Advanced Algorithms:
Shortest Path Comparison Study

A Comprehensive Analysis of Dijkstra’s Algorithm,
Bellman-Ford Algorithm, and A* Search

Author: Riccardo
Date: 2025-08-07

Implementation and Performance Comparison
of Three Fundamental Shortest Path Algorithms

Table of Contents

1. Introduction

The shortest path problem is one of the most fundamental and well-studied problems in computer science and graph theory. Given a weighted graph and two vertices, the goal is to find a path between them that minimizes the sum of edge weights. This problem has numerous practical applications, from GPS navigation systems to network routing protocols, from game AI pathfinding to social network analysis.

This study implements and analyzes three classical algorithms for solving shortest path problems:

- **Dijkstra's Algorithm**: The gold standard for graphs with non-negative edge weights
- **Bellman-Ford Algorithm**: A versatile algorithm that handles negative edge weights and detects negative cycles
- **A* Search**: An informed search algorithm that uses heuristics to guide the search toward the goal

Each algorithm represents a different approach to the problem, with distinct strengths, limitations, and computational complexities. Through systematic implementation and empirical analysis, this project aims to provide insights into their theoretical properties and practical performance characteristics.

1.1. Objectives

The primary objectives of this study are:

1. Implement all three algorithms with comprehensive documentation and performance monitoring
2. Generate diverse test instances including sparse graphs, dense graphs, grids, and graphs with negative edges
3. Conduct systematic performance benchmarking across different graph types and sizes
4. Analyze the relationship between theoretical complexity and practical performance
5. Visualize and interpret the results to gain deeper insights into algorithm behavior

1.2. Project Structure

The implementation is organized into modular components:

- `graph.py`: Graph representation and test instance generation utilities
- `dijkstra.py`: Complete implementation of Dijkstra's algorithm with performance metrics
- `bellman_ford.py`: Bellman-Ford algorithm with negative cycle detection
- `astar.py`: A* search for both general graphs and grid-based pathfinding
- `main.py`: Comprehensive benchmarking framework with visualization capabilities

2. Theoretical Background

2.1. Dijkstra's Algorithm

Dijkstra's algorithm, proposed by Edsger W. Dijkstra in 1956, is a greedy algorithm that finds the shortest path from a source vertex to all other vertices in a weighted graph. The algorithm maintains the invariant that once a vertex is processed, its shortest distance from the source is known and will never change.

2.1.1. Algorithm Description

The algorithm maintains a set S of vertices whose shortest distance from the source s has been determined, and a priority queue Q of vertices to be processed. Initially, $S = \emptyset$ and Q contains all vertices with distance $d[s] = 0$ and $d[v] = \infty$ for all $v \neq s$.

At each iteration, the algorithm:
1. Extracts the vertex u with minimum distance from Q
2. Adds u to S
3. For each neighbor v of u, performs the **relaxation** operation: $d[v] = \min(d[v], d[u] + w(u, v))$

The relaxation operation is the key insight: if we can reach vertex v through u with a shorter total distance, we update v's distance estimate.

2.1.2. Mathematical Formulation

Let $G = (V, E)$ be a weighted directed graph with weight function $w : E \to \mathbb{R}^+$. For a source vertex $s \in V$, Dijkstra's algorithm computes:

$$\delta(s, v) = \min\left\{\sum_{e \in P} w(e) : P \text{ is a path from } s \text{ to } v\right\}$$

The algorithm's correctness relies on the **optimal substructure** property: if P is a shortest path from s to v, then any subpath of P is also a shortest path between its endpoints.

2.1.3. Complexity Analysis

Time Complexity: The algorithm's performance depends on the implementation of the priority queue:
- With a binary heap: $O((|V| + |E|) \log|V|)$
- With a Fibonacci heap: $O(|E| + |V| \log|V|)$

Space Complexity: $O(|V|)$ for storing distances and predecessors

2.1.4. Limitations

Dijkstra's algorithm requires all edge weights to be non-negative. This restriction ensures that the greedy choice (always selecting the vertex with minimum current distance) leads to globally optimal solutions. With negative edges, this greedy property fails, and the algorithm may produce incorrect results.

2.2. Bellman-Ford Algorithm

The Bellman-Ford algorithm, developed by Richard Bellman and Lester Ford Jr., solves the single-source shortest path problem for graphs that may contain negative edge weights. Unlike Dijkstra's algorithm, it can handle negative weights and detect the presence of negative cycles.

2.2.1. Algorithm Description

The Bellman-Ford algorithm uses dynamic programming principles and operates in two phases:

Phase 1 - Relaxation: The algorithm performs $|V| - 1$ iterations, where each iteration relaxes all edges in the graph. This systematic relaxation ensures that shortest paths of length k are found in the k-th iteration.

Phase 2 - Negative Cycle Detection: After $|V| - 1$ iterations, the algorithm performs one additional iteration. If any distance can still be improved, a negative cycle exists.

The relaxation operation is identical to Dijkstra's:

$$d[v] = \min(d[v], d[u] + w(u, v))$$

However, instead of using a priority queue, Bellman-Ford systematically examines all edges in each iteration.

2.2.2. Mathematical Foundation

For a graph $G = (V, E)$ with source s, if no negative cycles are reachable from s, then after $|V| - 1$ iterations:

$$d[v] = \delta(s, v) \quad \forall v \in V$$

This correctness follows from the fact that any shortest path contains at most $|V| - 1$ edges (since it cannot repeat vertices without forming a cycle).

The negative cycle detection works because if a negative cycle exists, distances can be improved indefinitely, violating the convergence property.

2.2.3. Complexity Analysis

Time Complexity: $O(|V| \times |E|)$
- $|V| - 1$ iterations of relaxing all $|E|$ edges
- One additional iteration for cycle detection

Space Complexity: $O(|V|)$ for distance and predecessor arrays

2.2.4. Advantages and Applications

- Handles negative edge weights correctly
- Detects negative cycles reachable from the source
- Simpler implementation than Dijkstra (no priority queue needed)
- Can be easily parallelized or distributed
- Works with any edge ordering

2.3. A* Search Algorithm

A* (pronounced "A-star") is an informed search algorithm that uses heuristics to guide the search toward the goal more efficiently than uninformed algorithms like Dijkstra's algorithm.

2.3.1. Algorithm Description

A* maintains two functions for each vertex v:
- $g(v)$: The actual cost from the start vertex to v
- $h(v)$: The heuristic estimate of cost from v to the goal
- $f(v) = g(v) + h(v)$: The estimated total cost of a path through v

The algorithm uses $f(v)$ to prioritize which vertices to explore, always selecting the vertex with the lowest f-value.

2.3.2. Heuristic Properties

For A* to guarantee optimal solutions, the heuristic function must be:

Admissible: $h(v) \leq h^*(v)$ where $h^*(v)$ is the true cost from v to goal

$$h(v) \leq \delta(v, \text{goal}) \quad \forall v \in V$$

Consistent (Monotonic): $h(u) \leq w(u, v) + h(v)$ for every edge (u, v)

When these properties hold, A* is guaranteed to find optimal solutions while exploring fewer vertices than Dijkstra's algorithm.

2.3.3. Mathematical Formulation

A* maintains the invariant that for any vertex v in the closed set:

$$g(v) = \delta(\text{start}, v)$$

The algorithm terminates when the goal vertex is selected for expansion, ensuring optimality under admissible heuristics.

2.3.4. Complexity Analysis

Time Complexity: $O(b^d)$ where b is the branching factor and d is the depth of the solution
- In practice, much better than this worst-case bound with good heuristics
- Reduces to Dijkstra's complexity with $h(v) = 0$

Space Complexity: $O(b^d)$ for storing the open and closed sets

2.3.5. Grid-Specific Optimizations

For grid-based pathfinding, A* commonly uses:
- **Manhattan Distance**: $h((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_1 - y_2|$
- **Euclidean Distance**: $h((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$

Manhattan distance is admissible for 4-directional movement, while Euclidean distance works for 8-directional or continuous movement.

3. Implementation Details

3.1. Graph Representation

The implementation uses an adjacency list representation stored in a Python dictionary:

```
edges: Dict[int, List[Tuple[int, float]]] = {}
```

This representation provides:
- $O(1)$ vertex addition
- $O(1)$ edge addition
- $O(d)$ neighbor enumeration where d is vertex degree
- Memory efficiency for sparse graphs

3.2. Priority Queue Implementation

Both Dijkstra's algorithm and A* use Python's `heapq` module, which implements a binary min-heap:
- `heappush()`: $O(\log n)$
- `heappop()`: $O(\log n)$
- Space: $O(n)$

3.3. Performance Monitoring

Each algorithm implementation includes comprehensive performance metrics:
- Operation counting for algorithm-specific operations
- Vertex visitation tracking
- Path reconstruction with validation
- Memory usage monitoring

3.4. Test Instance Generation

The benchmarking framework generates diverse test cases:

Random Graphs:
- Erdős-Rényi model with configurable edge probability
- Controllable weight distributions
- Optional negative edges with specified probability

Grid Graphs:
- 2D grids with configurable dimensions
- Random obstacle placement
- Specialized for A* pathfinding evaluation

4. Experimental Results and Analysis

The comprehensive benchmarking study was conducted on 9 different graph types and 8 grid configurations, providing substantial empirical data for algorithm comparison. The experiments reveal significant performance differences and validate theoretical complexity predictions.

4.1. Performance Comparison

4.1.1. Execution Time Analysis

The experimental results demonstrate clear performance hierarchies among the algorithms:

Average Execution Times:
- Dijkstra's Algorithm: 0.000039 seconds
- A* Search (zero heuristic): 0.000033 seconds
- Bellman-Ford Algorithm: 0.000207 seconds
- A* Search (grids): 0.000155 seconds

A* with zero heuristic consistently outperformed Dijkstra's algorithm, despite being theoretically equivalent. This counter-intuitive result likely stems from implementation differences in the priority queue handling and early termination conditions.

Bellman-Ford showed the expected higher execution times, averaging approximately 5 times slower than Dijkstra's algorithm. This aligns with the theoretical complexity difference between $O(V \times E)$ and $O((V + E) \log V)$.

4.1.2. Operations Count Scaling

The operations count analysis reveals the practical impact of algorithmic complexity:

- **Dijkstra and A***: Operations count scales approximately $O(V \log V)$ for sparse graphs
- **Bellman-Ford**: Shows clear $O(V \times E)$ scaling, with operations counts reaching over 1,800 for graphs with 20 vertices

The scatter plots confirm that Bellman-Ford's operation count grows much more rapidly with graph size, particularly for dense graphs where E approaches V^2.

4.1.3. Graph Type Impact

Different graph structures significantly influence algorithm performance:

Sparse Graphs ($|E| \approx 0.1|V|^2$):
- All algorithms perform well
- Dijkstra and A* show minimal differences
- Bellman-Ford remains competitive for small instances

Dense Graphs ($|E| \approx 0.5|V|^2$):
- Performance gaps become more pronounced
- Bellman-Ford's $O(V \times E)$ complexity becomes problematic
- Dijkstra maintains consistent performance

Negative Edge Graphs:
- Only Bellman-Ford can handle these correctly
- All test instances with negative edges contained negative cycles
- Demonstrates the critical importance of negative cycle detection

4.2. Algorithm-Specific Analysis

4.2.1. Dijkstra's Algorithm Performance

Dijkstra's algorithm demonstrated excellent practical performance characteristics:

- **Consistency**: Low variance in execution times across different graph types
- **Scalability**: Operations count grew predictably with graph size
- **Efficiency**: Binary heap implementation proved effective for moderate graph sizes

The algorithm's performance was particularly strong on dense graphs, where the $\log V$ factor in the complexity becomes advantageous compared to Bellman-Ford's linear dependence on edge count.

4.2.2. Bellman-Ford Algorithm Behavior

Bellman-Ford exhibited the expected theoretical behavior:

- **Reliability**: Successfully detected negative cycles in all test cases containing them
- **Correctness**: When no negative cycles existed, produced identical path costs to Dijkstra
- **Scalability Issues**: Operations count grew quadratically with graph density

The path cost correlation plot confirms perfect agreement between Dijkstra and Bellman-Ford on graphs without negative edges, validating both implementations.

4.2.3. A* Search Effectiveness

A* demonstrated interesting performance characteristics across different scenarios:

General Graphs (with zero heuristic):
- Slightly outperformed Dijkstra in execution time
- Identical operation counts, confirming theoretical equivalence
- Benefits from implementation optimizations in the search loop

Grid Pathfinding:
- Manhattan distance heuristic provided significant efficiency gains
- Operations count remained low even for large grids (900 cells)
- Performance degraded gracefully with increasing obstacle density

The grid results show A*'s strength in spatial pathfinding, with execution times growing sublinearly with grid size when good heuristics are available.

4.3. Practical Implications

4.3.1. Algorithm Selection Guidelines

Based on the experimental results, the following guidelines emerge:

Use Dijkstra's Algorithm When:
- Graph has non-negative edge weights
- Seeking optimal performance for single-source shortest paths
- Graph density is moderate to high
- Implementation simplicity is valued

Use Bellman-Ford Algorithm When:
- Graph contains or may contain negative edge weights
- Negative cycle detection is required
- Graph is small to moderate in size
- Distributed computation is needed (Bellman-Ford parallelizes easily)

Use A* Search When:
- Pathfinding between specific source-target pairs
- Good heuristic functions are available
- Working with spatial or grid-based problems
- Early termination benefits are significant

4.3.2. Performance Scaling Considerations

The experimental data reveals critical scaling thresholds:

- For graphs with $|V| < 50$ and moderate density, all algorithms perform acceptably
- Bellman-Ford becomes prohibitively expensive for $|E| > 1000$
- A* with good heuristics scales better than both alternatives for pathfinding tasks

4.3.3. Real-World Application Mapping

- **GPS Navigation Systems**: A* with Euclidean distance heuristics on road networks
- **Network Routing Protocols**: Dijkstra for link-state protocols (OSPF)
- **Game AI Pathfinding**: A* with Manhattan distance on tile-based games
- **Financial Arbitrage Detection**: Bellman-Ford for detecting negative cycles in currency exchange

5. Conclusions

5.1. Summary of Findings

This comprehensive study of three fundamental shortest path algorithms reveals both the theoretical foundations and practical performance characteristics that guide algorithm selection in real applications.

Key Experimental Findings:

1. **Performance Hierarchy**: A* ≈ Dijkstra < Bellman-Ford in execution time for comparable problems

2. **Scalability**: Dijkstra and A* scale well with graph size; Bellman-Ford shows quadratic degradation with density
3. **Correctness Validation**: All algorithms produce identical optimal solutions when applicable
4. **Negative Cycle Handling**: Only Bellman-Ford correctly handles and detects negative cycles

Theoretical Validation:

The experimental results strongly support theoretical complexity predictions. Bellman-Ford's $O(V \times E)$ complexity manifested clearly in the operations count scaling, while Dijkstra's $O((V + E) \log V)$ complexity proved efficient across all test scenarios.

Practical Insights:

The choice between algorithms depends heavily on problem characteristics:
- **Edge weights**: Negative weights necessitate Bellman-Ford
- **Graph structure**: Sparse graphs favor all algorithms; dense graphs favor Dijkstra/A*
- **Query patterns**: Single-pair queries benefit from A*'s heuristic guidance
- **Scale requirements**: Large graphs require Dijkstra or A* for acceptable performance

5.2. Future Work

Several research directions emerge from this study:

Algorithmic Enhancements:
- Implementation of bidirectional search variants
- Integration of contraction hierarchies for large road networks
- Parallel and GPU-based implementations of Bellman-Ford

Extended Benchmarking:
- Analysis of real-world graph structures (social networks, web graphs)
- Performance evaluation on graphs with millions of vertices
- Comparison with modern variants (D* Lite, Jump Point Search)

Heuristic Development:
- Investigation of learned heuristics using machine learning
- Adaptive heuristic selection based on graph properties
- Multi-objective pathfinding with composite heuristics

Theoretical Extensions:
- Analysis of approximate algorithms for faster computation
- Study of dynamic shortest paths with changing edge weights
- Integration with modern graph processing frameworks

This study demonstrates that while theoretical analysis provides fundamental insights, empirical evaluation remains essential for understanding practical algorithm behavior and guiding implementation decisions in real-world applications.

6. References

1. Dijkstra, E. W. (1959). "A note on two problems in connexion with graphs". *Numerische Mathematik*, 1(1), 269-271.

2. Bellman, R. (1958). "On a routing problem". *Quarterly of Applied Mathematics*, 16(1), 87-90.

3. Ford Jr., L. R. (1956). "Network Flow Theory". RAND Corporation.

4. Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). "A formal basis for the heuristic determination of minimum cost paths". *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100-107.

5. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

6. Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley Professional.