COMPSCI 2XC3: WINTER 2024
FINAL PROJECT

Jenil Maru - maruj
Saad Khalid - khalis68
Pritha Saha - sahap9

Exploring Efficiency: An Experimental Study Of Shortest Path Algorithms And UML-Based Code Organization

INDEX

Part 1

Functions In 1.1(Dijkstra's Algorithm):

Time Complexity: The time Complexity of Dijkstra's Algorithm is O(V^2) but with a minpriority queue it drops down to O((V + E)log V). Here, The time complexity of Dijkstra's algorithm is O((V + E) log V), where V is the number of vertices and E is the number of edges. In the provided code, the modification with parameter k increases the time complexity, especially when k is close to V.

Space Complexity: The space complexity is O(V) for maintaining the priority queue and O(E) for the adjacency list.

Accuracy: Dijkstra's algorithm is accurate for non-negative edge weights, which is noticeable in the code.

Functions In 1.2(Bellman-Ford Algorithm):

Time Complexity: The time complexity of the Bellman-Ford algorithm is O(VE). It's less efficient compared to Dijkstra's algorithm, especially for dense graphs.

Space Complexity: Similar to Dijkstra's algorithm, the space complexity is O(V).

Accuracy: Bellman-Ford works correctly even with negative edge weights, but it's slower than Dijkstra's algorithm for non-negative weights.

Impact of Factors:

Graph Size: Both algorithms are affected by the size of the graph. As the number of vertices and edges increases, the execution time of both algorithms also increases.

Graph Density: Dense graphs have more edges, leading to longer execution times for both algorithms. Sparse graphs with fewer edges have shorter execution times.

Value of k: Higher values of k in the modified Dijkstra's algorithm increase the execution time, especially for large graphs. It's essential to choose an appropriate value of k based on the graph size and density to balance accuracy and performance.

Performance Comparison:

- Dijkstra's algorithm tends to perform better for graphs with non-negative edge weights, especially when the graph is sparse or when the value of k is relatively small.

- Bellman-Ford is suitable for graphs with negative edge weights but becomes inefficient for large graphs due to its higher time complexity.

- The code measures the execution time for both algorithms for different graph sizes, providing a practical comparison of their performance as we can see in the graph plot below.

1.3

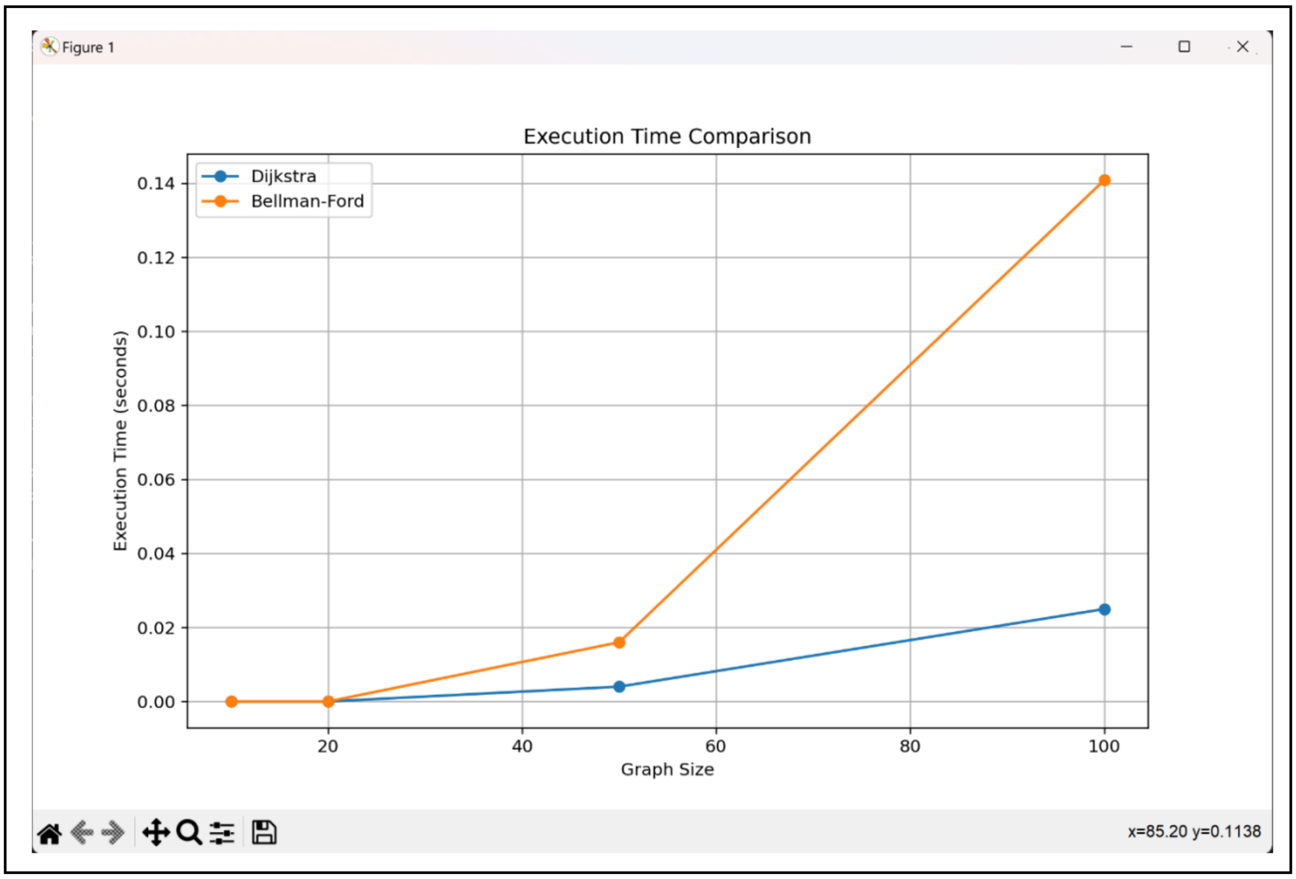
Figure 1
Execution Time Comparison
Dijkstra
Bellman-Ford
Execution Time (seconds)
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
20
40
60
80
100
Graph Size
x=85.20 y=0.1138

Part 2

MAIN TAKEAWAY:

The code defines a minimum heap structure and utilizes it within Dijkstra's algorithm to efficiently find the shortest paths from a single source to all other nodes in a graph with non-negative weights, repeating this process for each vertex to solve the all-pairs shortest path problem. For graphs that might have negative weights, it employs the Bellman-Ford algorithm to achieve the same goal, carefully checking for negative-weight cycles which would invalidate the shortest path calculations. Both algorithms return a matrix of shortest distances and a matrix of predecessors for path reconstruction.

The complexity of the all-pairs shortest path algorithms for dense graphs can be derived from the complexities of the single-source shortest path algorithms, Dijkstra's and Bellman-Ford, since they are effectively being run from each vertex in the graph.

For Dijkstra's algorithm, using a binary min-heap, the complexity is (O(E + Vlog V)) for each run from a single source. In a dense graph, where (E) is close to (V^2), this complexity becomes (O(V^2 + Vlog V)), which simplifies to (O(V^2)) for each run from a single source.

Since you need to run Dijkstra's algorithm from each of the (V) vertices for the all-pairs shortest path problem, the overall complexity for Dijkstra's algorithm in a dense graph becomes (O(V times V^2)), which simplifies to (O(V^3)).

For the Bellman-Ford algorithm, the complexity is (O(VE)) for each run from a single source. In a dense graph, where (E) is close to (V^2), this complexity is (O(V . V^2)), which is (O(V^3)) for each run from a single source.

Since you need to run the Bellman-Ford algorithm from each of the (V) vertices for the all-pairs shortest path problem, the overall complexity for Bellman-Ford in a dense graph becomes (O(V times V^3)), which simplifies to (O(V^4)).

Therefore, for dense graphs, the complexity of the provided algorithms would be:

- For all-pairs shortest path using Dijkstra's algorithm: (O(V^3)).
- For all-pairs shortest path using Bellman-Ford algorithm: (O(V^4)).

This conclusion holds if we assume that the graph is dense and the priority queue operations in Dijkstra's algorithm scale with (O(V^2)) due to the high number of edges, which is typical for dense graphs.

Part 3

3.2

The A* algorithms tries to address the issue with Dijkstra's algorithm by using a heuristic to estimate the cost of the path from the current node to the goal node. The algorithm uses a priority queue to keep track of the nodes that need to be visited. Rather than visiting all nodes and considering them to be equal candidates for the shortest path, A* uses the heuristic to prioritize nodes that are closer to the goal node. This allows the algorithm to converge faster and find the shortest path more efficiently.

We would test djikstra's algorithm on the same graph and compare the results with the A* algorithm. The graph in question would need to be designed in such a way that the heuristic value is somewhat useful in guiding the search.

If we were to use arbitrary (random) heuristic values, the A* algorithm would perform about as well as Dijkstra's algorithm. In this case the negative effects of the heuristic would cancel out the benefits (assuming that the possuble heuristic values are uniformly distributed). Since we visit neighbors using a composite weighting of the edge weight and the heuristic value, on average the edge weighting would dictate the search path, which is what we use in djikstra's algorithm.

A* should be used in applications where we can predictably and reliably assign heuristic values to the nodes in the graph. This is often the case in pathfinding applications where we have a good idea of the distance between two points.

Part 4

There is a slight performance advantage to using A* over Dijkstra's algorithm. this is because we're using the heuristic function to better select which nodes should be visited first. However for this dataset, the difference is nominal. This is likely because the nodes with the shortest path are close to each other, and the heuristic function doesn't provide much of an advantage. If the nodes were further apart, we would expect to see a larger difference between the two algorithms, only about 20% of pairs of nodes require a line change, other pairs are reachable by the same line.

Part 5

In the UML diagram, we have an inhieiarchy of classes, with the Graph class at the top. The Graph class has two subclasses, WeightedGraph and HeuristicGraph. The WeightedGraph class has a subclass called HeuristicGraph. The Graph class has a method called get_all_nodes, which returns a list of all nodes in the graph. The Graph class also has a method called get_adj_nodes, which returns a list of all adjacent nodes to a given node. The Graph class has a method called add_node, which adds a node to the graph. The Graph class has a method called add_edge, which adds an edge between two nodes. The Graph class has a method called get_num_of_nodes, which returns
the number of nodes in the graph. The Graph class has a method called w, which returns the weight of an edge between two nodes.

We also implement functions which depend on instances of the graph class, djiastra and a_star. These functions implement the djikstra and a_star algorithms respectively. The djikstra function finds the shortest path between two nodes.

Since we use the int representation of the node as they key, we would need to be able to hash the new type of the node, we could do this by defining a __hash__ method for the Station class. We would also need to define an __eq__ method for the Station class, so that we can compare two Station objects for equality. We need the value to be hashable since we use it as a key in a dictionary. We need the __eq__ method so that we can compare two Station objects for equality. We need to be able to compare two Station objects for equality so that we can check if a node.

The graph class could be implemented by network graphs, machine learning compilers for dependency graphs, databases for graph querying functionality, etc. Since the graph class implements weights, and heuristics, it could be used in a variety of applications where we need to find the shortest path between nodes.