B503 Algorithms Design and Analysis - Project Report

Group Members: Mildred Noronha, Prajna Peters

Project Topic: Computational Experiments on Shortest Path Algorithms

Project Report prepared by **Mildred Noronha**

Abstract

This project deals with performing computational experiments on shortest path algorithms, namely, Dijkstra's shortest path algorithm and Bellman-Ford shortest path algorithm. The results of these experiments are compared, and conclusions are drawn on them and presented in this report.

1. Introduction

Computing shortest paths in graphs is one of the most fundamental and well-studied problems in combinatorial optimization. There are a great number of real-world applications that come from figuring out the shortest path in a set of connected nodes. This is done more easily, if there are weights associated with each path, so as to determine the path with the lowest cost from a source to a destination. Some well-known applications are, finding routes in road and public transportation networks, determining routing schemes for computer networks (transmission time of messages is less when they are sent through a short sequence of routers) and social media graphs which show connections between various people or objects. [1]

Determining the shortest path in a network being beneficial to a variety of real-world applications, we have chosen this topic to perform computational experiments on. We have chosen two well-known shortest path algorithms, Dijkstra's shortest path algorithm and Bellman-Ford's shortest path algorithm. Both Dijkstra's and Bellman-Ford's algorithms are single-source shortest path algorithms. This is one type of shortest path algorithms, wherein the others are single-pair shortest path, single-destination shortest path and all-pairs shortest path algorithms. [2]

This project report will compare the performance of these two algorithms on varying input and will report the results. Dijkstra's algorithm will be implemented by Prajna Peters, Bellman-Ford algorithm by Mildred Noronha and Erdős–Rényi random graph generator by Prajna Peters and Mildred Noronha.

1.1 Dijkstra's Algorithm

Dijkstra thought about the shortest path problem when working at the Mathematical Center in Amsterdam in 1956 as a programmer to demonstrate capabilities of a new computer called ARMAC. His objective was to choose both a problem as well as an answer (that would be produced by computer) that non-computing people could understand. He designed the shortest path algorithm and later implemented it for ARMAC for a slightly simplified transportation map of 64 cities in the Netherlands. [3]

A brief description of Dijkstra's algorithm is as follows:

Let the node at which we are starting be called the initial node. Let the distance of node *Y* be the distance from the initial node to *Y*. Dijkstra's algorithm will assign some initial distance values and will try to improve them step by step.

1. Assign to every node a tentative distance value: set it to zero for our initial node and to infinity for all other nodes.
2. Set the initial node as current. Mark all other nodes unvisited. Create a set of all the unvisited nodes called the *unvisited set*.
3. For the current node, consider all of its unvisited neighbors and calculate their *tentative* distances. Compare the newly calculated *tentative* distance to the current assigned value and assign the smaller one. For example, if the current node *A* is marked with a distance of 6, and the edge connecting it with a neighbor *B* has length 2, then the distance to *B* (through *A*) will be 6 + 2 = 8. If B was previously marked with a distance greater than 8 then change it to 8. Otherwise, keep the current value.
4. When we are done considering all of the neighbors of the current node, mark the current node as visited and remove it from the *unvisited set*. A visited node will never be checked again.
5. If the destination node has been marked visited (when planning a route between two specific nodes) or if the smallest tentative distance among the nodes in the *unvisited set* is infinity (when planning a complete traversal; occurs when there is no connection between the initial node and remaining unvisited nodes), then stop. The algorithm has finished.
6. Otherwise, select the unvisited node that is marked with the smallest tentative distance, set it as the new "current node", and go back to step 3. [4]

Time complexity of Dijkstra's is O(V^2). [5] A more detailed analysis of this will be provided in Prajna Peters' report.

1.2 Bellman-Ford Algorithm

The Bellman-Ford algorithm was first proposed by Alfonso Shimbel in 1955, but is instead named after Richard Bellman and Lester Ford, Jr., who published it in 1958 and 1956, respectively. Edward F. Moore also published the same algorithm in 1957, and for this reason it is also sometimes called the Bellman–Ford–Moore algorithm. [6]

The Bellman–Ford algorithm is an algorithm that computes shortest paths from a single source vertex to all of the other vertices in a weighted digraph. [6] This algorithm can work with negative edge weights unlike Dijkstra's algorithm.

We start with the following setup:

- The starting node has cost 0, as his distance to itself is obviously 0.
- All other nodes have cost infinity

During each iteration, the following is checked: Are the cost of the source of the edge plus the cost for using the edge smaller than the cost of the edge's target?

If this is false, we can use the edge with the smaller cost i.e. the edge's target rather than the sum of source of the edge plus the cost for using the edge. In this way, we will compute the path from the source to one of the edges.

It is not sufficient to look at all edges only once. After the first iteration, the cost of all nodes for which the shortest path only uses one edge have been calculated correctly. After two phases all paths that use at most two edges have been computed correctly, and so on.
Since we consider the paths of all nodes from the source, the algorithm runs V-1 times, where V is the number of nodes(vertices).

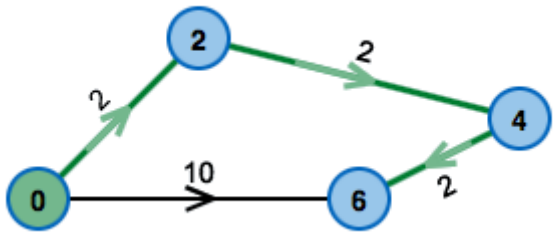

Figure 1:The green path from the starting node is the cheapest path. It uses 3 edges.

A negative circle can be reached if and only if after iterating all phases, one can still find a short-cut. Therefore, at the end the algorithm checks one more time for all edges whether the cost of the source node plus the cost of the edge are less than the cost of the target node. Bellman-Ford can identify such cycles and they can be removed for further computation.

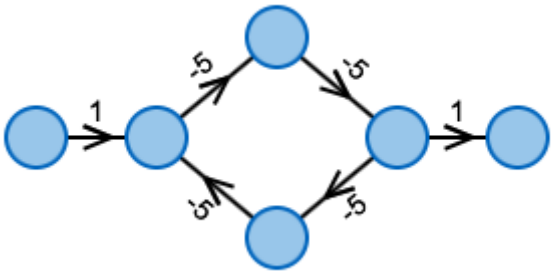

Figure 2:A cheapest path had to use this circle infinitely often. The cost would be reduced in each iteration.

We take the following algorithm into consideration where input is a graph G and a source vertex src. Output to this is the shortest distance to all vertices from src. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported. [7]

1) Initialize array distances from source to all vertices as infinite and distance to source itself as 0. Create an array distances[] of size |V| with all values as infinite except distances[src], where src is the source vertex.

2) Calculate shortest distances:

Compute the following |V|-1 times where |V| is the number of vertices in given graph.

a) Do following for each edge u-v:

If dist[v] > dist[u] + weight of edge u-v, then update dist[v]

dist[v] = dist[u] + weight of edge u-v

3) Reports if there is a negative weight cycle in graph. Do following for each edge u-v

If dist[v] > dist[u] + weight of edge uv,

then print “Graph contains negative weight cycle”

The idea of step 3 is, step 2 guarantees shortest distances if graph doesn’t contain negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle

We know that the algorithm gives the shortest path from source for every vertex in step 2. Step 3 is carried out to determine negative weight cycle. A negative weight cycle exists if there is a change in the distances array in step 3. [8]

The initialization in step 1 takes Θ(V) time, step 2 runs the loop (|V|-1).E times and step 3 takes Θ(E) time. Thus, we can say that the time complexity of Bellman-Ford is O(VE). [5]

2. *Experimental Setup*

Now that we have seen an overview of the two algorithms, we will see the procedure to implement the two algorithms and the data we will use as input.

Shortest path algorithms take graphs as input. Here, we will use a random graph generated using the Erdős–Rényi model of random graphs. The Erdős–Rényi model is implemented in MATLAB and takes the number of vertices, probability of each edge being present and the degree of each node as input. G(n,p) is really a probability distribution over the set of all possible graphs on n vertices, [] where G is the graph generated using Erdős–Rényi model, n is the number of vertices and p is the probability of the presence of each edge.

The output from the Erdős–Rényi model generator is a graph structure containing information about the presence of edge in the graph in an adjacency matrix, the x and y co-ordinates of each node, number of vertices and the number of edges in the graph.
The adjacency matrix from the graph structure can be given to the shortest path algorithms for further computation.

Bellman-Ford algorithm takes the number of vertices and the adjacency matrix as input. As mentioned before, the algorithm computes the shortest path from the source vertex to every other vertex and stores all distances in the *distances* matrix, over V-1 iterations, where V is the number of vertices. One final run-through all edges is done to check if negative weight cycle exists.

3. *Results*

We are comparing the time complexities of the two algorithms by varying the probabilities of edges in the graph generated by Erdős–Rényi model. We have taken a constant of 10 vertices in all graphs, and changed the probabilities from 0.1 to 0.9, incrementing by 0.2 every time.

The run-times of both Dijkstra's and Bellman-Ford have been shown below.

Probability	Edges	Dijkstra Runtime (ns)	Bellman-Ford Runtime (ns)
0.1	38	52,715	486,102
0.3	40	56,447	504,763
0.5	44	55,048	515,959
0.7	48	56,914	514,559
0.9	50	55,981	511,761

Table 1: Runtime comparisons of Dijkstra's and Bellman-Ford algorithms

Following are the graphs plotting the run-times of the two algorithms with different probabilities.

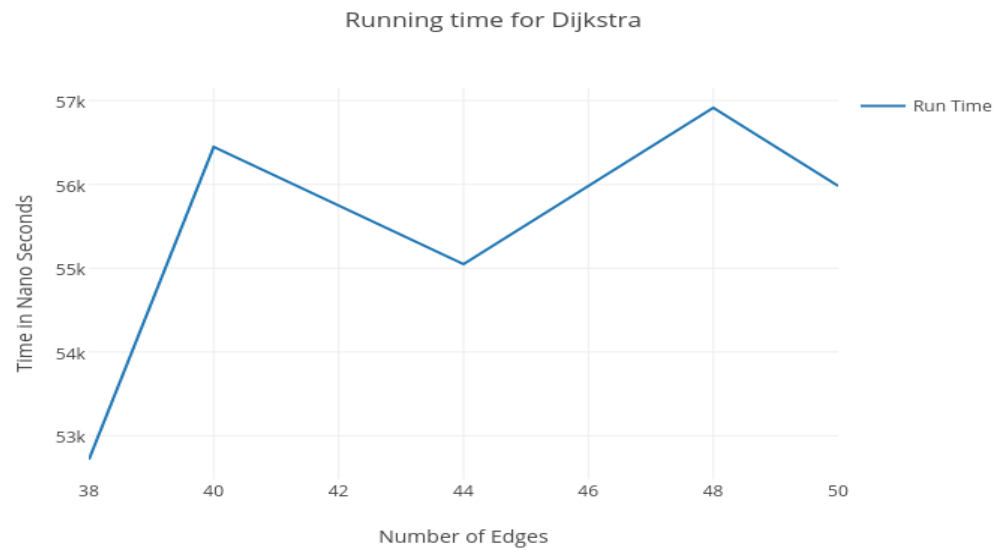

Figure 3: Running time for Dijkstra's algorithm (Time (nanosec) vs. Number of Edges)

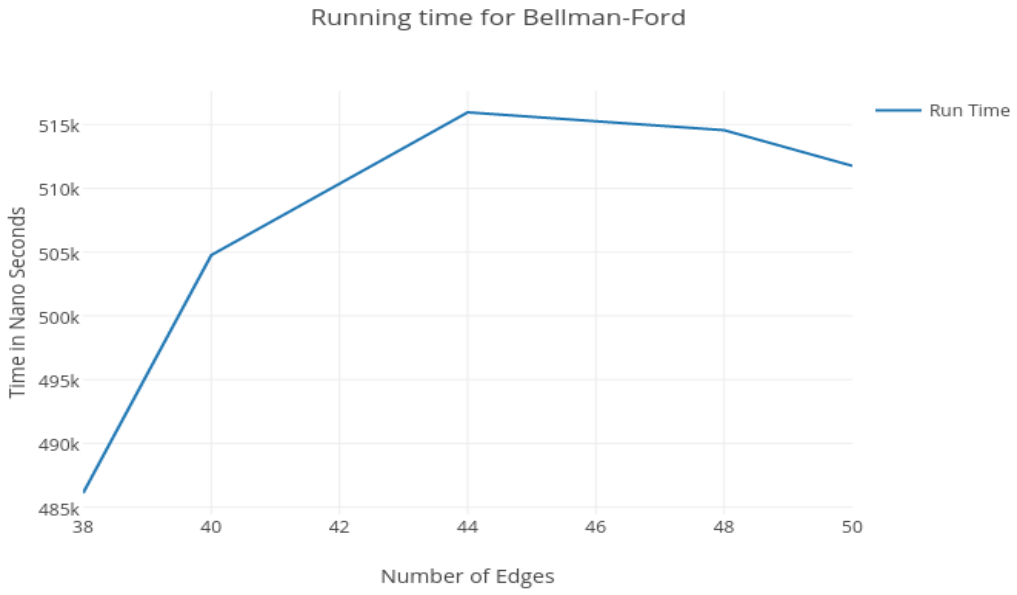

Figure 4: Running time for Bellman-Ford's algorithm (Time (nanosec) vs. Number of Edges)

4. *Conclusions*

From the above graphs, we see that Dijkstra's algorithm is much faster than the Bellman-Ford algorithm for all the cases. It runs ten times faster than Bellman-Ford. This is expected as Dijkstra's algorithm is known to be better and faster than Bellman-Ford. This report demonstrates a large difference in their run times. This difference in their run-times can also be attributed to the style of code-writing, computer specifications, etc.

The Bellman-Ford algorithm implemented in this is very slow in comparison to Dijkstra. A faster version of Bellman-Ford called Shortest Path Faster Algorithm exists wherein the working is very similar to the former, except that SPFA maintains a queue of candidate vertices and adds a vertex to the queue only if that vertex is relaxed. This is repeated till no more edges can be relaxed. [9]

It is advisable to use Dijkstra's algorithm when we know that all edge weights are positive. However, for negative edge weights, we recommend that Bellman-Ford algorithm should be used. For large number of vertices, we can use a faster variation of Bellman-Ford.

5. *References*

[1] http://www.sommer.jp/thesis.htm
[2] https://en.wikipedia.org/wiki/Shortest_path_problem
[3] https://en.wikipedia.org/wiki/Dijkstra's_algorithm#History
[4] https://en.wikipedia.org/wiki/Dijkstra's_algorithm#Algorithm
[5] Cormen, T.H. ;Leiserson, C.E.;Rivest R.L.;Stein C. Introduction to Algorithms, MIT Press & McGraw-Hill.
[6] https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm
[7] https://www-m9.ma.tum.de/graph-algorithms/spp-bellman-ford/index_en.html
[8] http://www.geeksforgeeks.org/dynamic-programming-set-23-bellman-ford-algorithm/
[9] https://en.wikipedia.org/wiki/Shortest_Path_Faster_Algorithm
[10] Code: https://drive.google.com/open?id=0B3dpWdHmAn-HYi1MUlhUVmJfZm8