CS 584 Final Project

Kamakshi Nagar

Project Proposal: Shortest Path

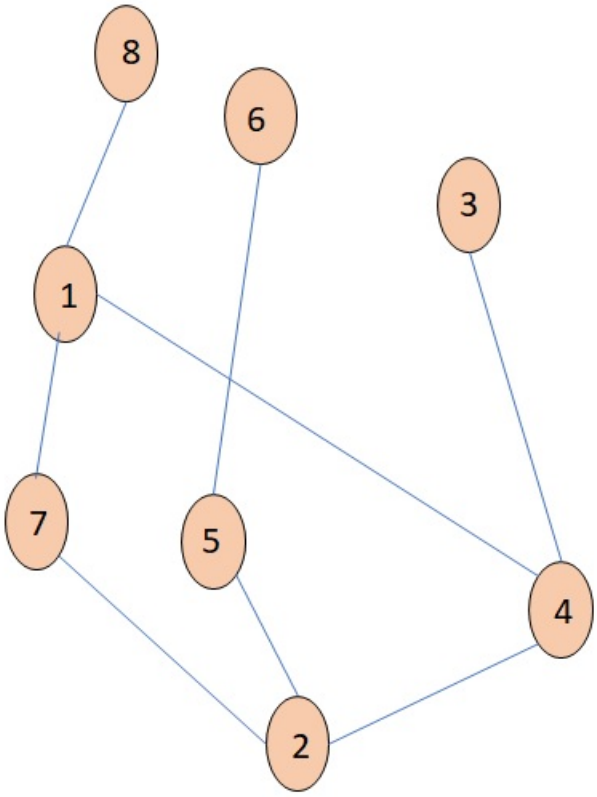
This project performs an empirical analysis of the shortest path algorithms Bellman Ford, Dijkstra's and Floyd Warshall algorithms.While Bellman Ford and Dijkstra's algorithm find shortest path from single source to all destination vertices, Floyd Warshall algorithm finds shortest path for every pair of vertices. Dijkstra's algorithm can also be used to find all pair shortest path with each vertex as a source. The experiments includes run time comparison of all algorithms with asymptotically increased number of vertices and edges.

Overview

Graph Graphs are mathematical structures used to model pairwise relations between set of vertices (also called nodes) which are connected by edges. A graph may be undirected, meaning that there is no distinction between the two vertices associated with each edge, or its edges may be directed from one vertex to another.

Representing 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. A potential disadvantage of the adjacency list representation is that there is no quicker way to determine if a given edge (u, v) is present in the graph than to search for v in the adjacency list of u given by $Adj[u]$.
The adjacency matrix approach is preferred for dense graphs, and for when we need to be able to tell quickly if there is an edge connecting two given vertices.
As far as representing weighted graphs go, the weight $w(u, v)$ of the edge is simply stored with vertex v in $Adj(u)$ - adjacency list for node u. For adjacency matrices, the weight $w(u, v)$ of the edge $(u, v) \in E$ is simply stored as the entry in row u and column v of the adjacency matrix. If an edge does not exist, a NIL value can be stored as its corresponding matrix entry,though for many problems it is convenient to use a value such as 0 or ∞.

Generating Random Graph A random graph is a graph in which properties such as the number of graph vertices, graph edges,and connections between them are determined in some random way.Each edge is included in the graph with probability independent from every other edge. For simplicity our focus is on directed graph with positive weight edges. we are not considering negative weights as we do not want to have negative weight cycles in our graphs.



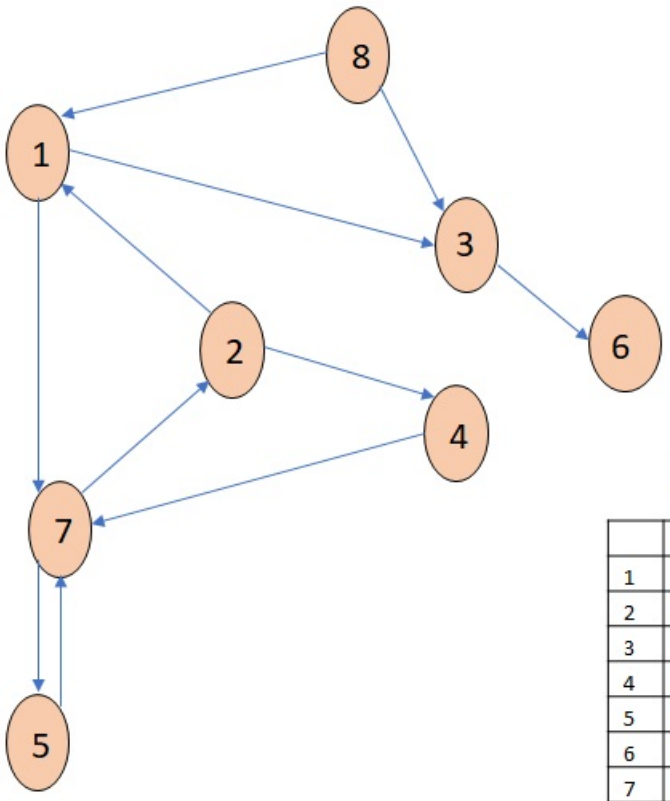
Undirected Graph

Adjacency List

1	4,7,8
2	4,5,7
3	4
4	1,2,3
5	2,6
6	5
7	1,2
8	1

Adjacency Matrix

	1	2	3	4	5	6	7	8
1	0	0	0	1	0	0	1	1
2	0	0	0	1	1	0	1	0
3	0	0	0	1	0	0	0	0
4	1	1	1	0	0	0	0	0
5	0	1	0	0	0	1	0	0
6	0	0	0	0	1	0	0	0
7	1	1	0	0	0	0	0	0
8	1	0	0	0	0	0	0	0



Directed Graph

Adjacency List

1	3,7
2	1,4
3	1,6
4	7
5	7
6	
7	2,5
8	1,3

Adjacency Matrix

	1	2	3	4	5	6	7	8
1	0	0	1	0	0	0	1	0
2	1	0	0	1	0	0	0	0
3	1	0	0	0	0	1	0	0
4	0	0	0	0	0	0	1	0
5	0	0	0	0	0	0	1	0
6	0	0	0	0	0	0	0	0
7	0	1	0	0	1	0	0	0
8	1	0	1	0	0	0	0	0

Shortest Path Problem

Statement of the problem In a shortest-paths problem, we are given a weighted, directed graph $G = (V, E)$

with weight function $w : \mathbb{R}$

The weight of path $p = v_0, v_1, ..., v_k$ is:

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

The shortest-path weight from u to v is

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

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

Algorithm Implementation

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

1 Initialize the set as an empty set. The set will contain minimum distance of all the vertices from source vertex.

2 Initialize distance value of all vertices to infinite except the source vertex.

3 Source vertex distance value will be zero and it will be first in the set.

4 Until all the vertex all included in the set

- pick a vertex u which is not in the set and has the minimum distance value.
- push the vertex u in the set.
- update the distance value of all adjacent vertices of u by iterating over all adjacent vertices.
- for every adjacent vertex v, If sum of distance value of u and weight of edge (u, v) is less than the distance value of v, then update the distance value of v.

Pseudocode: Dijkstra(Graph g, src)

```
1  initialize dist[V],Set[V]
2  for i in V
3        dist[i] = ∞ , Set[i] = false
4  dist[src] = 0
5  for i in V − 1
6        u = minDistance(dist, Set, V)
7        Set[u] = true
8        for i in g.adj[u]
9              v = i.first
10             if !Set[v] and dist[u] + i.second < dist[v]
11                   dist[v] = dist[u] + i.second
12 return Set[V]
```

minDistance(dist, Set, V)

```
1  initialize min value min = , min_idx
2  for v in V
3        if Set[v] == false and dist[v] <= min
4              min = dist[v], min_idx = v
5  return min_idx
```

Complexity Analysis:

- The time complexity of the above code/algorithm looks $O(V^2)$ as there are two nested for loops. The outer loop goes for V time and a loop to find minimum distance runs V times.

- If we take a closer look, we can observe that the statements in another inner loop are executed only E (number of edges) times, as we are using adjacency list .
- So overall Complexity of this implementation $V^2 + O(E) = O(V^2 + E)$.

Bellman-Ford Algorithm The Bellman-Ford algorithm solves the single source shortest paths problem on graph $G(V, E)$ when edge weights may be negative. We are only considering positive weight edges for simplicity.

- Initialize distance from source to all vertices as infinite and source vertex to itself as 0.
- For each edge $(u, v) \in E(G)$
  - Relax each edges of the graph
  - If distance of vertex v is greater than the sum of the distance of u and weight of the edge
    then update the distance of v.
- For each edge $(u, v) \in E(G)$
  - If distance of vertex v is greater than the sum of the distance of u and weight of the edge
    then graph has negative weight cycle.

Pseudocode: BellmanFord(Graph g, int src)

```
1  initialize V, E, dist[V]
2  for i in V
3        dist[i] = ∞
4  dist[src] = 0
5  for i in V − 1
6        for u in V
7              for i in g.adj[u]
8                    initialize v = i.first, weight = i.second
9                    if dist[u] + weight < dist[v]
10                         dist[v] = dist[u] + weight
11 return dist[v]
12 for i in E
13       if dist[u]! = ∞ and dist[u] + weight < dist[v]
14             return Graph has negative weight cycle
```

Complexity Analysis:

- This algorithm solves shortest path problem using Dynamic Programming approach, and calculate shortest paths in bottom-up manner.
- It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates shortest paths with at-most 2 edges, and so on.

- After the i^{th} iteration of outer loop, the shortest paths with at most i edges are calculated. There can be maximum $|V|-1$ edges in any simple path, that is why the outer loop runs $|V|-1$ times.
- The worst-case scenario is that Bellman-Ford runs in time $O(|V|.|E|)$. For certain graphs, only one iteration is needed, and so in the best case scenario, only $O(|E|)$ time is needed.

Floyd Warshall Algorithm: Floyd Warshall algorithm finds shortest path from u to v, for every pair of vertices $(u, v) \in V$.The output display a table form:the entry in $u's$ row and $v's$ column should be the weight of a shortest path from u to v. It is another dynamic programming approach where When we consider k^{th} vertex as an intermediate vertex, we already have considered vertices $\{1, 2, .., k-1\}$ as intermediate vertices. Thus know the shortest path through $\{1, ..., k-1\}$. This qualifies two property of dynamic programming optimal substructure and overlapping sub problems.

- Initialize the output matrix same as the input graph matrix
- Update the output matrix by considering all vertices as an intermediate vertex. Thus, update all $\{1, 2, .., k-1\}$ shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
- For every pair (i, j) of the source and destination vertices respectively
  - If k is not an intermediate vertex in shortest path from i to j then keep the value of $dist[i][j]$ as it is.
  - Else if $dist[i][j] > dist[i][k] + dist[k][j]$ update the value of $dist[i][j]$ as $dist[i][k] + dist[k][j$

Pseudocode: FloydWarshall (Graph g)

```
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 k in V
12      for i in V
13           for j in V
14                if dist[i][k] + dist[k][j] < dist[i][j]
15                dist[i][j] = dist[i][k] + dist[k][j]
```

Complexity Analysis: The algorithm finds shortest path of all pair of vertices, which is n^2. shortest path(i,j,k) from path (i,j,k-1) requires $2n^2$ operations. The algorithm computes shortest path $(i, j, 1), (i, j, 2), .., (i, j, n)$ requires $n * 2n^2 = 2n^3$ operations. Therefore, the overall complexity of the algorithm is $\Theta(n^3)$.

Results

Experiment Set Up: We will compare the run time for Dijkstra and Bellman ford and then Floyd-Warshall Algorithm and Dijkstra's All Pair version. Graph are generated randomly with vertexes and edges sampled independently. We have kept weight to be positive and it ranges from 1 to 10. Weights are assigned to the edges with same probability.

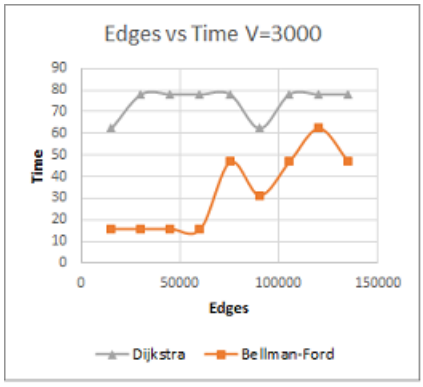
Sample input looks like this:

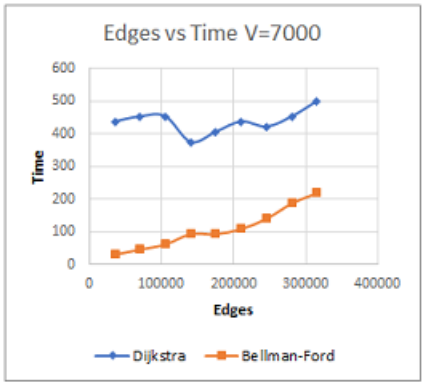
V	100	100	100	500	500	500	500	1000	1000	1000	1000	1000
E	50	550	1050	2550	5050	7550	10050	10050	15050	20050	25050	45050

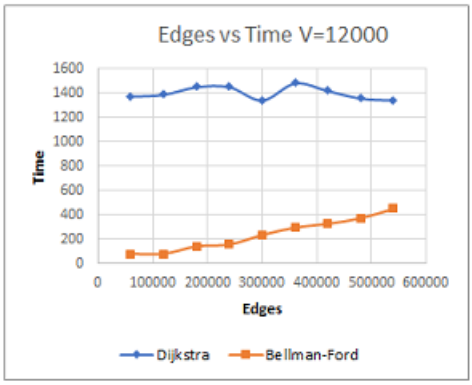
We start the clock after the control goes to the particular function and end the clock once the algorithm finds the shortest path. Then we compute the difference between the start time and end time, that will be run time for that algorithm. Runtimes also depends on how the memory is allocated, For adjacency list the pointer will jump to different memory locations to find the next address. If memory allocation is sequential the performance will be faster. we have changed the units to milliseconds for better performance measures.

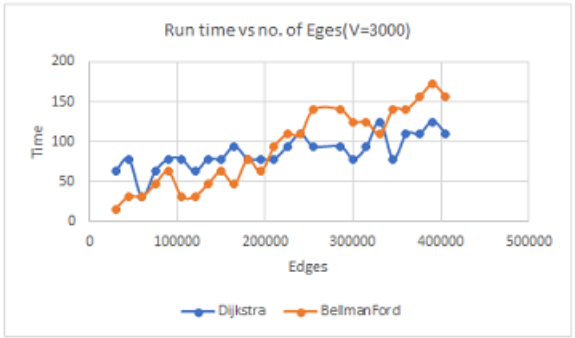
Tables and Plots:

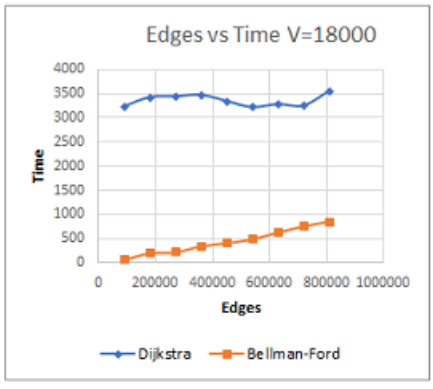
Single Source Shortest Path









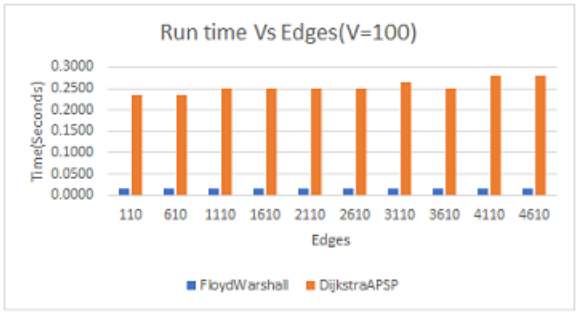


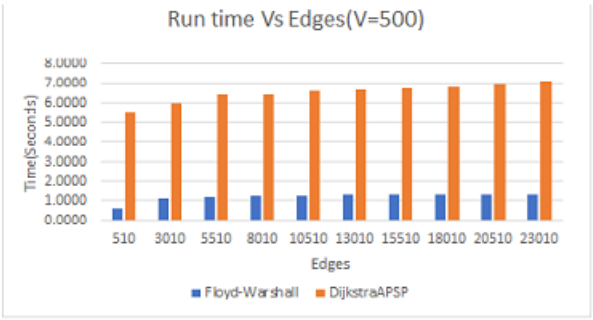
V	E	Dijkstra	Bellman-Ford
3000	15050	62.500	15.625
	30050	78.125	15.625
	45050	78.125	15.625
	60050	78.125	15.625
	75050	78.124	46.876
	90050	62.500	31.252
	105050	78.124	46.876
	120050	78.124	62.500
	135050	78.124	46.872
7000	35050	437.504	31.248
	70050	453.128	46.872
	105050	453.120	62.496
	140050	375.000	93.752
	175050	406.256	93.760
	210050	437.504	109.376
	245050	421.872	140.624
	280050	453.120	187.488
	315050	500.000	218.752
12000	60050	1374.976	78.112
	120050	1390.624	78.144
	180050	1453.120	140.608
	240050	1453.120	156.256
	300050	1343.744	234.368
	360050	1484.352	296.832
	420050	1421.888	328.128
	480050	1359.360	375.040
	540050	1343.744	453.184
18000	90050	3234.368	62.528
	180050	3421.888	203.072
	270050	3437.440	218.752
	360050	3468.800	343.680
	450050	3343.744	406.272
	540050	3218.688	484.352
	630050	3281.152	625.024
	720050	3249.920	749.952
	810050	3546.880	843.776

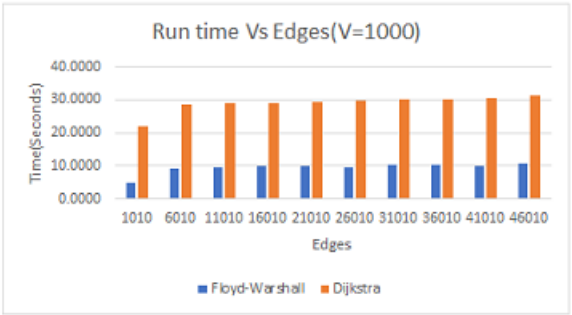
V	E	Dijkstra	BellmanFord
3000	30050	62.500	15.625
	45050	78.125	31.250
	60050	31.250	31.250
	75050	62.500	46.876
	90050	78.124	62.500
	105050	78.126	31.248
	120050	62.504	31.252
	135050	78.128	46.876
	150050	78.124	62.500
	165050	93.748	46.876
	180050	78.124	78.124
	195050	78.128	62.504
	210050	78.128	93.752
	225050	93.752	109.376
	240050	109.376	109.376
	255050	93.752	140.624
	285050	93.744	140.624
	300050	78.120	125.000
	315050	93.760	125.008
	330050	124.992	109.360
	345050	78.128	140.624
	360050	109.376	140.624
	375050	109.376	156.256
	390050	125.008	171.872
	405050	109.376	156.256

All Pair Shortest Path

V	E	FloydWarshall	DijstraAPSP
100	110	0.0156	0.2344
	610	0.0156	0.2344
	1110	0.0156	0.2500
	1610	0.0156	0.2500
	2110	0.0156	0.2500
	2610	0.0156	0.2500
	3110	0.0156	0.2656
	3610	0.0156	0.2500
	4110	0.0156	0.2813
	4610	0.0156	0.2813
500	510	0.6250	5.5781
	3010	1.0781	6.0000
	5510	1.1875	6.4531
	8010	1.2344	6.4375
	10510	1.2500	6.5938
	13010	1.2813	6.6875
	15510	1.2656	6.7656
	18010	1.2969	6.7813
	20510	1.2813	6.9688
	23010	1.2813	7.1250
1000	1010	5.0000	22.0625
	6010	9.0156	28.6406
	11010	9.6875	28.8906
	16010	9.8438	28.9062
	21010	9.9375	29.3125
	26010	9.5469	29.5781
	31010	10.1563	30.2812
	36010	10.1562	30.1406
	41010	10.0312	30.3906
	46010	10.5781	31.2969







Observation and algorithms comparison: Looking at the above table and Plots we can observe the following observation for large number of vertices and columns:

Single Source SP:

- Bellman Ford Algorithm runs faster than Dijkstra's Algorithm When the number of edges are smaller than vertices.
- However Dijkstra's Algorithm outperforms Bellman ford when the number of edges nearly approaches V^2.
- we can look at this complexity wise -
  - Dijkstra's Complexity is $O(V^2 + E)$ and Bellman-Ford's Complexity is $O(V * E)$
  - When E reaches V^2, Dijkstra $= V^2 + V^2 = 2V^2$ and Bellman-Ford $= V * V^2 = V^3$
  - This explains above plot when Dijkstra outperforms Bellman-Ford.

All Pair SP:

- Comparison between Floyd-Warshall and DijkstraAPSP in implementation shows that even though both the algorithm has worst case $O(V^3)$, their run time are quite different.
- Floyd-Warshall is way faster than Dijkstra.

Conclusion

Learning:

- There are lot of algorithms for finding shortest path. However Dijkstra and Bellman Ford written in (1955-56) and Floyd Warshall (1962) are still pretty relevant and useful.
- Representation of the graph(adjacency list or matrix) has huge impact on performance of the algorithm.
- In SSSP for sparse graphs Bellman-Ford outperforms Dijkstra but as number of edges increase, Dijkstra outperforms Bellman-Ford.
- C++ does not have garbage collector, memory can leak if is not handled properly in the program, resulting in the segmentation fault.
- In APSP for dense graph Floyd-Warshall is preferred, Dijkstra is good choice when the graph is sparse.

Future study:

- All the algorithms can be studied for modern applications such as self driving cars, social networking graphs etc.
- Implementation using different data structure is also one of the interesting areas.
- All algorithms studied above has been improved over time and have better performance, project can be enhance to include those implementation.
- Also we can include space complexity comparisons to see memory usage by each implementation.

References:

- Charles E. Leiserson, Clifford Stein, Ronald Rivest, and Thomas H. Cormen, Introduction to Algorithms, MIT Press, 2009.
- https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2015
- https://en.wikipedia.org/wiki/BellmanFord
- http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf
- https://en.wikipedia.org/wiki/FloydWarshall-algorithm

- https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/
- https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/
- https://en.wikipedia.org/wiki/Parallel-all-pairs-shortest-path-algorithmExample