no invalid paths were returned.

In each case, the paths and distances returned matched those from standard Dijkstra and NetworkX's implementation.

9 Challenges and Enhancements

The main challenge was implementing the correct stopping condition. In early tests, the algorithm stopped too early or too late, depending on how we computed the current best path length μ. We fixed this by tracking the shortest overlapping path during exploration and applying the correct check condition.

We enhanced our code to collect the number of edge relaxations and average execution time for each algorithm over multiple iterations. This allowed us to generate meaningful visual comparisons and see how performance scaled with graph size. We also generated synthetic graphs of different types using a helper function and NetworkX's random graph generators.

10 Empirical Verification

To prove that the proposed algorithm is, in fact, instance-optimal, we evaluated its performance on varying graphs and compared it against classical Dijkstra and A*. We focused on two metrics: **execution time** and **number of edges traversed**.

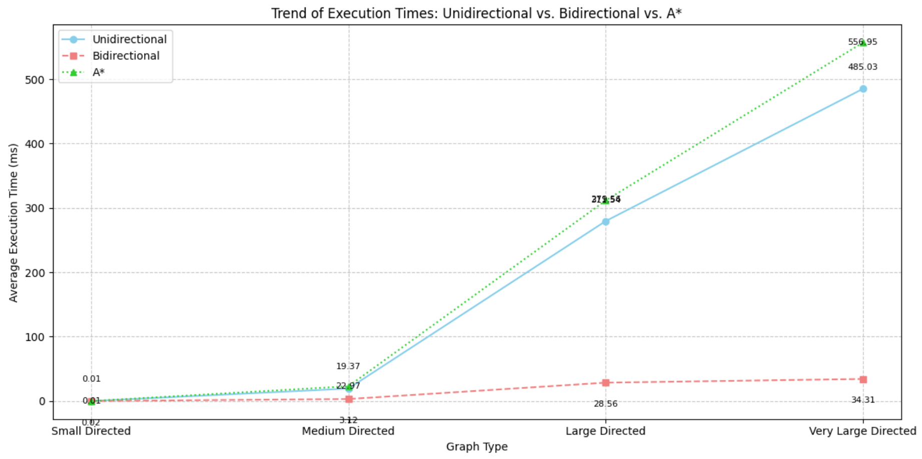

Figure 1: Comparison of Execution Time across Algorithms

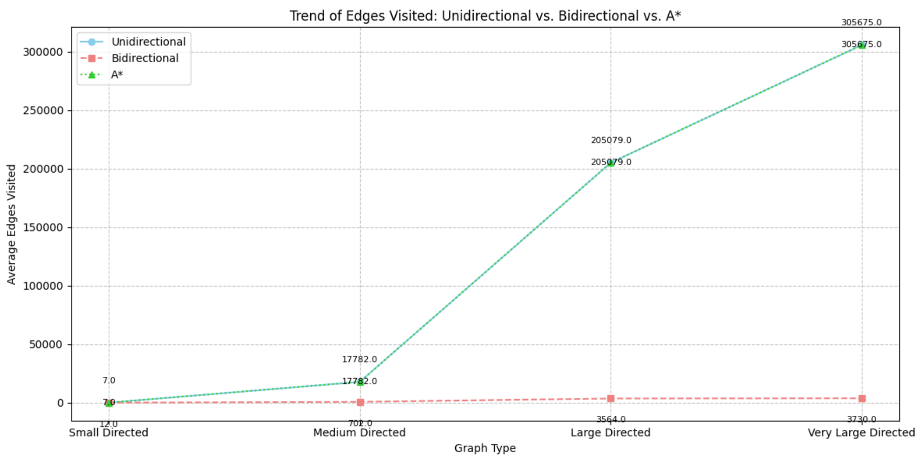

Figure 2: Comparison of Edges Traversed across Algorithms

From the figures 1 and 2, it is evident that the bidirectional Dijkstra algorithm outperforms traditional Dijkstra and A* in the no-heuristic setting. This difference becomes especially clear in larger graphs. The number of edges it explores is much lower, and this leads to shorter execution times as well. This supports the paper's claim that bidirectional Dijkstra is instance-optimal.

11 Complexity Analysis

The time and space complexity of Bidirectional Dijkstra's algorithm depends on the number of vertices V and edges E in the graph.

In the worst case, the algorithm behaves similarly to standard Dijkstra's algorithm:

- Time complexity: $O((V + E) \log V)$, assuming a binary heap is used for the priority queue.
- Space complexity: $O(V + E)$, due to storing distances, predecessors, and the graph structure.

However, in practice, the bidirectional approach explores far fewer nodes and edges, especially in large graphs. Since the searches start from both ends, the area explored is roughly half in each direction. This leads to a practical improvement in performance, though the theoretical worst-case remains similar. This is evident in the trends we monitored in figures 1 and 2.

In graphs with small diameters or dense connections, the benefits of bidirectional search are even more noticeable. Our empirical results confirm that the average number of edge relaxations and execution time are significantly lower than unidirectional Dijkstra.

12 Datasets and Testing Material

We tested the algorithm on both manually constructed graphs and synthetic data. The larger test cases were generated using our own functions and **NetworkX**'s random graph tools. Each algorithm was run multiple times on the same inputs, and average times and edge counts were recorded.

13 Future Work and Reflections

It would be interesting to see if these instance-optimality ideas can be extended to A* or other heuristic-based algorithms. Another idea is testing the algorithm on real-world graphs like road networks or social graphs to see if the observed gains carry over. We are also curious about how the results would differ on undirected graphs or graphs with zero-weight edges.

14 GitHub repository

URL: `https://github.com/HibaShahidA/Bidirectional-Dijkstra`