Shortest-Path Algorithms

- Sparsh Gupta, Sid Taylor, Mark Belanger

What are they?

Shortest-path algorithms are algorithms used to find a route between a set of connected vertices on a graph that minimizes the distance traveled between them, often by minimizing the edge weights. Shortest-path algorithms have application ranging from modeling games like Rubik's Cubes to finding the quickest way to get from the store to your house.

Algorithms

Dijkstra's

A single-source shortest path algorithm that finds the shortest path from a source node to all other nodes in a non-negative weighted graph.

- **Algorithm Overview**
  - Start at a specified node (e.g., node 0).
  - Find the shortest path to all other nodes in the graph.
  - Save the shortest distance for each node.
  - Mark visited nodes as the algorithm progresses.
  - Continue until all nodes are visited.
- **Calculating Shortest Paths**
  - For each unvisited node, it takes O(V) time to find the shortest path.
  - Constant time (O(1)) for updating minimum path values.
  - For each node, update its neighbors in O(V) time.
  - Overall, time complexity is O(V^2).
- **Applications**
  - Used in GPS devices to find the shortest path between locations.
  - Creates a shortest-path tree, allowing for filtering based on restrictions.

A*

Finds the shortest path in a graph by combining the benefits of Dijkstra's algorithm with a heuristic to efficiently explore promising routes.

- **Algorithm Overview**
  - Start at the initial node and set it as the current node.
  - Expand the current node's neighbors.
  - Calculate two values for each neighbor:
    - G-value: The cost to reach the neighbor from the start node.
    - H-value: The estimated cost to reach the goal from the neighbor (using Manhattan distance heuristic).
  - Calculate the F-value (F = G + H) for each neighbor.
  - Select the neighbor with the lowest F-value as the new current node.
  - Repeat the process until the goal node is reached.
- **Manhattan Distance Heuristic**
  - It estimates the distance from a node to the goal by summing the horizontal and vertical distances, ignoring diagonal movement.
- **Applications and Pros**
  - A* algorithm guarantees the shortest path.
  - Manhattan distance is efficient for grid-based maps.
  - Often used in route planning, puzzle solving, and games.

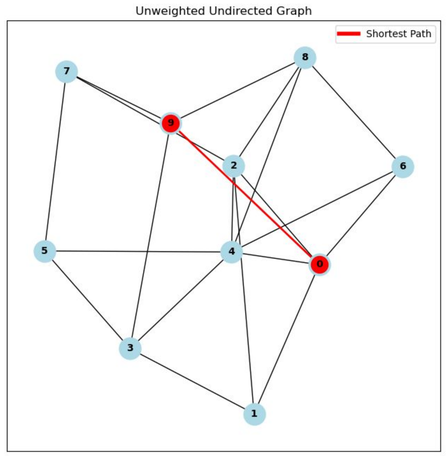
Bellman-Ford

A single-source shortest path algorithm for finding the shortest paths in weighted graphs, including graphs with negative edge weights.

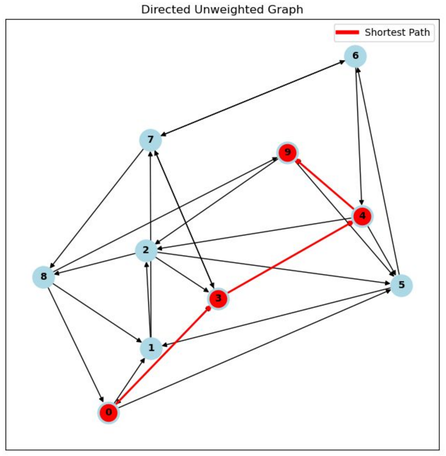
- **Algorithm Overview**
  - Start at a specified source node.
  - Initialize the shortest distance to all nodes as infinity except the source node (set its distance to 0).
  - Iterate through all edges in the graph and relax (update) distances.
  - Repeat the process (edge relaxation) for V-1 times (V is the number of nodes).
  - Detect negative cycles (if any) during the Vth iteration.
- **Key Features**
  - Handles graphs with negative edge weights.
  - Relaxes edges iteratively to find the shortest paths.
  - Detects negative cycles, which may indicate no shortest path.
- **Use Cases**
  - Pathfinding in networks with varying edge weights.
  - Detecting negative cycles in financial modeling or network analysis.

Our Testing

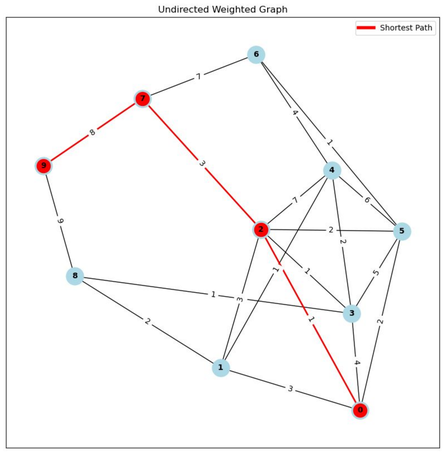
Many different shortest-path algorithms exist, each with different efficiencies and use cases. We explored several different algorithms to measure how efficient each algorithm was. The algorithms were implemented using C, and Python was used to visualize the graphs and the shortest path to our goal vertex. The algorithms were applied to four different types of graphs. For maintaining consistency, our goal vertex was set to vertex 9 for each graph and each graph had a total of 10 vertices (from vertex 0 to vertex 9). The algorithms would start the path at vertex 0 and find the minimum distance path to vertex 9.



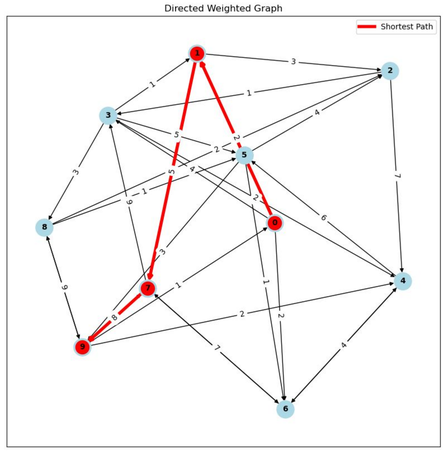
Example of shortest-path on a undirected unweighted graph



Example of shortest-path on a directed unweighted graph



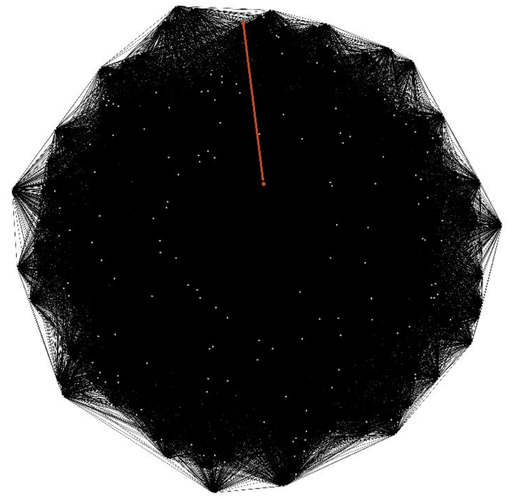
Example of shortest-path on a undirected weighted graph



Example of shortest-path on a directed weighted graph

Network Graph Example

We wanted to determine if our results on smaller graphs actually represent how runtimes would look like on a more bigger, complex, and a real-example graph. However, real-example graphs publicly available contain thousands of edges and nodes and it is not feasible to computationally look at their shortest-path runtimes as they are tougher to visualize, understand and are very sparse in nature so there are not necessarily connections from one vertex to another. Therefore, we decided to generate a random directed-weighted connected graph with 200 vertices and test our algorithms on it. The start vertex was 0 and the goal vertex was randomly selected to be 186, and the algorithms were run to find the shortest path between these vertices. The results correlate to the results on the smaller graphs as the runtime for A* is the least and Bellman-Ford is the worst. The graph can be seen below (node labels omitted for visualization purposes and the red line indicates the shortest path):



Results

The results for the algorithms run on four different kinds of graphs could be interpreted from the Results Table below. The results clearly indicate that A* is the fastest algorithm out of the three tested in this project which seems to be reasonable because A* algorithm guarantees the shortest path while improving the Dijkstra's algorithm by introducing a cost function to make better path choices. It also uses the least time and space complexity. In terms of time complexity, Bellman-Ford is relatively the worst but it has a benefit that it can work with negative edge weights. In terms of space complexity, Dijkstra conducts a blind scan and therefore takes up a lot of memory compared to A* and Bellman Ford, however it has a reasonable time complexity.

Results Table

Algorithm	**Time Complexity**	**Space Complexity**	**Runtime (on graphs)**				
			Undirected Unweighted	Directed Unweighted	Undirected Weighted	Directed Weighted	Network Graph (Directed Weighted)
Dijkstra's	O(V^2)	O(V^2)	0.009 ms	0.008 ms	0.008 ms	0.011 ms	0.335 ms
A*	O(V + E)	O(V)	0.001 ms	0.001 ms	0.002 ms	0.003 ms	0.015 ms
Bellman-Ford	O(V*E) = O(V^3)	O(V)	0.009 ms	0.009 ms	0.010 ms	0.011 ms	18.411 ms