CS430: Project Assignment

Title:

Algorithms, implementations and asymptotic behavior for the shortest path problem.

Team Members:

Shivanshu Misra A20279849
Saurabh Katkar A20320336

Table of Contents

Project Objectives

The main purpose of this project assignment is as follows,

- Implementation of a Single Source Shortest Path algorithm i.e. Djikstra's Algorithm using C++ technology.
- Implementation of Bellman-Ford Algorithm using Java technology.
- Analysis of asymptotic behavior and performance by calculating the time complexity of the algorithm and testing the program on different test cases i.e. Number of vertices '*N*'.
- Compare and contrast with that of the aforementioned shortest path algorithms by examining the derived test cases and the optimality of their performance is measured.
- List the applications of algorithms.

Shortest Path Algorithm

In graph theory, the shortest path problem is the problem of finding a path between two vertices in a graph such that the sum of the weights of its constituent edges is minimized.
In shortest-paths problem, we are given a weighted, directed graph G = (V, E), with the weight function $w : E \rightarrow R$ mapping edges to real valued weights. The weight *w(p)* of path p = {v_0, v_1, ... ,v_k}is the sum of weights of its constitute edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Similarly, we define the shortest-path weight $\Delta(u, v)$ from *u* to *v* by

$$\Delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightarrow} v\} & \text{if there is a path from u to v} \\ \infty & \text{otherwise} \end{cases}$$

A shortest path from vertex u to vertex v is then defined as any path p with weight w(p) = $\Delta(u, v)$.
This problem has following variations,

1. **Single-Source Shortest Path**: Find shortest paths from a source vertex v to all other vertices in the graph. **Examples**: *Djikstra's algorithm, Bellman-Ford algorithm*.
2. **Single-Destination Shortest Path Problem**: Find shortest paths from all vertices in the directed graph to a single destination vertex v.
3. **All-Pairs Shortest Path Problem**: Find shortest paths between every pair of vertices v, v' in the graph. **Examples**: *Floyd-Warshall algorithm, Johnson's algorithm*.

In our project we have only implemented Single-Source Shortest Path problem using Bellman-Ford and Djikstra's algorithm.

Djikstra's Algorithm and Implementation

Djikstra's Algorithm solves the single-source shortest path problem on a weighted directed graph G = (V, E) where all edge weights are non-negative. Thus we assume that $w(u, v) >= 0$ for each edge $(u, v) \epsilon E$. In this algorithm, we maintain two sets, one set contains vertices included in shortest path tree, other set includes vertices not yet included in shortest path tree. At every step of the algorithm, we find a vertex which is in the other set (set of not yet included) and has minimum distance from source.

Detailed explanation of implemented algorithm for given graph,

- Create a boolean set *short_dist_set[N]* which keeps track of vertices included in shortest path tree, i.e. whose minimum distance from source is calculated and finalized. Initially, this set is empty by assigning it as false.

  short_dist_set[i] = false;

- Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE.

  short_dist[i] = INT_MAX

  Assign distance value as 0 for the source vertex so that it is picked first.

  short_dist[source] = 0;

- After initialization in function *Djikstra_Algorithm(int Tree[N][N], int source)*, where *Tree[N][N]* as input parameter which is a *N * N* matrix representing the directed graph, we

further call function i.e. *Min_Dist_Vertex(short_dist, short_dist_set)* to calculate minimum distance between vertices. The input parameters for this function are minimum distance from source and shortest path tree set.

- In function *Min_Dist_Vertex(short_dist, short_dist_set),* which is called *N-1* times while *short_dist_set* doesn't include all vertices and , *short_dist* is less than maximum value, return the *index* which has minimum distance value.
- Include the returned value *min_value* in shortest path tree set,

  short_dist_set[min_value] = true;
- Update distance value of all adjacent vertices of *min_value.* To update the distance values, iterate through all adjacent vertices. For every adjacent vertex *t*, if sum of distance value of *min_value* (from source) and weight of edge *min_value* - *t*, is less than the distance value of *t*, then update the distance value of *t*.

```
if(!short_dist_set[t] && Tree[min_value][t] && short_dist[min_value] != INT_MAX
        && short_dist[min_value]+Tree[min_value][t] < short_dist[t])
    {
       short_dist[t] = short_dist[min_value] + Tree[min_value][t];
    }
```

- Print the result with all vertices and their shortest distance from the source node in function *Print_Result(short_dist, N);*.

Analysis of Asymptotic Behavior of Djikstra's Algorithm

Dijkstra's algorithm maintains the min-priority queue Q by calling three priority-queue operations:

INSERT(implicit)
EXTRACT-MIN and
DECREASE-KEY(implicit in RELAX).

The algorithm calls both INSERT and EXTRACT-MIN once per vertex. Because each vertex is added to set S exactly once, each edge in the adjacency list is examined in for loop exactly once during the course of the algorithm. Since the total number of edges in all the adjacency lists is |E| , this for loop iterates a total of |E| times, and thus the algorithm calls DECREASE-KEY at most |E| times overall.

Variations
The bottleneck that determines the overall complexity of the algorithm is the node selection. Here are three different approaches to select nodes. For a given graph G, let E represent the number of edges in G and V be the number of nodes/vertices in G.

a. Linear Search

One approach that can be used to find the closest distance is by using a linear search. Provided an array with all the valid distances to the neighbors, a single loop can be used to determine the node with the minimum distance from the source node. Since this linear search will be applied to each iteration until the destination node is found, the linear search (O(V)) could be applied for each node (O(V)) and we must examine the edges when determining the minimum; therefore, the overall complexity of using linear search for node selection is $\boldsymbol{O(V^2 + E) = O(V^2)}$

b. Binary Heap

Another approach that can determine the closest node from the source node is through the use of a minimum binary heap. Inserting and deleting nodes from the binary heap takes O(log V) time. The algorithm inserts each edge when computing the minimum distances from to the neighboring nodes and deletes a node from the heap after selecting the closest node from the heap. Finding the minimum distance from the binary heap is O(1), overall complexity of Dijkstra's Algorithm with a binary heap is $\boldsymbol{O((E + V) \log V)}$.

c. Fibonacci Heap

Another data structure that can be used for node selection is the Fibonacci Heap. This structure is a heap that can insert nodes and find the minimum distance in amortized constant time O(1) and delete nodes in O(log V) [4]. Because we only need to delete nodes when we are pulling the closest node from the heap, this deletion will be applied at most V times. When updating the distances to all the neighbor nodes, we will be inserting E edges into the heap. From this, Dijkstra's Algorithm with a Fibonacci heap will take ***O(E + V log V)*** time.

In our implementation, we have selected nodes using *Linear Search* approach thus resulting in time complexity as $\boldsymbol{O(V^2 + E) = O(V^2)}$.

The total running time complexity of our Djikstra's algorithm for varied number of vertices i.e *N* is as below,

Number of Vertices (N)	**Number of Runs (n)**	**Time Complexity (T_N) in msec**	**Average Time Complexity [($\sum T_N$) / n]**
5	1	156	157.67
	2	158	
	3	159	
6	1	210	168.33
	2	151	
	3	144	
9	1	182	188.67
	2	220	
	3	164	
15	1	294	219.67
	2	173	
	3	192	

The graphical representation of running Time Complexity of Djikstra's Algorithm as below in Fig. 1

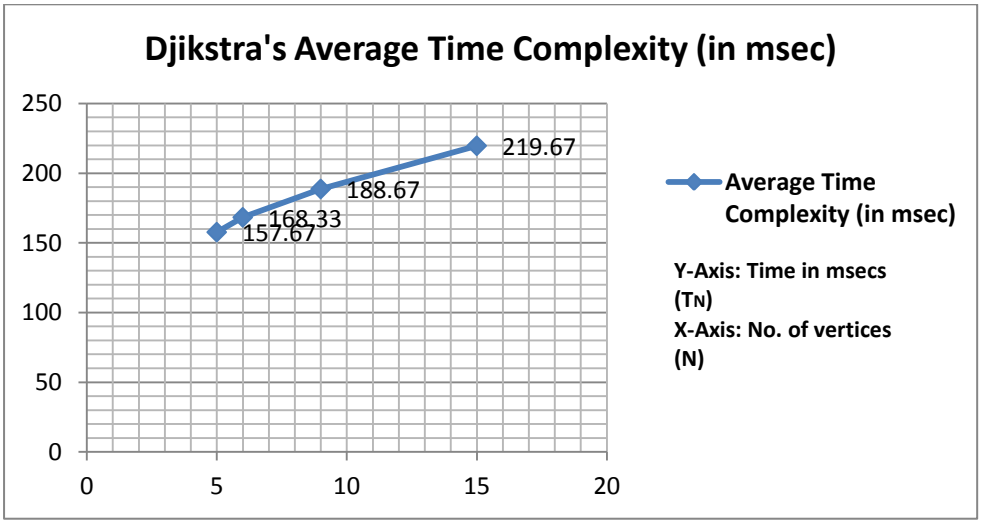

Figure 1. Djikstra's Algorithm Average Time Complexities for varied value of N.

Bellman-Ford Algorithm and Implementation

The Bellman-Ford algorithm uses relaxation to find single source shortest paths on directed graphs that may contain negative weight edges. The algorithm will also detect if there are any negative weight cycles (such that there is no solution).
Negative edge weights are found in various applications of graphs, hence the usefulness of this algorithm. If a graph contains a "negative cycle" (i.e. a cycle whose edges sum to a negative value) that is reachable from the source, then there is no *cheapest* path: any path can be made cheaper by one more walk around the negative cycle. In such a case, the Bellman–Ford algorithm can detect negative cycles and report their existence.

The Bellman-Ford algorithm is as follows:

```
BELLMAN-FORD(G,w,s)
1.  INITIALIZE-SINGLE-SOURCE(G,s)
2.  for i = 1 to |G.V|-1
3.     for each edge (u,v) ∈ G.E
4.        RELAX(u,v,w)
5.  for each edge (u,v) ∈ G.E
6.     if v.d > u.d + w(u,v)
7.        return FALSE
8.  return TRUE

INITIALIZE-SINGLE-SOURCE(G,s)
1.  for each vertex v ∈ G.V
2.     v.d = ∞
3.     v.pi = NIL
4.  s.d = 0

RELAX(u,v,w)
1.  if v.d > u.d + w(u,v)
2.     v.d = u.d + w(u,v)
3.     v.pi = u
```

The workflow of the algorithm as follows:

i. Initialize d's, π's, and set $s.d = 0$.
ii. Loop $|V|$-1 times through all edges checking the relaxation condition to compute minimum distances.
iii. Loop through all edges checking for negative weight cycles which occurs if any of the relaxation conditions fail.

Analysis of Asymptotic Behavior of Bellman-Ford Algorithm

The process of initializing d's parents and setting d to 0 takes O(V) time. Looping through the edges and computing minimum distances takes ($|V|$-1) O(E) = O(VE) time and looping through all edges to check for negative weight cycles takes O(E) time. Thus the run time of the Bellman-Ford algorithm is **O(V + VE + E) = O(VE).**

If the graph is a DAG (and thus is known to not have any cycles), we can make Bellman-Ford more efficient by first topologically sorting G (O(V+E)), performing the same initialization (O(V)), and then simply looping through each vertex u in topological order relaxing only the edges in Adj[u] (O(E)). This method only takes **O(V + E)** time.

The total running time complexity of Bellman-Ford algorithm for varied number of vertices (N) is as below,

Number of Vertices (N)	Number of Runs (n)	Time Complexity (T_N) in msec	Average Time Complexity [($\sum T_N$) / n]
5	1	7608	6710
	2	6713	
	3	5809	
6	1	6913	7831
	2	9534	
	3	7046	
9	1	7236	8564
	2	8654	
	3	9802	
15	1	7204	8787.67
	2	9223	
	3	9936	

The graphical representation of running Time Complexity of Bellman-Ford Algorithm as below in Fig. 2

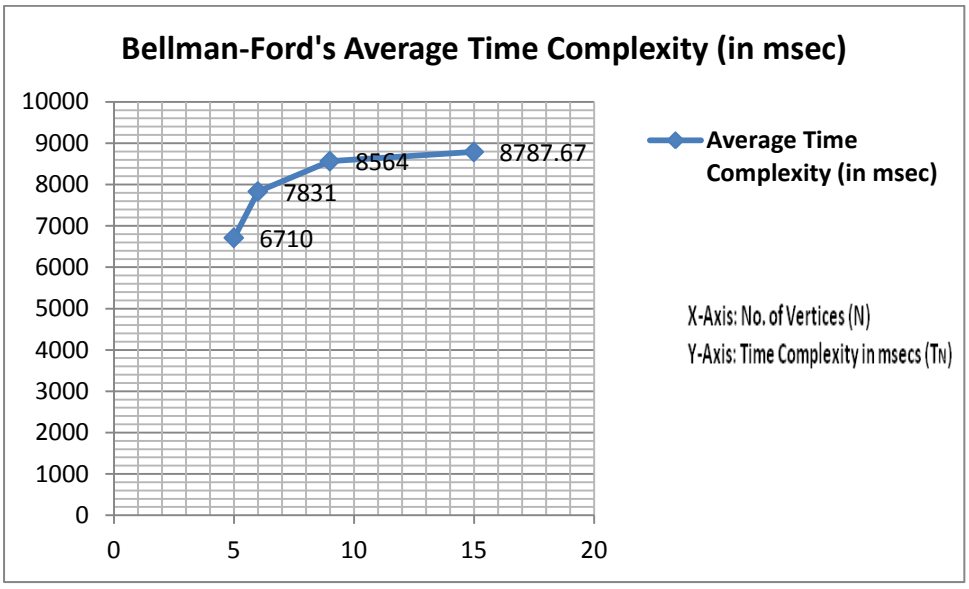

Fig. 2: Bellman-Ford Algorithm Average Time Complexities for varied value of N.

Comparison and Contrast of Performance

Though, both of these functions solve the single source shortest path problem. The primary difference in the function of the two algorithms is that Dijkstra's algorithm cannot handle negative edge weights. Bellman-Ford's algorithm can handle some edges with negative weight. However, if there is a negative cycle there is no shortest path.
In Bellman-Ford algorithm, detection of a negative cycle is done by looping through all edges checking for negative weight cycles which occurs if any of the relaxation conditions fail. This additional loop thus increases the running time of the algorithm. Hence for a typical implementation with binary heap, Dijkstra's algorithm has **O((|E|+|V|)log|V|)** time complexity, while Bellman-Ford algorithm has **O(|V||E|)** complexity.

In both cases however, if there are more than 1 path that has minimum cost, the actual path returned is implementation dependent.

The below graph Fig.3 shows the comparison of running time-complexities of Djikstra's and Bellman-Ford algorithm.

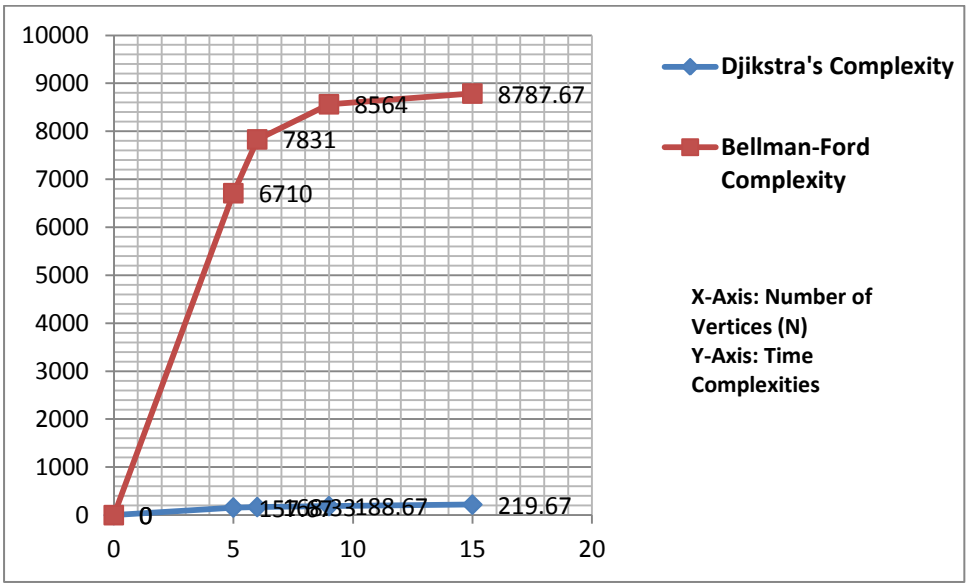

Fig. 3: Comparison of Bellman-Ford Algorithm & Djikstra's Algorithm Average Time Complexities for varied value of N.

Therefore in case of non-negative edge weights only, from above figure we conclude that with better implementation of Djikstra's Algorithm has lower running time than Bellman-Ford algorithm.

Applications of Shortest Path Problems

Shortest-paths is a broadly useful problem-solving model

- Maps
- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Subroutine in advanced algorithms.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Approximating piecewise linear functions.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.