Shortest Path Algorithms

Contents

1.Abstract

Shortest path problem is the problem of finding a path between two vertices (or nodes) in a graph such that the sum of the weights of its constituent edges is minimized. This paper provides an overview of different shortest path algorithms and comparative analysis of these algorithms. In this paper, I have evaluated single source shortest path and all pair shortest path algorithms. Dijkstra and Bellman Ford algorithms find the single source shortest path from single source to all destination vertices, where as Dijkstra all pair shortest path version and Floyd Warshall algorithms find all pair shortest path for every pair of vertices. This paper discusses time complexity and comparison of all algorithms with increased number of vertices and edges.

2.Introduction

2.1 Graphs

Graphs are a pervasive data structure in computer science, and algorithms for working with them are fundamental to the field. A graph data structure consists of a finite set of ordered pairs called edges of certain entities called vertices. In describing the running time of a graph algorithm on a given graph $G = (V, E)$, we usually measure the size of the input in terms of the number of vertices $|V|$ and the number of edges $|E|$ of the graph. Searching a graph means systematically following the edges of the graph so as to visit the vertices of the graph. There are two standard ways to represent a graph $G = (V, E)$ as a collection of adjacency lists or as an adjacency matrix. Either way is applicable to both directed and undirected graphs. The adjacency-list representation is usually preferred, because it provides a compact way to represent sparse graphs-those for which $|E|$ is much less than $|V|^2$.

The adjacency-list representation of a graph $G = (V, E)$ consists of an array Adj of $|V|$ lists, one for each vertex in V. For each u € V, the adjacency list $Adj[u]$ contains all the vertices v such that there is an edge (u, v) € E i.e., $Adj[u]$ consists of all the vertices adjacent to u in G. Adjacency lists can readily be adapted to represent weighted graphs, that is, graphs for which each edge has an associated weight, typically given by a weight function $w : E \rightarrow R$. For example, let $G = (V, E)$ be a weighted graph with weight function w. The weight $w(u, v)$ of the edge (u, v) € E is simply stored with vertex v in u's adjacency list.

For the adjacency-matrix representation of a graph $G = (V, E)$, we assume that the vertices are numbered 1, 2, …, $|V|$ in some arbitrary manner. Then the adjacency-matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$(a_{ij}) = \begin{cases} 1 \; if \; (i,j) \in E, \\ 0, Otherwise. \end{cases}$$

2.2 Generating Random Graph

A random graph is generated with vertices, edges, weights. For simplicity, let us focus on positive edges. The weights are assigned randomly between 1 to 10. Each vertex is connected to any number of vertices. Vertices, edges, weights are assigned randomly and writes to a text file and it is given as input for the shortest path algorithms. A sample graph with vertices, edges, weights are shown below.

Directed Graph written to a text file:

```
10
8
1  9  6
2  4  7
2  8  8
3  9  4
3  5  3
4  9  5
5  3  5
8  5  7
```

2.3 Shortest Path Problem

In a shortest-paths problem, we are given a weighted, directed graph G = (V, E), with weight function $w : E \rightarrow R$ mapping edges to real-valued-weights.

The weight of path $p = v0, v1, \ldots, v_k$ is the sum of the weights of its constituent edges:

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

We define the shortest-path weight from u to v by

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \rightarrow v\} \\ \infty \end{cases}$$

A shortest path from vertex u to vertex v is then defined as any path p with weight, $w(p) = \delta(u, v)$.

Single-source shortest-path problem: Finding a shortest path from u *to* v for given vertices u *and* v. If we solve the single-source problem with source vertex u, we can solve this problem.

All-pairs shortest-paths problem: Finding a shortest path from u *to* v for every pair of vertices u *and* v. Although this problem can be solved by running a single-source algorithm once from each vertex, it can usually be solved faster.

3.Algorithms and Complexity

3.1 Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph $G = (V, E)$ for the case in which all edge weights are nonnegative. In this section, therefore, we assume that $w(u, v) \geq 0$ for each edge $(u, v) \in E$.

Pseudocode:

$Dijkstra\ (Graph\ g, Src\ s)$

1. $Initializing\ dist[V], distTo[V]$
2. $for\ i\ in\ V$
3. $\quad distTo[i] = \infty, dist[i] = 0;$
4. $dist[src] = 0$
5. $for\ i\ in\ V - 1$
6. $\quad u = minDistance(dist, distTo, V)$
7. $\quad distTo\ [u] = 1;$
8. $\quad for\ i\ in\ g.adj[u]$
9. $\quad\quad v = i.first$
10. $\quad\quad if\ (!distTo[v] and\ dist[u] + i:second < dist[v])$
11. $\quad\quad\quad dist[v] = dist[u] + i:second$
12. return $distTo$[V]

- Dijkstra's algorithm initializes dist[s] to 0 and all other distTo[] entries to positive infinity except the source vertex.
- Then, it repeatedly relaxes and adds to the tree a non-tree vertex with the lowest distTo[] value.
- It continues untill all vertices are on the tree or no non-tree vertex has a finite distTo[] value.

Complexity:

- Dijkstra's algorithm solves the single-source shortest-paths problem in edge-weighted digraphs with nonnegative weights with the time complexity proportional to $E\ log\ V$ in the worst case.
- Overall Complexity of this implementation $V^2 + O(E) = O(V^2 + E)$.

3.2 Bellman Ford Algorithm

The Bellman-Ford algorithm solves the single-source shortest-paths problem in the general case in which edge weights may be negative. Given a weighted, directed graph $G = (V, E)$ with source s and weight function $w : E \rightarrow R$, the Bellman-Ford algorithm returns a Boolean value indicating whether or not there is a negative-weight cycle that is reachable from the source. If there is such a cycle, the algorithm indicates that no solution exists. If there is no such cycle, the algorithm produces the shortest paths and their weights.

Pseudocode:

$Bellmanford\ (Graph\ g, Src\ s)$

1. $Initializing\ distTo[V]$, V, E
2. $for\ i\ in\ V$
3. $\quad distTo[i] = \infty$
4. $distTo\ [src] = 0$
5. $for\ i\ in\ V - 1$
6. $\quad for\ u\ in\ V$
7. $\quad\quad for\ i\ in\ g.adj[u]$
8. $\quad\quad\quad initialize\ v = i.first,$
9. $\quad\quad\quad initialize\ weight = i.second$
10. $\quad\quad\quad if\ (distTo[u] + weight < distTo[v])$
11. $\quad\quad\quad\quad distTo\ [v] = distTo\ [u] + weight$
12. $return\ distTo\ [v]$
13. $for\ j\ in\ E$
14. $\quad if\ (distTo\ [u]! = \infty\ and\ dist[u] + weight < dist[v])$
15. $\quad\quad return\ Graph\ has\ negative\ cycle$

- Initialize distTo[s] to 0 and all other distTo[] values to infinity.
- Then, considering the digraph's edges in any order, and relax all edges.
- The only edges that could lead to a change in distTo[] are those leaving a vertex whose distTo[] value changed in the previous pass. To keep track of such vertices, we use a FIFO queue.

Complexity:

- For some graphs, only one iteration is needed, and best case scenario would be $O(|E|)$ time is needed.
- The worst-case time complexity of Bellman-Ford is $((|V||E|)$.

3.3 Floyd-Warshall Algorithm

Floyd Warshall algorithm uses dynamic-programming formulation to solve the all pairs shortest-paths problem on a directed graph $G = (V, E)$. we use a different characterization of the structure of a shortest path than we used in the matrix-multiplication-based all-pairs algorithms.

Pseudocode:
$FloydWarshall\ (Graph\ g, Src\ s)$

```
1.  initialize dist[V ][V ]
2.  for i in V
3.      for j in V
4.          if i == j
5.              dist[i][j] = 0
6.          else
7.              dist[i][j] = ∞
8.  for i in V
9.          for i in g: adj[i]
10.             dist[i][k: first] = k: second
11. for i in V
12.     for j in V
13.         for k in V
14.             if (dist[j][i] + dist[i][k] < dist[j][k])
15.                 dist[j][k] = dist[j][i] + dist[i][k]
```

- The algorithm considers the "intermediate" vertices of a shortest path, where an intermediate vertex of a simple path $p = v_{1,}v_{2,\ldots,}v_l$ is any vertex of p other than v1 or vl, that is, any vertex in the set $\{v_{2,}v_{3,\ldots,}v_{l-1}\}$.
- The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from $i\ to\ j$ with all intermediate vertices in the set $\{1, 2, \ldots, k - 1\}$.
- The relationship depends on whether or not k is an intermediate vertex of path p.

Complexity:

- The Floyd-Warshall algorithm requires $O(n^3)$ space, since we compute for $i, j, k = 1, 2, \ldots, n$.

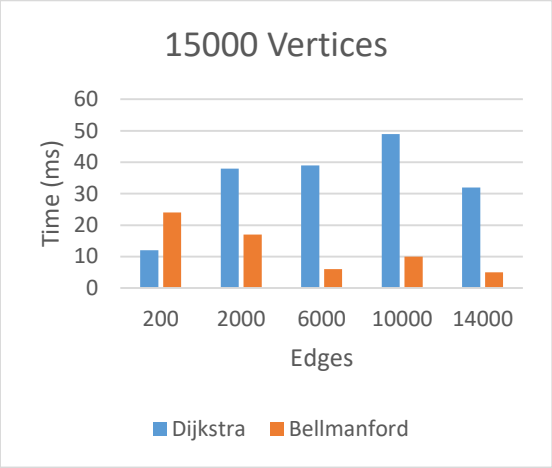
4.Results and Analysis

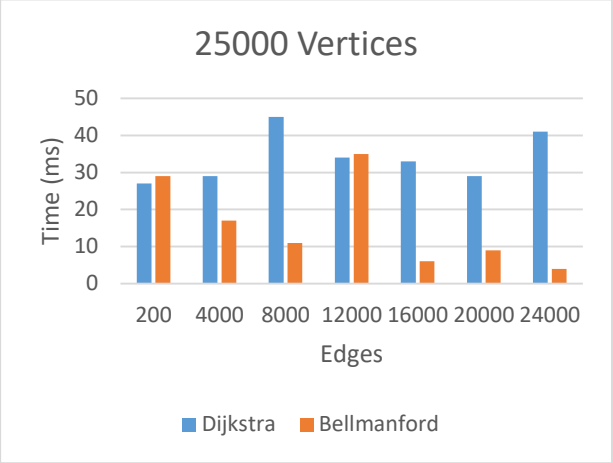
I have taken a random graph generator and restricted the weight to be positive and between 1,10. Time recording is started just before the algorithm starts and finishes after the algorithm ends. Then difference between the start time and end time is the resulting time algorithm has taken.

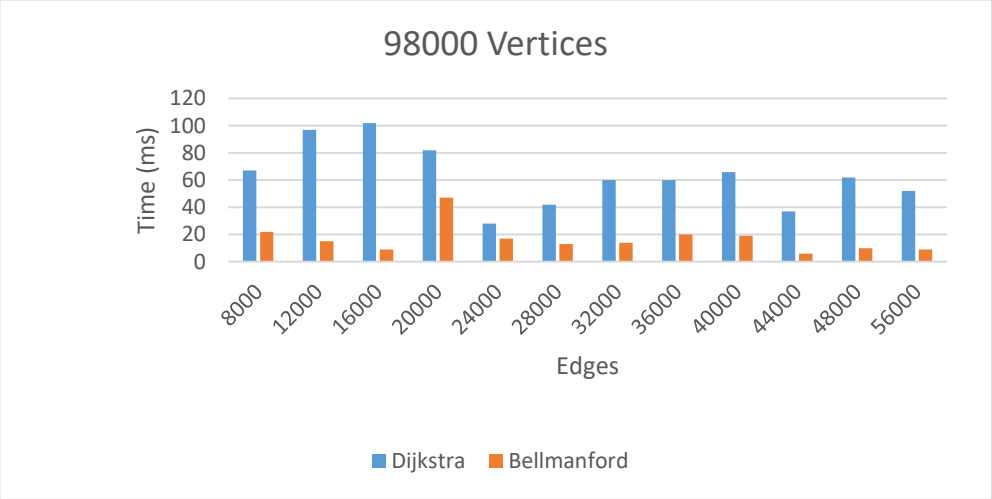
Below is the tabular data recorded for Dijkstra's and Bellman Ford Algorithms:

Vertices		Dijkstra	Bellmanford
1000	200	15	4
	500	11	4
	800	10	4
3000	200	12	6
	800	10	6
	1300	45	25
	2000	39	4
	2500	17	3
5000	200	12	4
	800	28	7
	2000	10	6
	4000	18	8
7000	200	11	5
	2000	11	21
	4000	22	17
	6000	17	6
9000	200	24	8
	2000	15	9
	4000	13	5
	6000	22	14
	8000	33	13
15000	200	12	24
	2000	38	17
	6000	39	6
	10000	49	10
	14000	32	5
20000	200	42	6
	2000	26	11
	6000	22	9
	10000	28	65
	14000	22	7
	18000	40	8
25000	200	27	29
	4000	29	17
	8000	45	11
	12000	34	35
	16000	33	6
	20000	29	9
	24000	41	4
30000	200	19	50
	6000	17	18
	12000	31	8
	18000	23	5
	24000	23	11
35000	200	24	22
	6000	33	17
	12000	37	38
	18000	34	8
	24000	28	7
	30000	56	6
98000	8000	67	22
	12000	97	15
	16000	102	9
	20000	82	47
	24000	28	17
	28000	42	13
	32000	60	14
	36000	60	20
	40000	66	19
	44000	37	6
	48000	62	10
	56000	52	9

Below are the plots for Dijkstra and Bellmanford Algorithms:







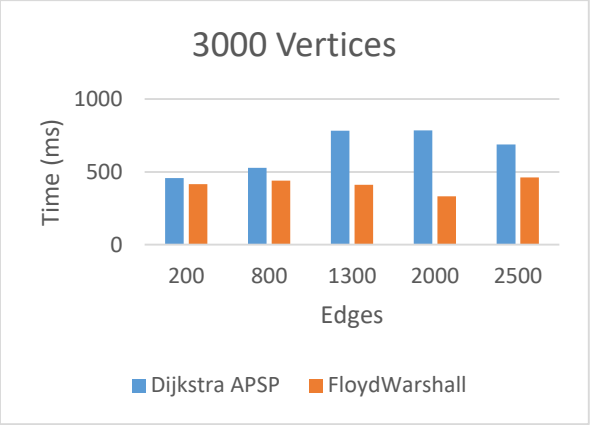
Single Source Shortest Path: Analysis of Dijkstra and Bellman Ford Algorithm

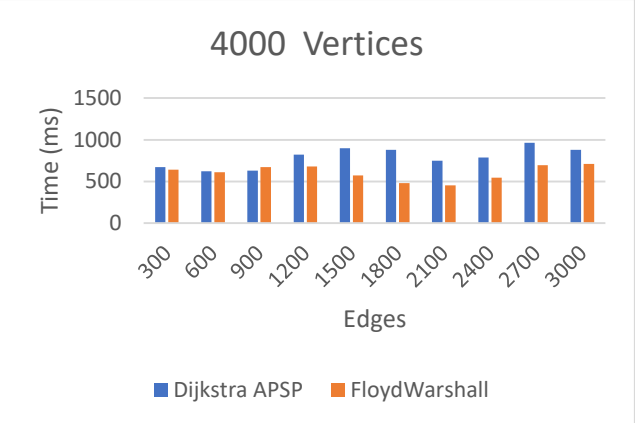
- From the above graphs and tabular data, it can be observed that Bellman Ford Algorithm runs faster than Dijkstra's Algorithm When the number of edges are smaller than vertices.
- Dijkstra's Algorithm performs better than Bellman ford when the number of edges nearly approaches V^2.

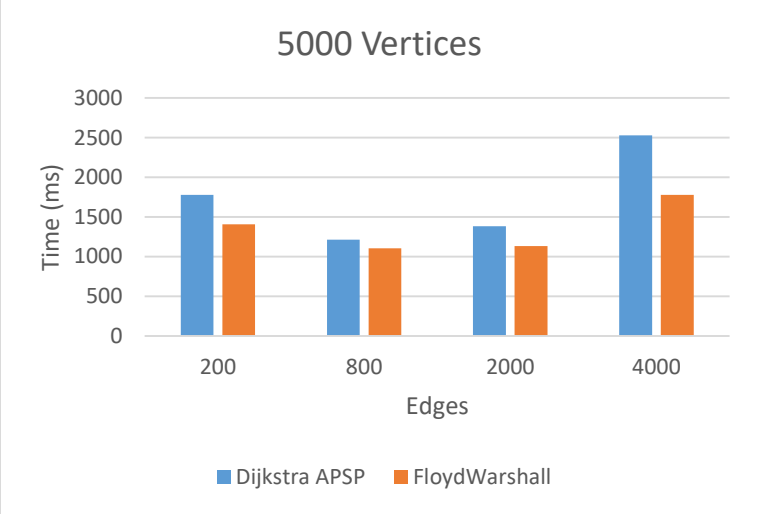
Below is the tabular data recorded for Dijkstra's APSP and Floyd Warshall Algorithms:

Vertices		Dijkstra APSP	FloydWarshall
1000	200	357	92
	500	206	118
	800	172	85
3000	200	457	415
	800	527	439
	1300	784	412
	2000	785	333
	2500	688	462
4000	300	672	643
	600	623	611
	900	629	673
	1200	820	680
	1500	900	572
	1800	879	481
	2100	749	452
	2400	786	545
	2700	962	695
	3000	881	711
5000	200	1778	1408
	800	1214	1106
	2000	1382	1133
	4000	2528	1777

Below are the plots for Dijkstra's APSP and Floyd Warshall Algorithms:







All Pair Shortest Path: Analysis of Dijkstra's APSP and Floyd Warshall Algorithms:

- From the above graphs and tabular data, both the algorithms has worst case complexity $O(V)^3$.
- FloydWarshall performs just better than Dijkstra's APSP version when there are more edges.

5.Conclusion

- Graph representation plays a crucial role in performance of the algorithm.
- The best algorithm to use will be dependent upon the type of graph you are using and the shortest path problem that is being solved.
- for problems with negative weight edges Bellman-Ford works better, whereas for sparse graphs with no negative edges Dijsktra's is choosen.
- For All pair version if there is dense graph Floyd-Warshall is preferred, Dijkstra would be preferred when the sparse graph.

6.References

1. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
2. https://en.wikipedia.org/wiki/FloydWarshall-algorithm
3. CharlesE.Leiserson,Cliord Stein, Ronald Rivest, and Thomas H. Cormen, Introduction to Algorithms, MIT Press, 2009. (Second Edition)
4. https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/
5. https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
6. https://algs4.cs.princeton.edu/44sp/