Study on Single Source Shortest Path Algorithms

Jannatul Ferdous
Student Id: 011 131 119

Md. Afser
Student Id: 011 131 123

Mejbah Uddin Shameem
Student Id: 011 132 038

Md. Mahtab Hossain Milon
Student Id: 011 132 046

A Thesis

in

The Department

of

Computer Science and Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of Bachelor of Science in Computer Science and Engineering

United International University

Dhaka, Bangladesh

December 9, 2017

Approval Certificate

This thesis titled "**Study on Single Source Shortest Path Algorithm**" submitted by **Jannatul Ferdous**, Student ID: 011 131 119, **Md. Afser**, Student ID: 011 131 123, **Mejbah Uddin Shameem**, Student ID: 011 132 038, **Md Mahtab Hossain Milon**, Student ID: 011 132 046, has been accepted as Satisfactory in fulfillment of the requirement for the degree of Bachelor of Science in Computer Science and Engineering on December 9, 2017.

Board of Examiners

_____ Supervisor

Dr. Mohammad Nurul Huda
Professor and MSCSE Coordinator
Department of Computer Science and Engineering
United International University

Ex-Officio

_____

Dr. Salekul Islam

Associate Professor and Head

Department of Computer Science and Engineering

United International University

Declaration

This is to certify that the work entitled "**Study on Single Source Shortest Path Algorithm**" is the outcome of the research carried out by me under the supervision of **Dr. Mohammad Nurul Huda,** Professor and MSCSE Coordinator, United International University (UIU), Dhaka, Bangladesh.

Jannatul Ferdous, Student ID: 011 131 119, CSE

Md. Afser, Student ID: 011 131 123, CSE

Mejbah Uddin Shameem, Student ID: 011 132 038, CSE

Md Mahtab Hossain Milon , Student ID: 011 132 046, CSE

In my capacity as supervisor of the candidate's thesis, I certify that the above statements are true to the best of my knowledge.

Dr. Mohammad Nurul Huda
Professor and MSCSE Coordinator
Department of Computer Science and Engineering
United International University

Abstract

Single Source Shortest Path Algorithm is one of the most used algorithm for routing and tracking path around the world for communication. Nowadays various famous apps are also using this algorithm to track the path and gaining information through the Google Map using different API. We implement Dijkstra, Floyd Warshall, Bellman Ford, Nearest Neighbor, Johnson algorithm for analysis then we implement shortest path algorithm using Genetic Algorithm.

Genetic algorithms are used to solve optimization problems and we have tried to implement this optimization idea into shortest path algorithm to make an innovative shortest path calculation algorithm. The advantage of the algorithm is the more number of edges the less it will take time to calculate shortest path.

This paper evaluates the various single source shortest path algorithms. We have implemented algorithms with graph to analysis the algorithm and then try to calculate shortest path with the Genetic Algorithm and then compare them with each other.

Acknowledgement

We would like to start by expressing our deepest gratitude to the Almighty Allah for giving us the ability and the strength to finish the task successfully within the scheduled time.

This thesis titled "**Study on Single Source Shortcut Path Algorithms**" has been prepared to fulfill the requirement of Bachelor degree. We are very much fortunate that We have received sincere guidance, supervision and co-operation from various persons.

We would like to express our heartiest gratitude to our supervisor, **Dr. Mohammad Nurul Huda**, Professor and MSCSE Coordinator, United International University, for his continuous guidance, encouragement, and patience, and for giving us the opportunity to do this work. His valuable suggestions and strict guidance made it possible to prepare a well-organized thesis report.

Finally, our deepest gratitude and love to our parents for their support, encouragement, and endless love.

Table of Contents

LIST OF FIGURES

Chapter 1

Introduction

1.1 Objectives

This paper consists of some of the existing algorithms for single source shortest path algorithm and we implemented an algorithm by using genetic algorithm to calculate the shortest path and cost for a single source. We also compare the results of the implemented algorithms.

1.2 Problems

Shortest path problems are classical combinatorial problems that arise as sub problems when solving many optimization problems. As they are relatively easy to solve and at the same time they contain the most important ingredients of network flows, shortest path problems are a starting point for studying more complex network problems. In fact, they have been widely studied leading to a great number of algorithms adapted to find an optimal solution in various special conditions and/or constraint formulations Shortest path problems have involved the interest of both researchers and practitioners, because they appear in a wide variety of contexts, whenever some material, or a telephone call, a computer data packet, etc. needs to be sent as cheaply or as quickly as possible.

1.3 Motivation

Single source shortest path algorithm is one of the most used algorithms all over the world. These types of algorithms mainly used in mainly different types of networks like telephone networks, IP routing, tracking software where shortest path is the most important factor. Therefore, several researches are being carried out to minimize the elapsed time of work done to evaluate shortest path with the optimal cost.

Chapter 2

Previous Study

2.1 Introduction

The shortest path algorithm is used mainly for calculating the shortest path of a graph or network. The shortest path problem is the problem of finding a path between two nodes in a graph or network such that sum of the weights of its constituent edges is minimized. The algorithms which are used to find the shortest path are shortest path algorithms [2]. In this paper we have discussed this algorithm only for single source.

2.2 Dijkstra's Algorithm

Dijkstra's algorithm[1] is one of the fastest algorithm to track the shortest path from any graph. It is almost similar to Prim's algorithm for minimum spanning tree. In Prim's algorithm we generate a shortest path tree with the given source. We maintain two sets where one set is for vertices included in that tree and the other set contains vertices not yet included in that shortest path tree. At each iteration of the algorithm, we find a vertex which is in the other set and has minimum distance from the source.

Steps to measure the shortest for a single source vertex to all other vertices is given below:

1. Need to create a set that keeps track of vertices included in shortest path tree. Initiallay the set is empty.
2. Need to assign distance value to all vertices in the input graph. All the distance value must be initialized as INFINITE. Must assign distance value as 0 for the source vertex so that it is picked first.
3. While the set (which is mentioned in 1) does not includes all the vertices
   i) Need to pick a vertex which is not there in the set and has minimum distance value.
   ii) Then include that vertex into the set.

ii) Update distance value of all adjacent vertices of that vertex. To update the distance values, iterate through all adjacent vertices. For any adjacent vertex, if the sum of the distance value for that vertex and weight of edge from vertex to adjacent vertex is less than the distance value of the adjacent vertex then update the distance value of adjacent vertex.

2.2.1 Pseudocode

Algorithm 2.1 Pseudo code for Dijkstra's Algorithm

```
1: Create vertex set Q.
2: for each vertex v in Graph do
3:     distance[v] ←∞;
4:     predecessor[v] ← null;
5:     add v to Q
6: end for
7: distance[source] ← 0
8: while Q is not empty, do
9:     u ← vertex in Q with min distance[u]
10:    remove u from Q
11:    for each neighbor v of u: do
12:    alt ← distance[u] + length(u,v)
13:      if alt < distance[v] then
14:        distance[v] ← alt
15:        predecessor[v] ← u
16:      end if
17:    end for
18: end while
```

2.2.2 Advantages and Disadvantages

Advantages: -

- It is used in Google Maps
- It is used in finding Shortest Path.
- It is used in geographical Maps

- To find locations of Map which refers to vertices of graph.
- Distance between the location refers to edges.
- It is used in IP routing to find Open Shortest Path First.
- It is used in the telephone network.

Disadvantages: -

- It does blind search so wastes lot of time while processing.
- It cannot handle negative edges.
- This leads to acyclic graphs and most often cannot obtain the right shortest path.

2.2.3 Complexity

It depends on the implementation of Dijkstra's Algorithm.

Time Complexity: O(s*|E|log|E|+|V|)

Space Complexity: O(V^2)

Here,

s: the number of sources

n: the number of vertices to calculate

V: the number of vertices in the graph

E: the number of edges in the graph

2.3 Bellman Ford Algorithm

Bellman Ford's algorithm does the same job which dijkstra's algorithm does but the difference is it can calculate the shortest path even if any edge of the graph has negative weight. Negative weight vertex may seem useless but it explain a lot of phenomena like cash flow, heat released or absorbed in any chemical reaction etc.

Negative weight edges can create negative weight cycle which will reduce the total path distance. Shortest path algorithms like Dijkstra's algorithm that does not work in the presence of negative cycle because they can go through a negative weight cycle and reduce the path length.

2.3.1 Pseudo Code

Algorithm 2.4 Pseudo code for Bellman Ford Algorithm

```
1: for each vertex v in vertices: do
2:    distance[v] ←∞;
3:    predecessor[v] ← null;
4: end for
5: distance[source] ← 0
6:  for i from 1 to size(vertices)-1: do
7:    for each edge (u, v) with weight w in edges do
8:       if distance[u] + w < distance[v] then
9:          distance[v] ← distance[u] + w;
10:          predecessor[v] ← u;
11:      end if
12:    end for
13: end for
14: for each edge (u, v) with weight w in edges do
15:     if distance[u] + w < distance[v] then
16:         error ”Graph contains a negative-weight cycle”
17:     end if
18: end for
```

2.3.2 How Bellman Ford’s Algorithm Works:

Bellman Ford algorithm overestimates the distance of the path from source vertex to all other vertices. Then it iteratively relaxes those estimation by finding new shorter paths.

By doing that for all vertices the final outcome will be the optimized one. All the steps for gaining the optimal cost and path given below:

1. Start with a weighted graph
2. Except the source vertex assign every path values as infinite to all other vertices.

3. Need to go through each of the edges and relax the distance if that is inaccurate. Need to repeat it unless all the edges are visited.
4. After all the vertices get their path distance , need to check if any negative cycle exists or not.

2.3.3 Cost of Bellman Ford using Sequences

On the off chance that we accept the vertices are the whole numbers {0, 1, . . ., |V|−1} then we can utilize exhibit successions to execute a vtxTable. We can use nth requiring only O(1) work instead of using a formula which needs O(log n) work. This change in expenses can be connected for turning upward in the diagram to discover the neighbors of a vertex, and gazing upward out there table to locate the present separation.Using the improved costs,

$$W = O\left(\sum_{v \in V} (1 + |N_G(v)| + \sum_{u \in NG(v)} 1)\right)$$

$$= O(m)$$

$$S = O\left(\max_{v \in V} (1 + \log |N_G(v)| + \max_{u \in N(v)} 1)\right)$$

$$= O(\log n)$$

Therefore the overall complexity for BellmanFord with array sequences is:

$$W(n, m) = O(nm)$$

$$S(n, m) = O(n \log n)$$

By using this sequences, we have decreased the work by a O (log n) factor.

2.3.4 Complexity

Time Complexity: O(s*|E|*|V|)
Space Complexity: O(V^2)
Here,
s: the number of sources
n: the number of vertices to calculate
V: the number of vertices in the graph
E: the number of edges in the graph

2. 4 Floyd - Warshall Algorithm

The **Floyd – Warshall algorithm** is an algorithm to have shortest paths with positive or negative edge weights. Lengths of the shortest paths between all vertices will be found by executing once. Although it does not give full details of the paths but we can recreate the paths by modifying the algorithm.

The Floyd– Warshall calculation analyzes every conceivable way through the chart between each combine of vertices.It can work with θ **($|V|^3$)** comparisons in a graph. This is exceptional that there may be up to **Ω ($|V|^2$)** edges in the graph, and every combination of edges is examine. It does as such by incrementally enhancing a gauge on the most brief way between two vertices, until the point when the gauge is ideal.

Suppose a graph **G** with vertices **V** numbering 1 through N, a function **findshortPath(m,n,o)** that gives the possible shortest path from **m** to **n** using the set {1,2,....,o} as medial points along the path.Our plan is to search the shortest path from each **m** to each n using only vertices in {1,2,...,**N**}

For each of these pairs of vertices, the **findshortPath(m,n,o)** could be either

1) a path that **doesn't** go through o.

2) a path that goes through **o**.

If w(i,j) is the weight of the edge between m and n , we can conclude **findshortPath(m,n,o)**) by recursive formula given below

findshortPath(m,n,0) = w(m,n)

and the recursive case is

findshortPath(m,n,o) = min(**findshortPath** (m,n,o=1),
findshortPath (m,o,o-1)+ **findshortPath** (o,n,o-1)

2.4.1 Pseudo Code

Algorithm 2.5 Pseudo code for Floyd Warshall Algorithm

```
1: for i = 1 to N do
2:   for j = 1 to N do
3:      if there is an edge from i to j then
4:            dist[0][i][j] = the length of the edge from i to j
5:      else
6:            dist[0][i][j] = INFINITY
7:      end if
8:   end for
9: end for
10: for k = 1 to N do
11:   for i = 1 to N do
12:      for j = 1 to N do
13:         dist[k][i][j] = min(dist[k-1][i][j], dist[k-1][i][k] + dist[k-1][k][j])
14:      end for
15:   end for
16: end for
```

2.4.2 Advantages and Disadvantages

The Floyd Warshall algorithm is known for its plainness.

Advantages: -

- Easy to write.
- Features all the most limited way combines in a chart.

Disadvantages: -

- Slower than all algorithms.

2.4.3 Complexity

Time Complexity: $O(n^3)$

Space complexity: $O(n^2)$

2. 5 Johnson's Algorithm

Johnson's algorithm takes an input graph. This graph has a set of vertices that retail a set of edges. Each edge has a weight function. Directed, weighted graphs can be solved by using Johnson's algorithm. It allows edges to have negative weights, yet there can be no negative weight cycles. Because of its subroutines it can support negative weight.

Johnson's algorithm has three main steps.

i. Another vertex is added to the diagram, and it is associated by edges of zero weight to all different vertices in the chart.
ii. Negative weights are being extracted by a process called reweighting.
iii. The included vertex from stage 1 is expelled and on every node Dijkstra's algorithm is being runned.

2.5.1 Pseudo Code

Algorithm 2.6 Pseudo code for Johnson's Algorithm

1: *Compute* G^0, *where* $V[G^0] = V[G] \cup s$;
2: $E[G^0] = E[G] \cup (s,v) : v \in V[G]$;
3: w(s,v) = 0 FOR all $v \in V[G]$;
4: if BELLMAN-FORD(G^0, w, s) = FALSE then
5: Print the input graph contains a negative weight cycle
6: else
7: for each vertex $v \in V[G^0]$ do
8: set $h(v)$ to the value of $\delta(s,v)$computed by the Bellman-Ford alg.
9: end for
10: for each edge $(u,v) \in E[G^0]$ do
11: $w^0(u,v) \leftarrow w(u,v) + h(u) - h(v)$
12: end for

13: for each vertex $u \in V[G]$ do
14: run DIJKSTRA(G,w^0,u) to compute $\delta(s,v)$ for all $v \in V[G]$
15: for each vertex$v \in V[G]$ do
16: $d(u,v) \leftarrow \delta(s,v) + h(v) - h(u)$
17: end for
18: end for
19: end if

2.5.2 Complexity

Here V=total no. of vertices. and E= total no. of edges

Time Complexity: $O(V^2\log(V) + VE)$

Chapter 3

Proposed Work

3.1 Genetic Algorithm

GAs is a searched based adaptive heuristic algorithm [3]. The idea of GAs comes from natural selection and genetics. It finds the optimal or near-optimal solutions to solve the problems. It can look through changed blends of materials and outlines to locate the ideal mix of both which could bring about a more grounded, lighter and by and large, better last item. GA simulate the survival of the fittest among individuals over consecutive generation for solving a problem. It takes random solutions as initial population, then using Crossover and Mutation function to produce more offspring. And then supply them through the Fitness Function to produce better offspring sometime worse than parent.

3.2 Proposed method using GA

There are various types of Shortest path algorithm. We are experimenting some of these existing algorithms and discuss about them in the previous chapter. For our proposed method we are using concept of GA. We are using GA because it can search through a huge combination of parameters to find the best match. Here, in our proposed technique we are trying to find the optimal cost path among many possible paths. We use Crossover, Mutation and Fitness function to get the far most better shortest path. Through this chapter we will discuss about our proposed algorithm and how we use GA in terms of proposed system.

3.2.1 Sample Input Graph

Here, we are using following weighted graph as input to demonstrate how our proposed algorithm works and our goal is to find a shortest path from start node to end node for the minimum cost.

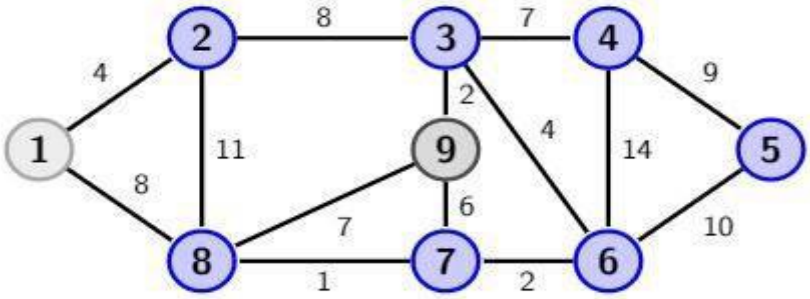


Figure 1: Input Graph

3.2.2 Initial population

As initial population for our proposed method we randomly generate 20 chromosomes or individuals. Here 20 is called the population size and it can vary according to the input graph size. Each of this chromosome contains various randomly generated solution. From our input graph (fig:3.3.1) we select 1 as our starting node and 9 as end node. In 20 randomly generated chromosomes, we have combinations with rest seven nodes. Why we are not using 1st and 9th node for randomly generated chromosomes will be discussed in Fitness Function Section. After creating 20 population, we pass them to the Crossover function.

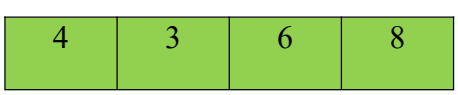
3.2.3 Crossover

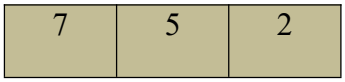
The idea of crossover in GA is to mix two elements and make two different offspring from them. If we consider all input as individual chromosomes, then consider 2 chromosomes as parent (parent1 and parent2). Now if we consider 2 parents as an array, then we have to select a valid cutting point to decide how much gene the new offspring get from each parent.

After initializing the population which consists of some initial random solutions, now it is time to pass these initial solutions to the next stage of the algorithm. The next stage initiates with the calling of the function crossover. By now we already know how a generic crossover operator works for genetic algorithm. However, in this section we are going to have discussion how this crossover technique is used for solving shortest path problem.

We have the initial population which has the number of individuals of maximum population size. Each of the individual is also called chromosome. Two different chromosomes can be used as parent while they are reproducing some child. First of all, we have to select two random different individuals from initial population to impose the crossover functionality on them.

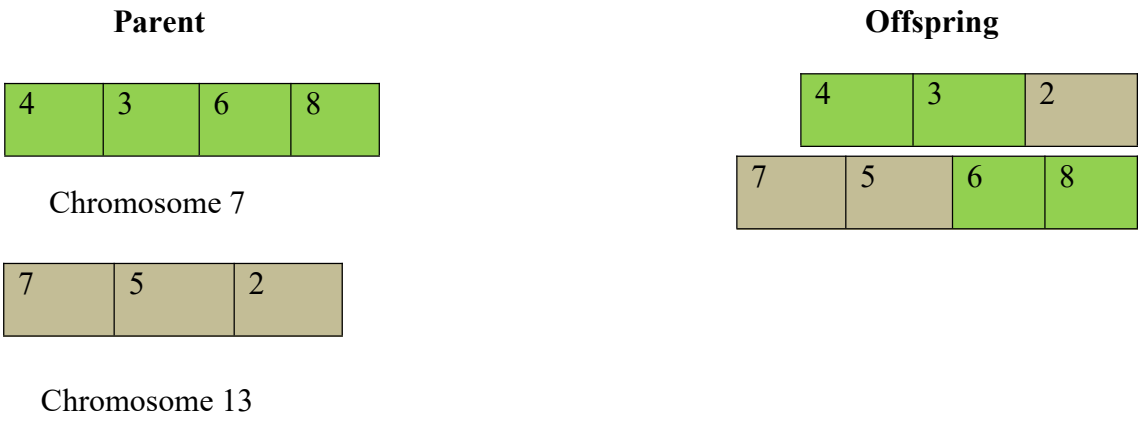
As we are working with 20 initial solutions or individuals the tag of parents can be from 1 to 20. For example, let's consider, we randomly choose chromosome of tag 7 and 13. We already know that the chromosomes have the node number in them which are genes and they don't contain source and destination node number (why? Will be discussed later). So, for the particular example of the figure 3.3.1, the possible node number can be from 2 to 8, if node 1 and 9 is the source and destination respectively. From the following figures, we can understand how the chromosomes look like.





If we examine carefully, we can see that the size of these two chromosomes is not same. One of the novel part of crossover is the selection of crossover point. A crossover point indicates which nodes from which chromosome are selected. So, the maximum value of crossover point will be the size of the parent which is containing minimum nodes. In this case, the crossover point can be chosen randomly from the range of 1 to 3, as 3 is the minimum size of the parents.

Let's consider, 2 is the randomly picked crossover point. The first offspring produced from the crossover of chromosome 7 and 13 will contain node number 4 and 3 from chromosome 7 and node number 2 from chromosome 13. The second offspring will be consisted in opposite action. 2 is the crossover point, so, take 7 and 5 from chromosome 13 and the rest of the nodes will be from the later positions from position 2 of chromosome 7. The task is illustrated below.



The task of crossover should be carefully handled so that no node is repeated in an offspring. As in each single crossover two new offspring are produced, we need to perform the above operation half of the times of the number of population. In our example the population size is 20 so we need to crossover two random chromosomes for 10 times.

3.3.4 Mutation

In simple terms, mutation may be defined as small random changes in the chromosome, to get a new candidate solution. It is used to maintain and introduce diversity in the genetic population and is usually applied with a low probability – p_m. If the probability is very high, the GA gets reduced to a random search. We will use $p_m = 1$ to get more different chromosomes.

Mutation is the part of the GA which is related to the "exploration" of the search space. It has been observed that mutation is essential to the convergence of the GA while crossover is not.

In our mutation function we take the initial population and call the crossover function again. It will give us another 20 solutions. Now mutate these 20 solutions. let's consider, we randomly choose chromosome number 7.

Chromosome 7

Now, we mutate this chromosome. We are using bit flip method to mutate the chromosome. Randomly we flip one of these gene out of 4 gene. We flipped the 3rd gene with node 6 in it. We get a new chromosome with different solution. We will get 20 different mutated chromosomes. While mutating we have to be aware so that there is no repeated node in the new mutated chromosome.

Chromosome 7(mutated)

After Complete the crossover and mutation we will have 60 solutions in hand. Now we send them to the Fitness Function.

3.7 Fitness Function

GA uses Fitness function[7] in various ways. While solving maximization problem fitness function returns the solutions with larger values whereas in minimization problem it does the opposite. Here, we are trying to get the path from source to destination at a minimum cost. So, this problem turns into minimization problem. In fitness function we will receive total 60 solutions. 20 initial, 20 from crossover and last 20 from mutation. Now the function calculates each chromosome to find the minimum cost solution. It is possible that all the solution will give us the Infinite cost. To calculate the total weight, we take a chromosome from 60 solutions. Add 1st node in the beginning and 9th node at the end of each chromosome as we ignored the presence of source and destination in the chromosomes earlier. The reason we did so is we don't want to generate such random solutions where source is not at the beginning of a chromosome and destination is not there as a last gene. These chromosomes can never be the solution as in a real solution the source node will take the first place of the candidate chromosome and the destination will take the last place. Add these nodes (source and destination) for each 60 solutions. Now we calculate each connected edge weight for each solution individually. Let's take 5 random solutions and calculate their total cost to destination from input graph. The calculation is done in the following way:

Chromosomes									**Calculation of Cost from graph**	
1	5	8	2	7	4	3	6	9		Infinite
1	8	3	2	4	5	9				Infinite
1	2	3	6	7	9				4+8+4+2+6	24
1	3	5	7	2	4	9				Infinite
1	2	8	9						4+11+7	22

Figure 2: Fitness function calculating fitness of Chromosomes

We get 60 calculated total cost and consider lowest 20 from them. Now we have found new 20 solutions with minimum path cost.

3.3 Stopping Criteria

The best 20 candidate solutions got from fitness function will be the next population and again crossover, mutation is applied to those solution. We will get another 20 solutions in which we may get a better one than the previous iteration. Now the question arises; how long the same repeated task will be continued. Is there any way to understand that we got the solution which takes minimum cost to find the destination? The answer is “NO”. So, when will we stop the algorithm? Well, the stopping criteria is problem dependent. For the shortest path problem, we think that there are possibly two ways of reaching the stopping condition. First method- algorithm has found a solution such that successive iterations no longer produce better results. The second one is- manual inspection. We are using manual inspection as stopping criteria that means manually we will fix number of iterations. Here we need to have some real guessing power. We should carefully look at the input graph and analyze its number of nodes and edges. Then guessing a number of iterations by which our proposed algorithm should return the best result. So, the success of having the optimal cost from our proposed system is fully depended on number of iterations. The more the iteration, the more the chance of getting the optimal path [8].

Chapter 4

Experiment and Analysis

4.1 Introduction

In this chapter we are going to experiment over the algorithm we implemented and we will analysis with different graphs to analysis the time complexity and performance.

4.2 Graphical Experiment

In this segment we will experiment all the algorithm with which we studied and the algorithm we have proposed for the shortest path calculation. In order to experiment that, we will use different graphs with different number of nodes and we will try to experiment all the experiment through those graphs and we will analysis the result also from which we can make a decision about the algorithms we studied. All the base graphs for experimenting all the algorithms are drawn randomly. For every graphical experiment we will highlight the basic graph, output for the algorithms and analysis the outcome from the outputs.

i) When the number of nodes is 5:

Base Graph:

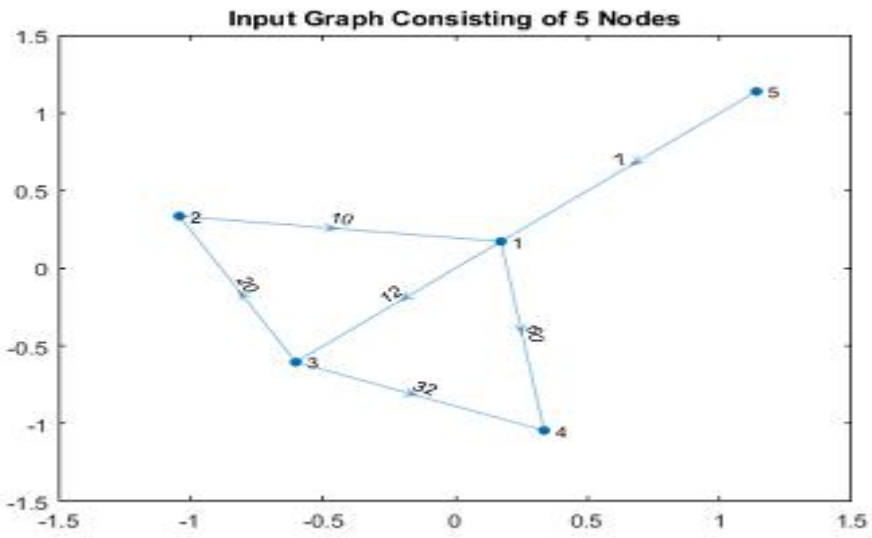


Figure 3: Input Graph

Dijkstra's algorithm:

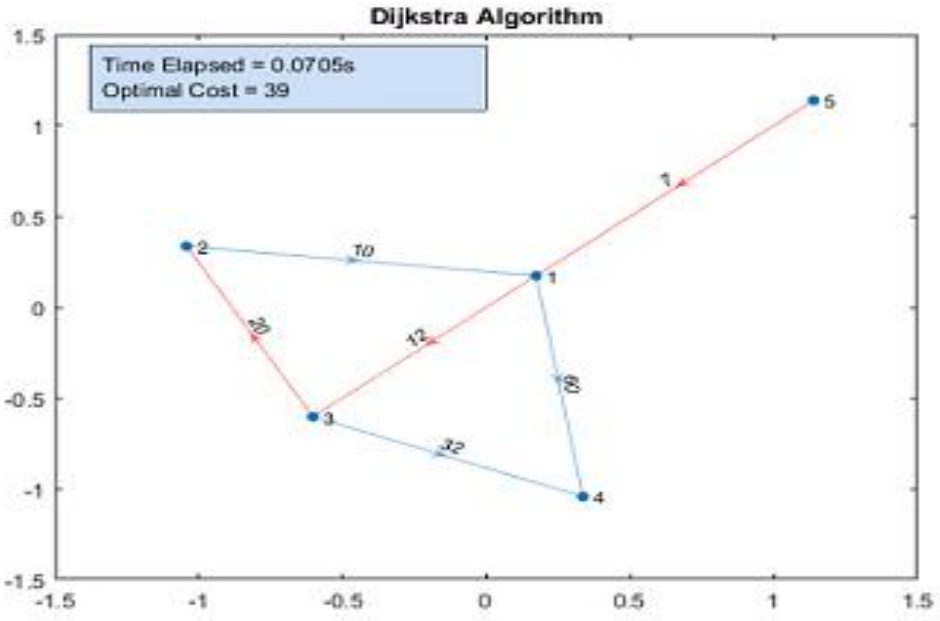


Figure 4: Dijkstra's algorithm output for base graph with elapsed time and optimal cost

Bellman Ford algorithm:

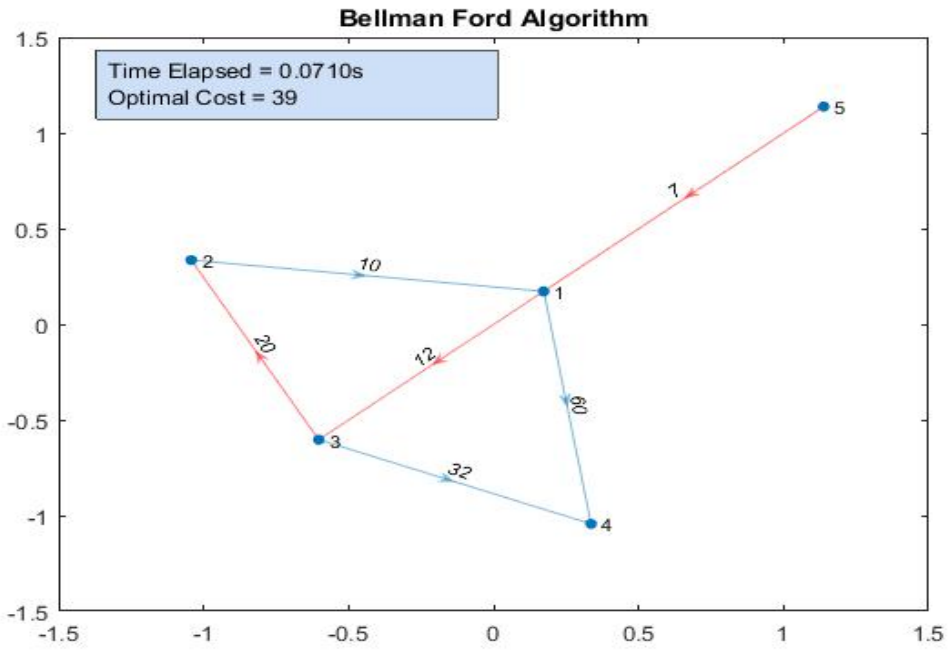


Figure 5: Bellman Ford algorithm output for base graph with time and optimal cost

Floyd Warshall Algorithm:

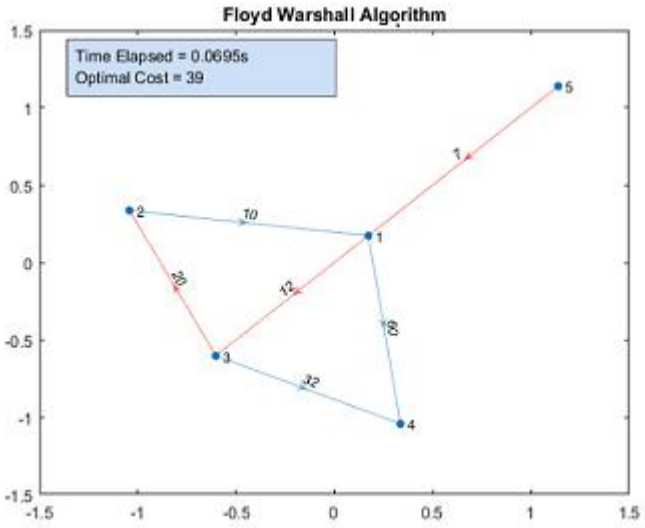


Figure 6: Floyd Warshall algorithm output for base graph with elapsed time and optimal cost

Johnson's Algorithm:

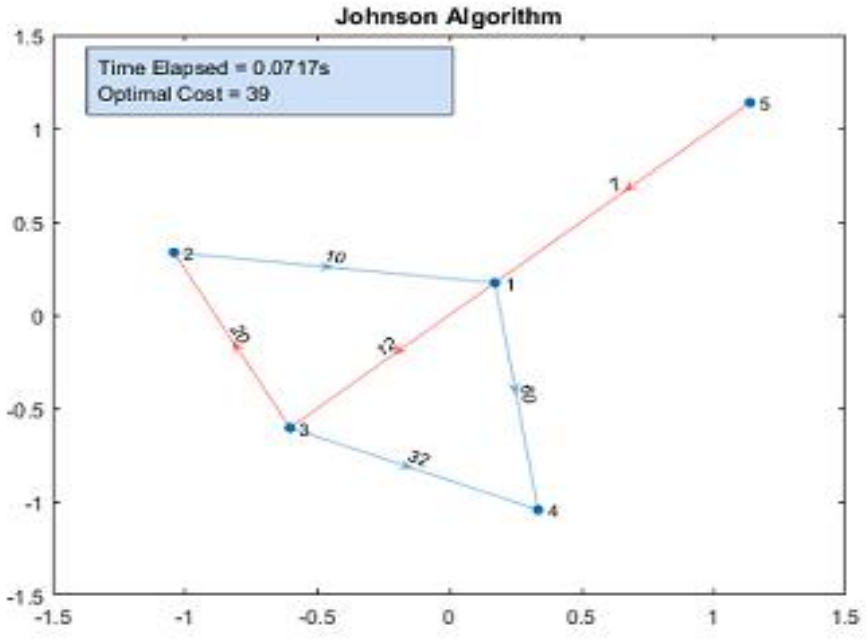
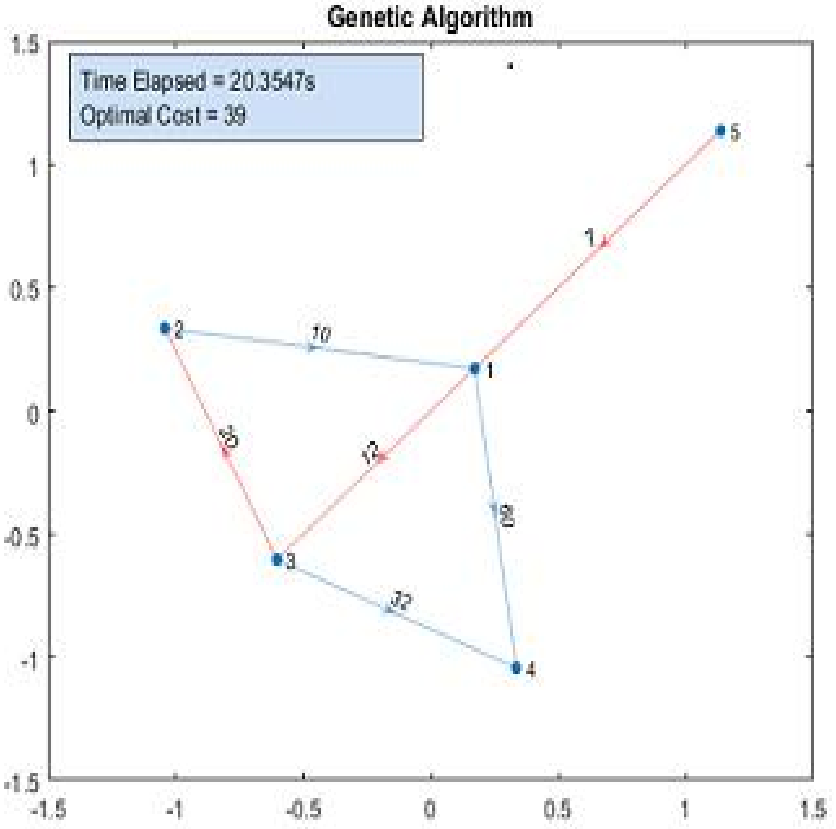


Figure 7: Johnson's algorithm output for base graph with elapsed time and optimal cost

Genetic



Algorithm:

Figure 8: Genetic algorithm output for base graph with elapsed time and optimal cost

Analysis:

From all the graph we got that every algorithm provides the same optimal cost but these are showing different times to produce the result where genetic algorithm taking the highest time whereas Floyd Warshall taking the minimum time. So from this case where node number is 5 we can definitely say that Floyd Warshall is the most efficient algorithm.

ii) When the number of nodes is 9:

Base graph:

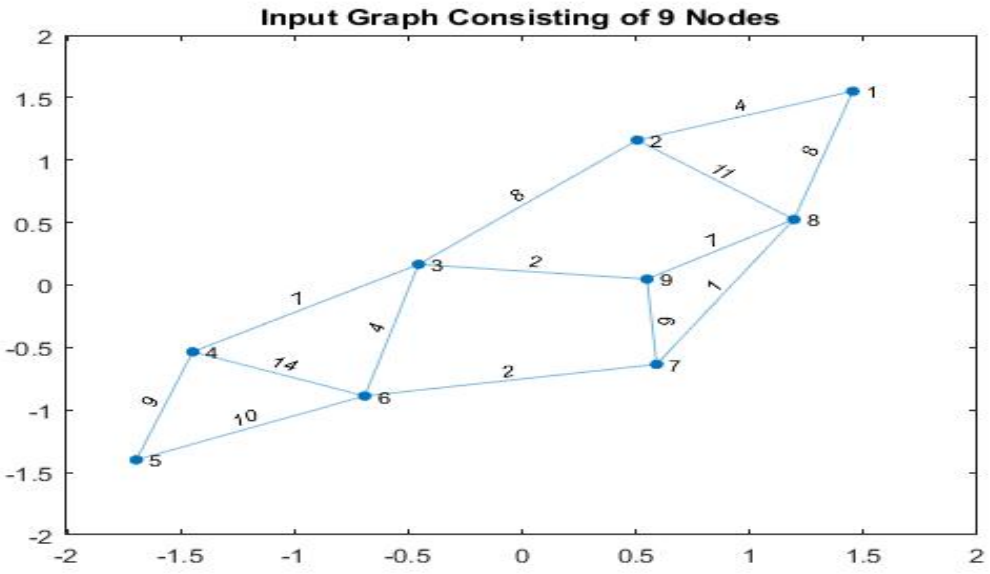


Figure 9: Input Graph

Dijkstra's algorithm:

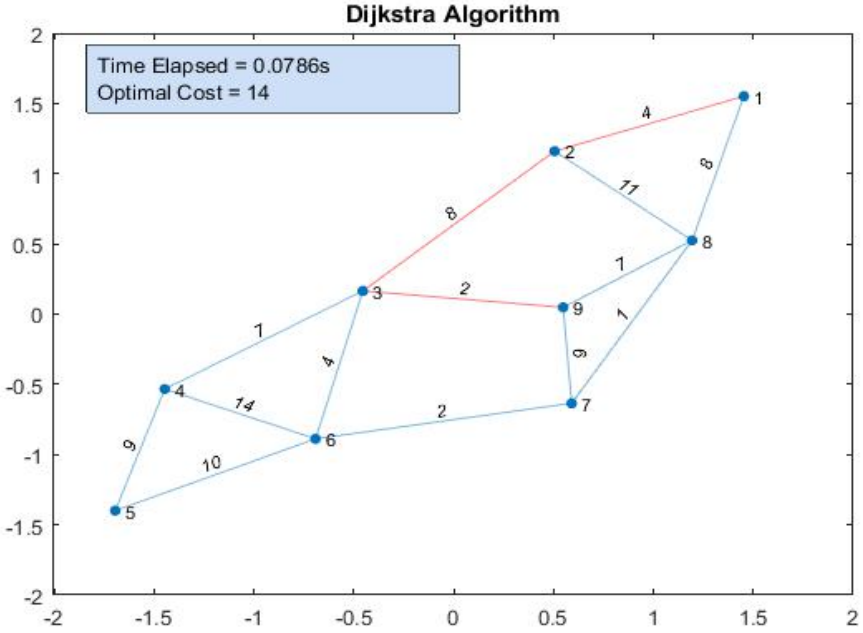


Figure 10: Dijkstra's algorithm output for base graph with elapsed time and optimal cost

Bellman Ford algorithm:

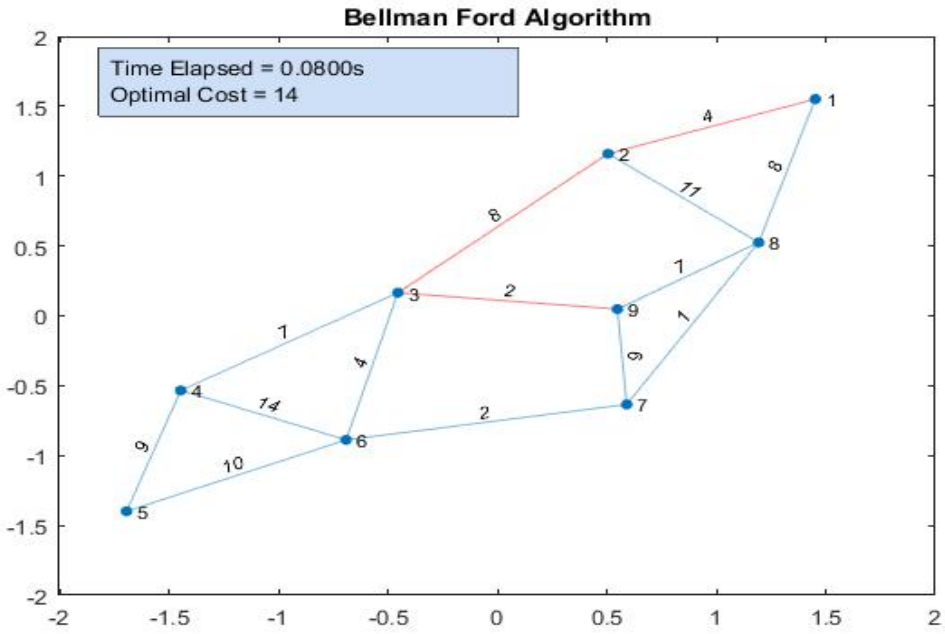


Figure 11: Bellman Ford algorithm output for base graph with elapsed time and optimal cost

Floyd Warshall Algorithm:

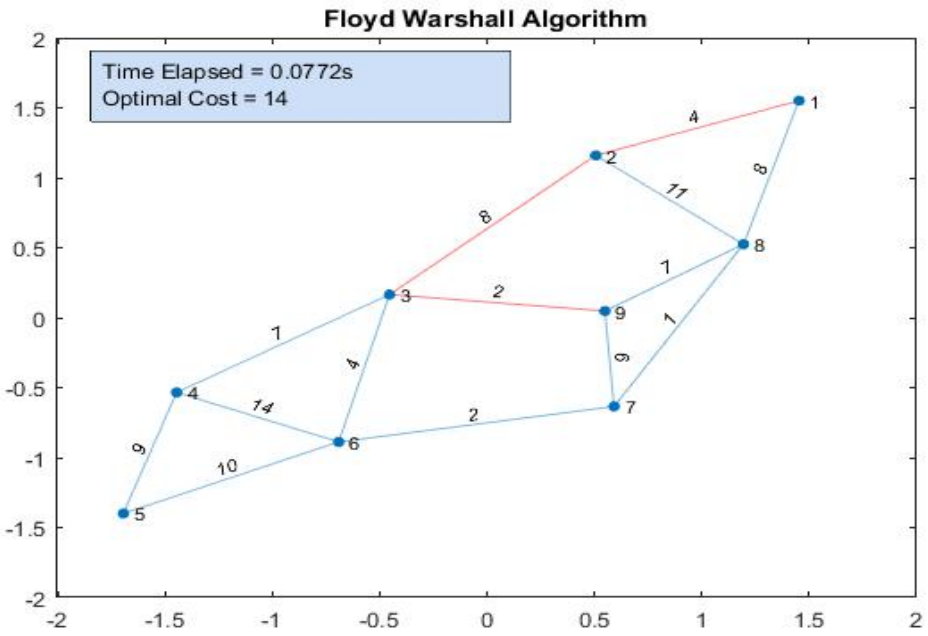


Figure 12: Floyd Warshall algorithm output for base graph with elapsed time and optimal cost

Johnson's Algorithm:

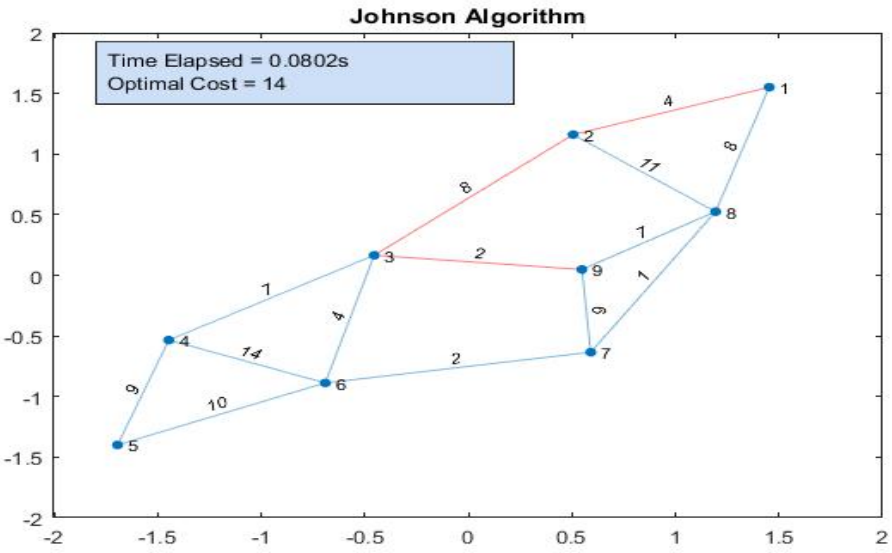


Figure 13: Johnson's algorithm output for base graph with elapsed time and optimal cost

Genetic Algorithm:

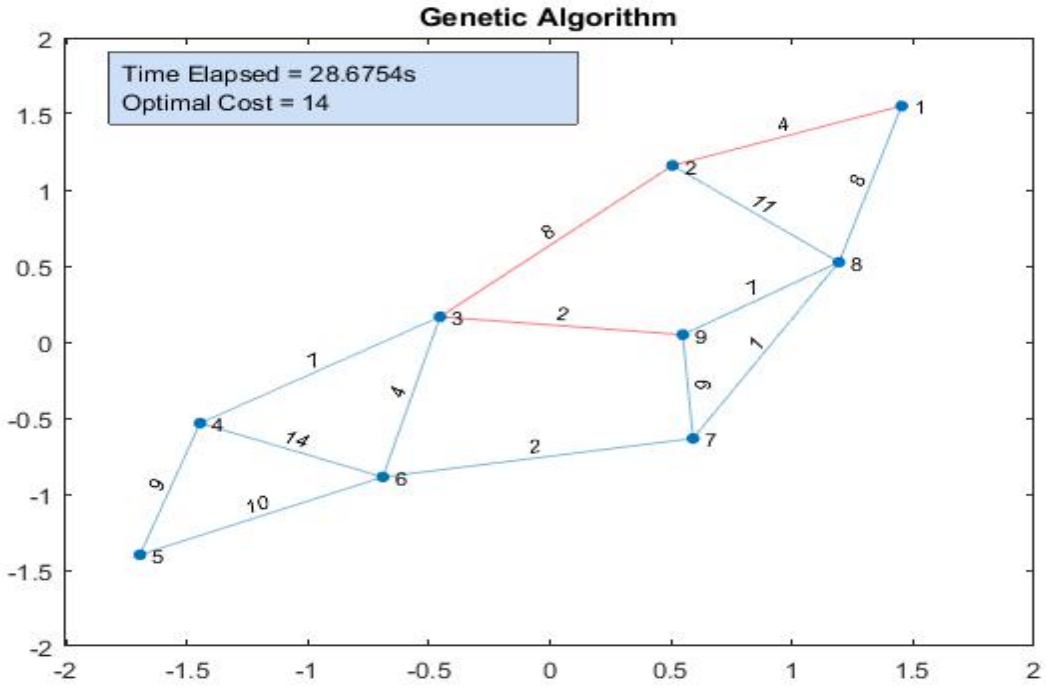


Figure 14: Genetic algorithm output for base graph with elapsed time and optimal cost

Analysis:

From all the graph we got that every algorithm provides the same optimal cost but these are showing different times to produce the result where genetic algorithm taking the highest time whereas Floyd warshall taking the minimum time. So from this case where node number is 9 we can definitely say that Floyd Warshall is the most efficient algorithm.

iii) When the number of nodes is 15:

Base Graph:

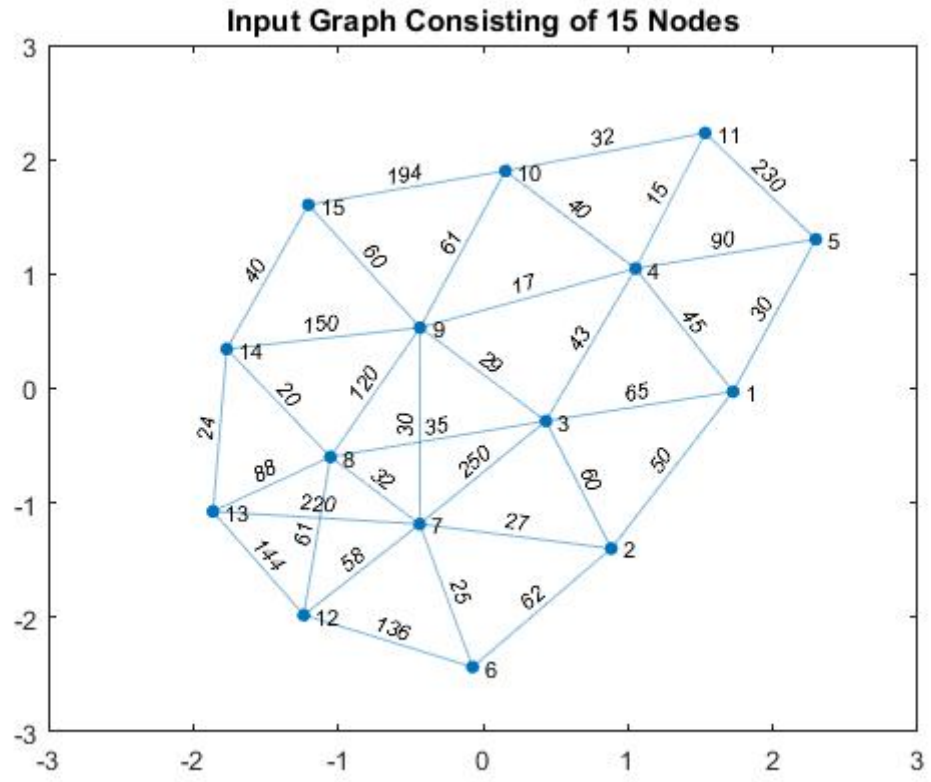


Figure 15: Input Graph

Dijkstra's algorithm:

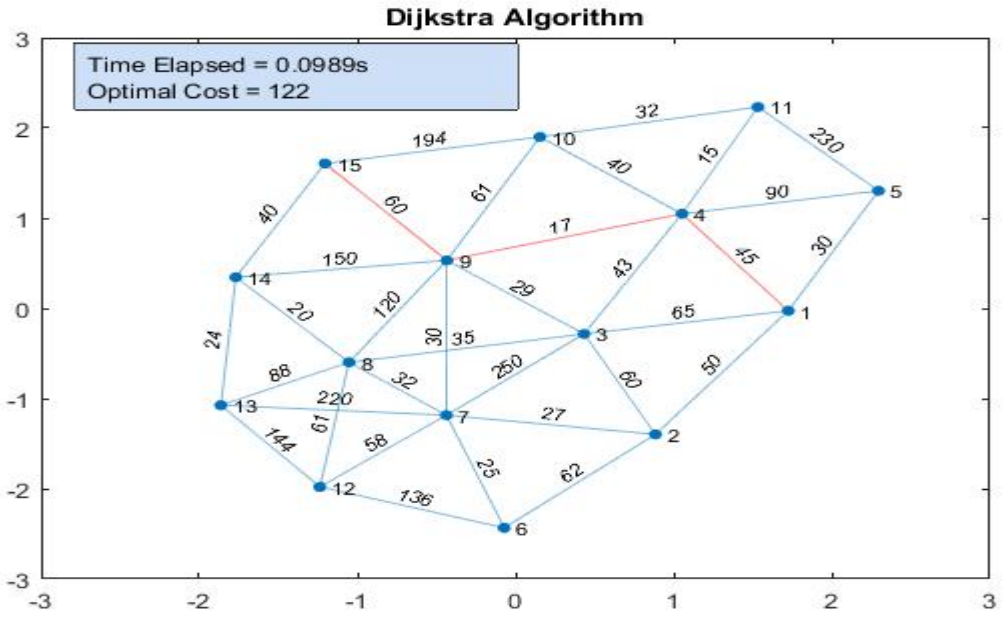


Figure 16: Dijkstra's algorithm output for base graph with elapsed time and optimal cost

Bellman Ford algorithm:

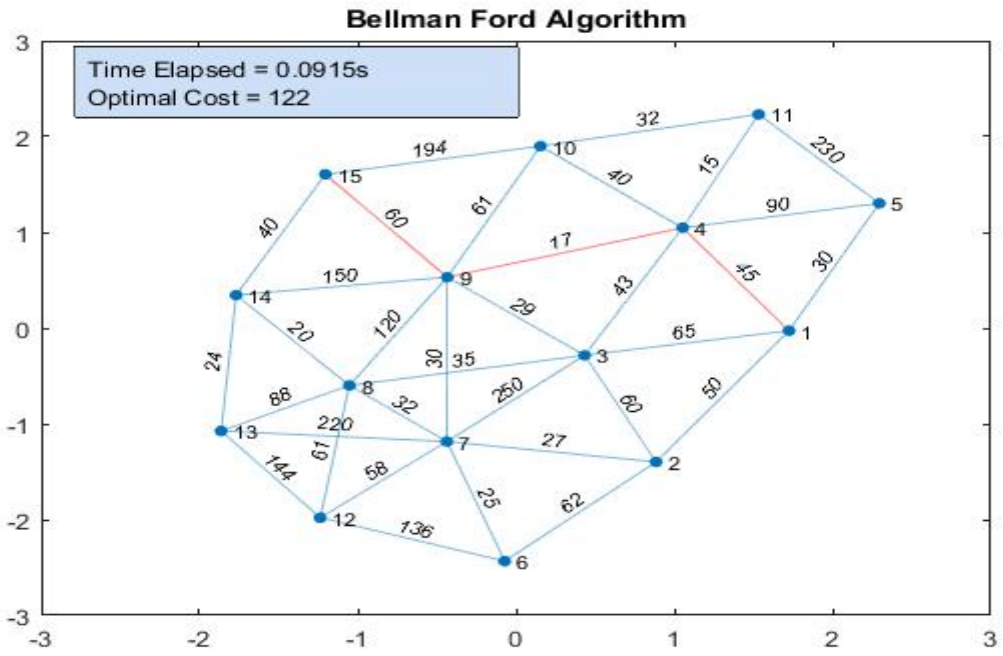


Figure 17: Bellman Ford algorithm output for base graph with elapsed time and optimal cost

Floyd Warshall Algorithm:

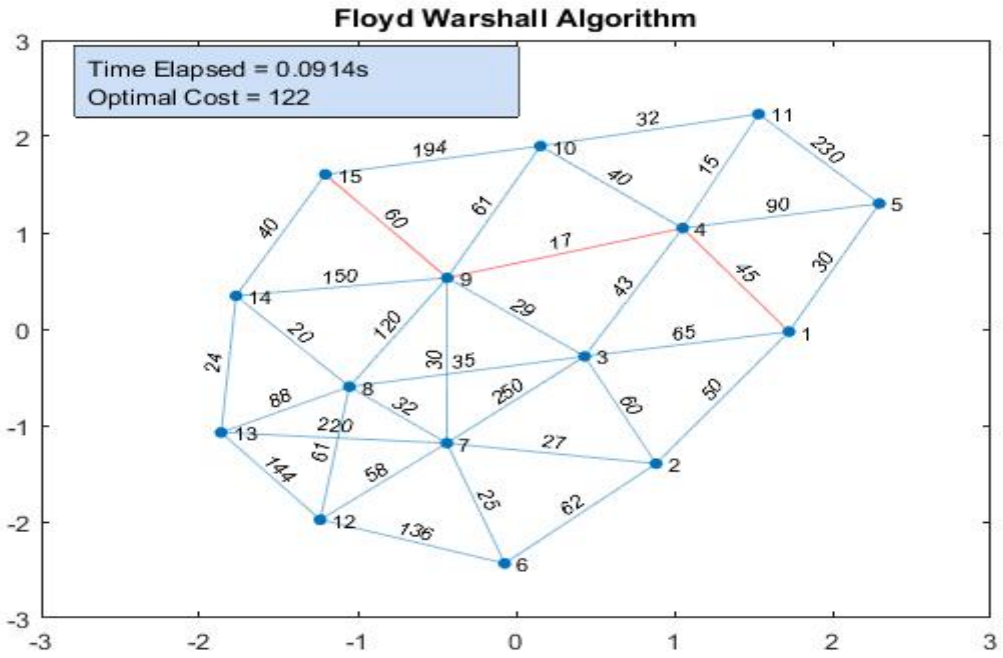


Figure 18: Floyd Warshall algorithm output for base graph with elapsed time and optimal cost

Johnson's Algorithm:

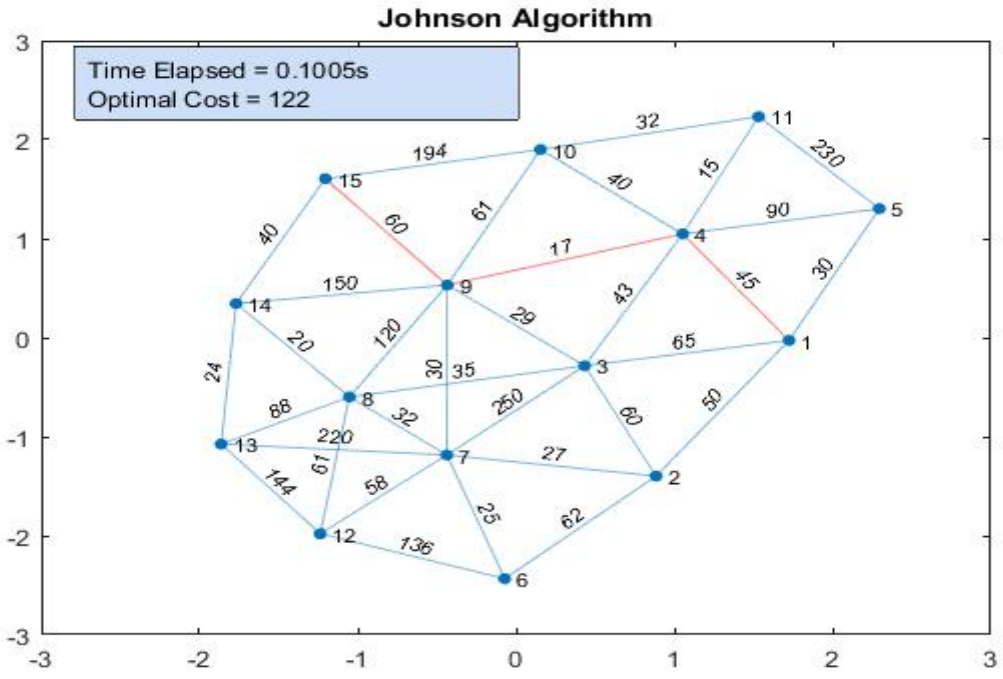


Figure 19: Johnson's algorithm output for base graph with elapsed time and optimal cost

Genetic Algorithm:

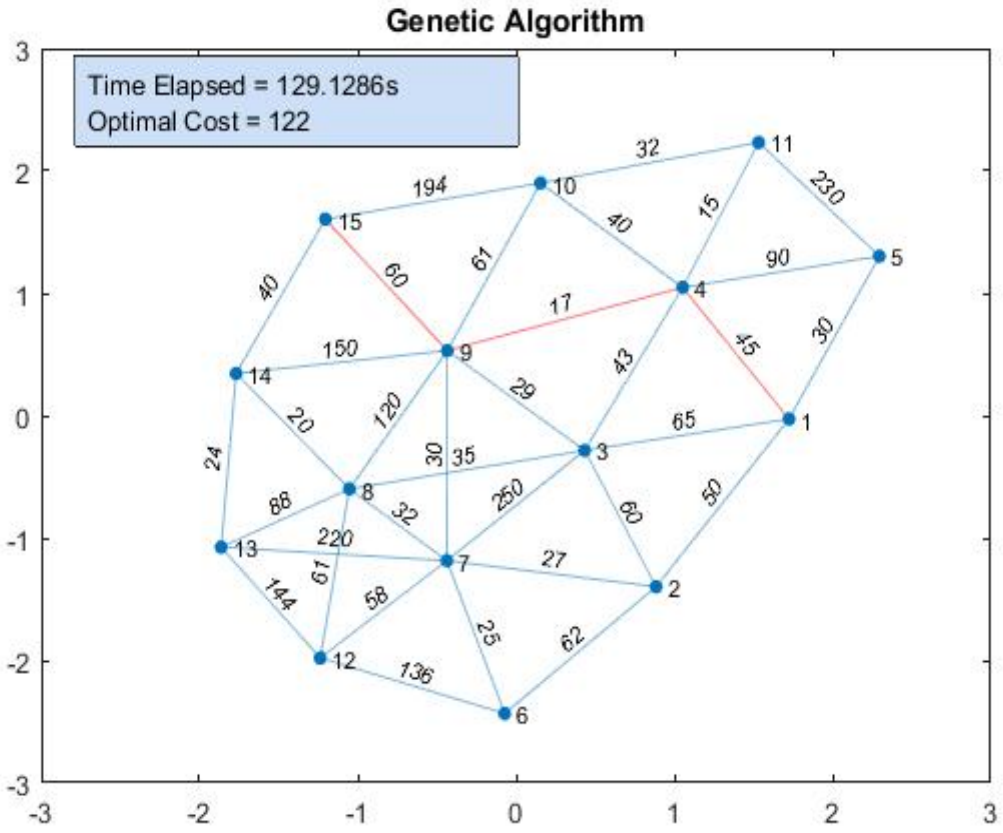


Figure 20: Genetic algorithm output for base graph with elapsed time and optimal cost

Analysis:

From all the graph we got that every algorithm provides the same optimal cost but these are showing different times to produce the result where genetic algorithm taking the highest time whereas Floyd Warshall taking the minimum time. So, from this case where node number is 15 we can definitely say that Floyd Warshall is the most efficient algorithm.

iv) When the number of nodes is 20:

Base Graph:

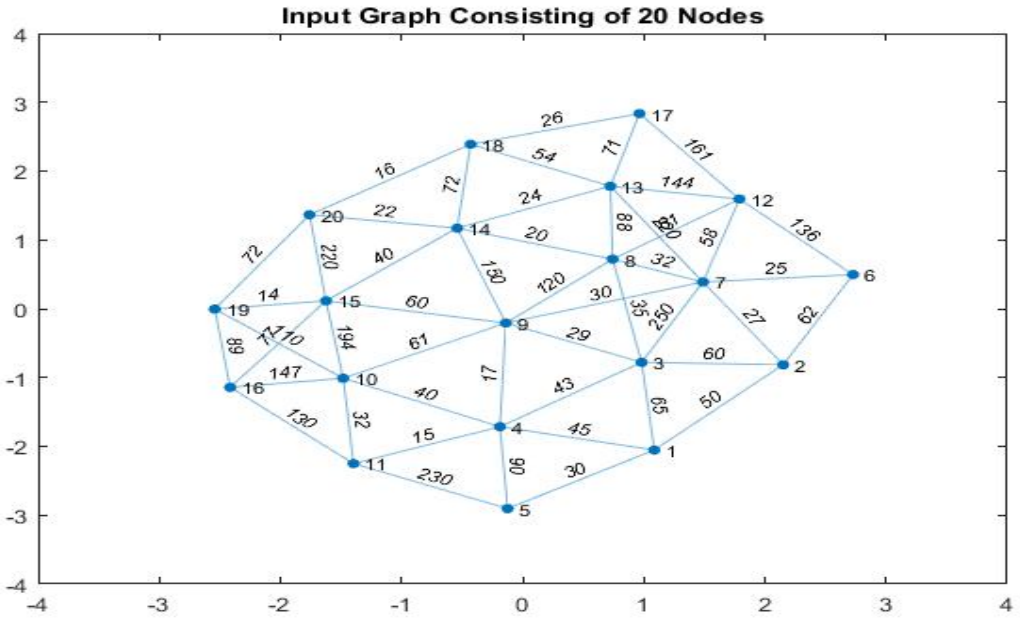


Figure 21: Input Graph

Dijkstra's algorithm:

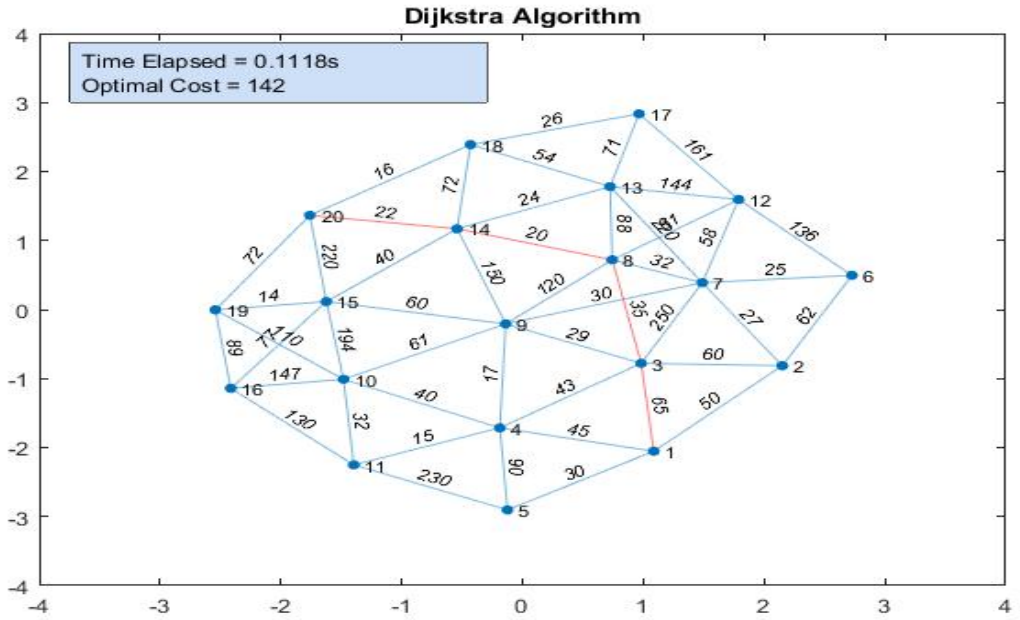


Figure 22: Dijkstra's algorithm output for base graph with elapsed time and optimal cost

Bellman Ford algorithm:

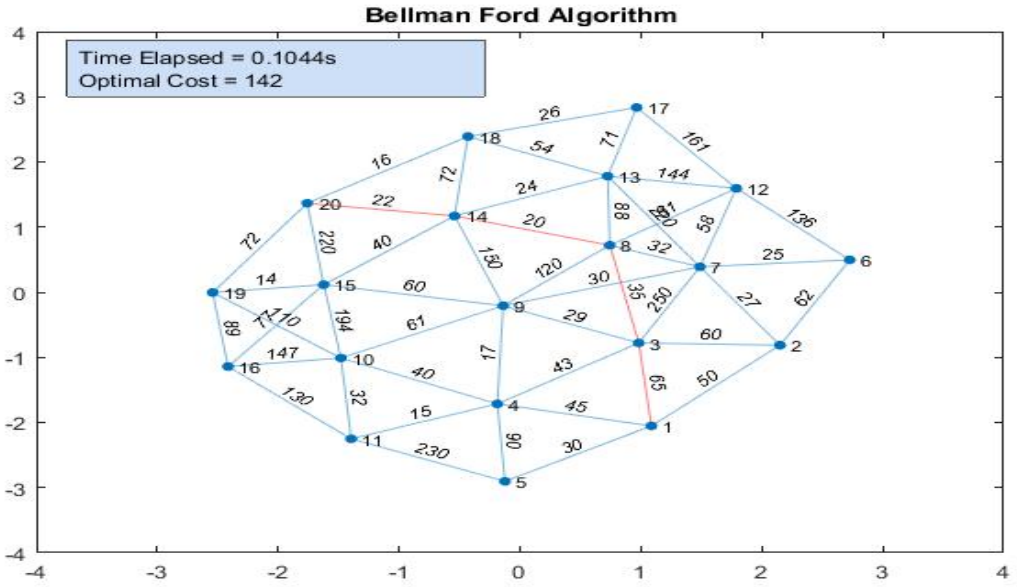


Figure 23: Bellman Ford algorithm output for base graph with time and optimal cost

Floyd Warshall Algorithm:

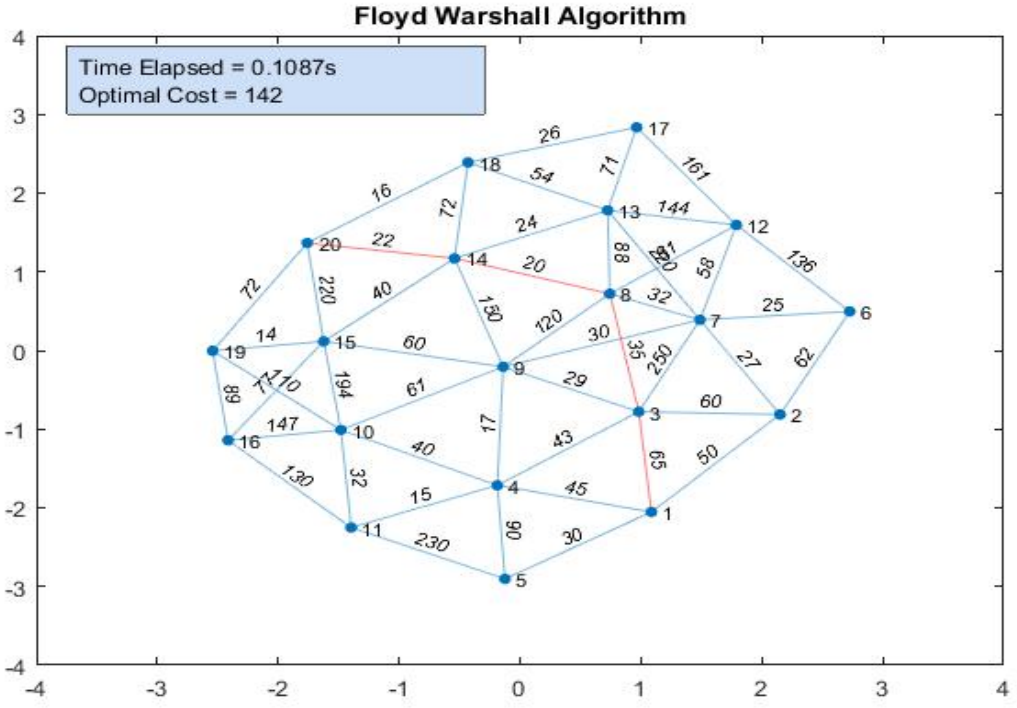


Figure 24: Floyd Warshall algorithm output for base graph with time and optimal cost

Johnson's Algorithm:

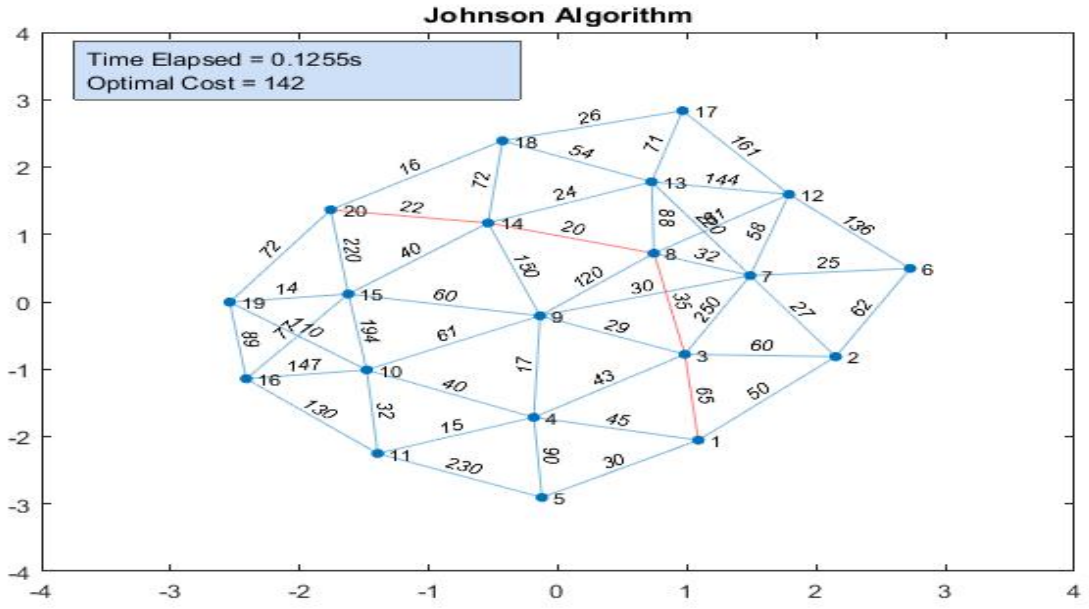


Figure 25: Johnson's algorithm output for base graph with elapsed time and optimal cost

Genetic Algorithm:

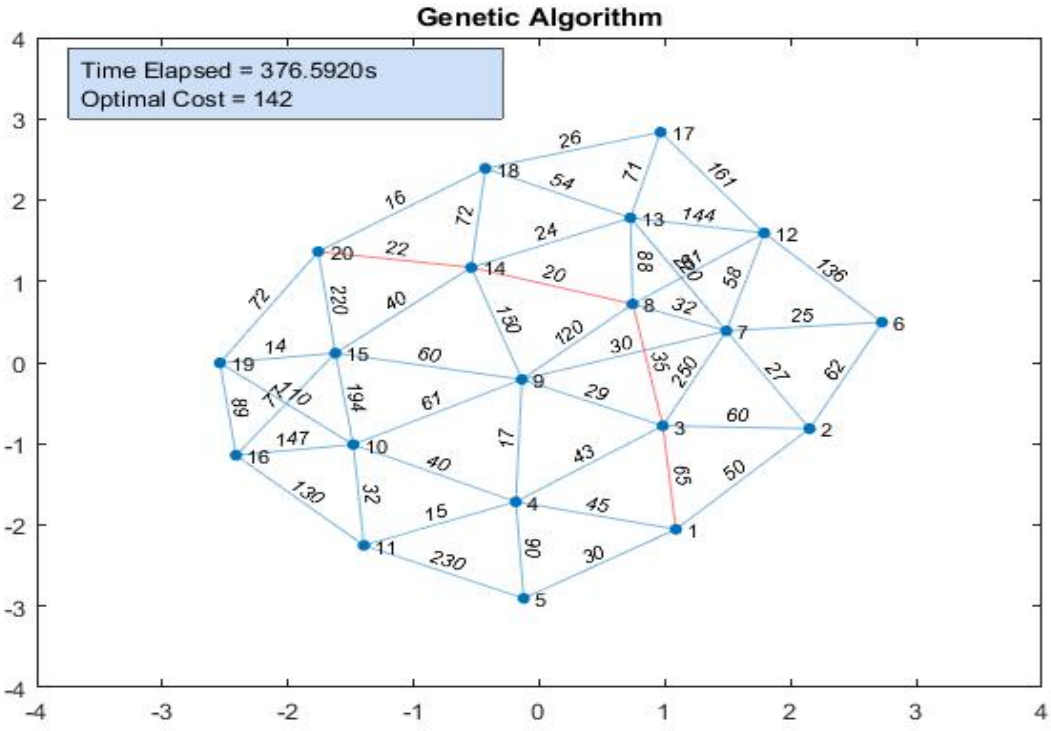


Figure 26: Genetic algorithm output for base graph with elapsed time and optimal cost

Analysis:

From all the graph we got that every algorithm provides the same optimal cost but these are showing different times to produce the result where genetic algorithm taking the highest time whereas this time Bellman Ford taking the minimum time. So, from this case where node number is 20 we can definitely say that Bellman Ford is the most efficient algorithm.

4.3 Graphical Analysis

From all the graphical experiment we can easily decide that all the algorithms provide perfect shortest path with the minimal cost and beside that we find from our experiment that Bellman Ford is the fastest among all the algorithms although there is very short difference between the other algorithm's elapsed time. It shows the minimal cost with corresponding paths with the shortest time among all the algorithms.

4.4 Time Complexity Comparison

In this segment we will show the time comparison graph for all the four cases again for which case we studied earlier.

i) When the node number is 5:

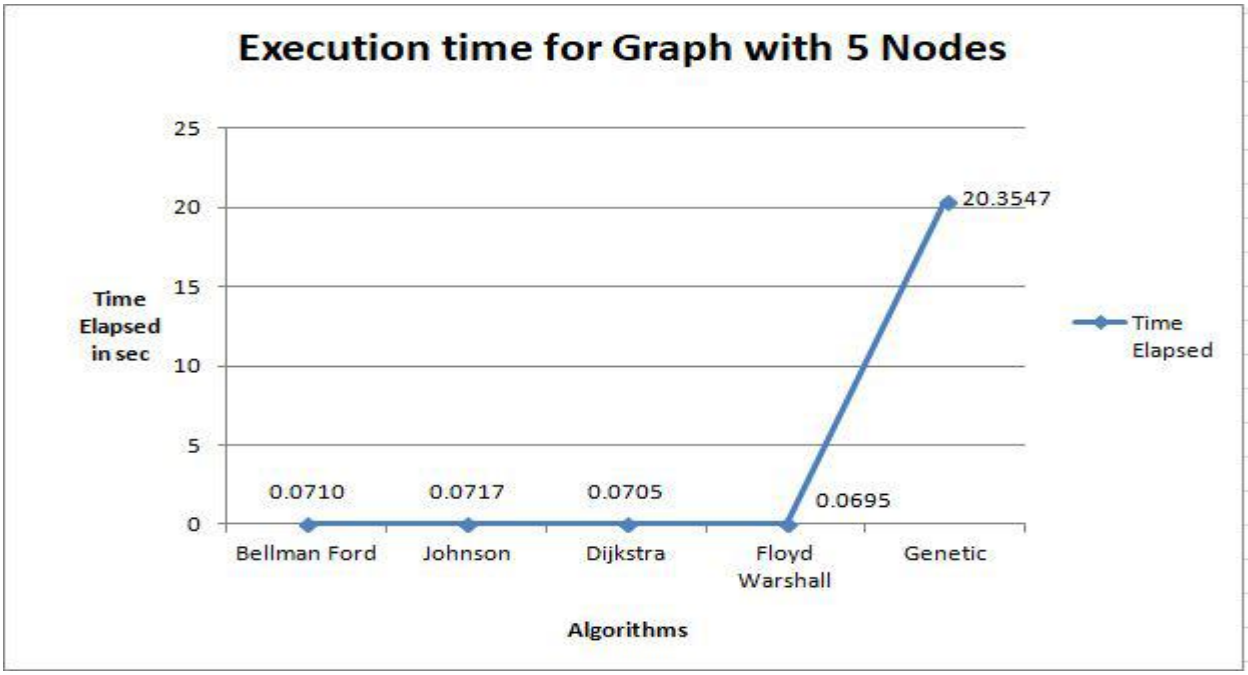


Figure 27: Time comparison graph among all the algorithm for 5 nodes

ii) When the node number is 9:

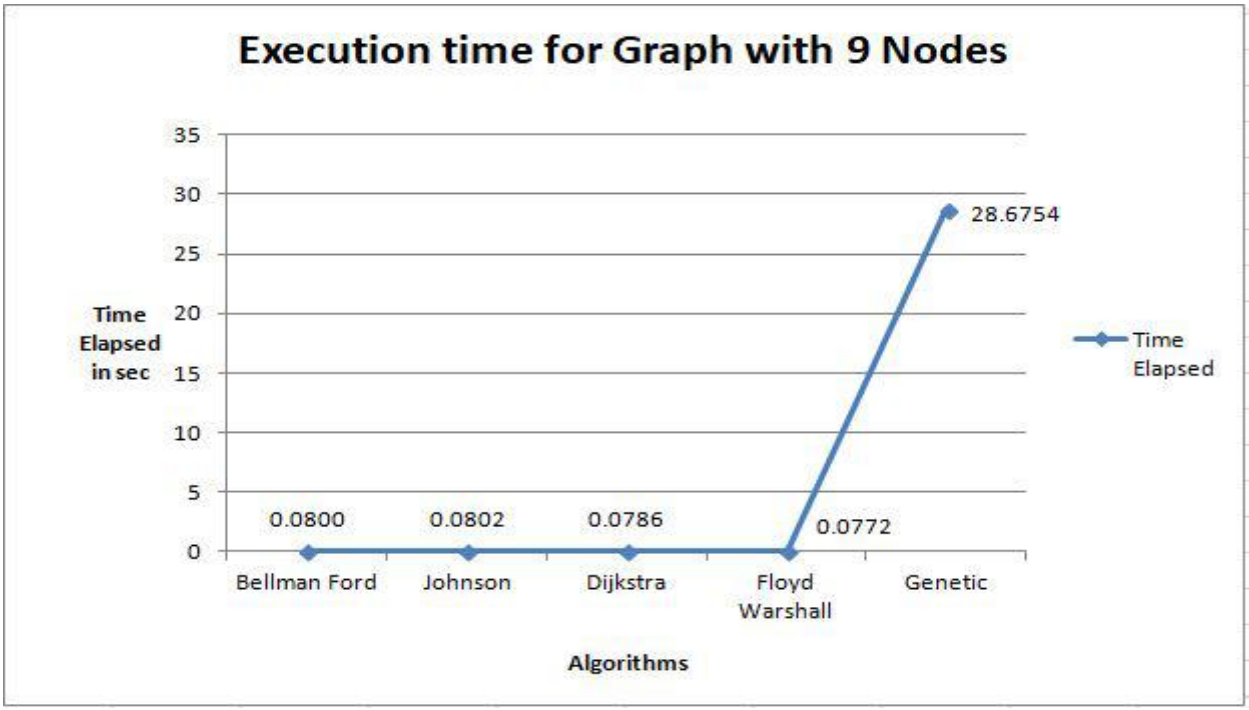


Figure 28: Time comparison graph among all the algorithm for 9 nodes

iii) When the number of node is 15:

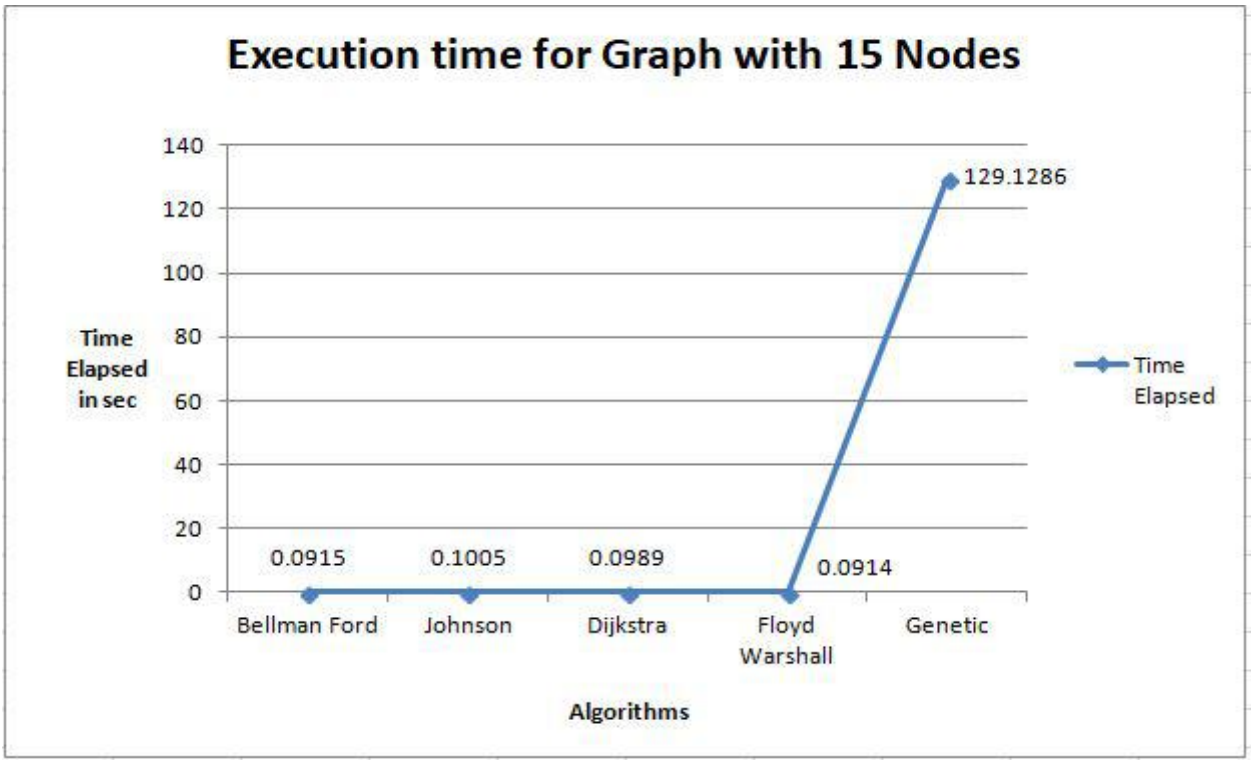


Figure 29: Time comparison graph among all the algorithm for 15 nodes

iv) When the number of node is 20:

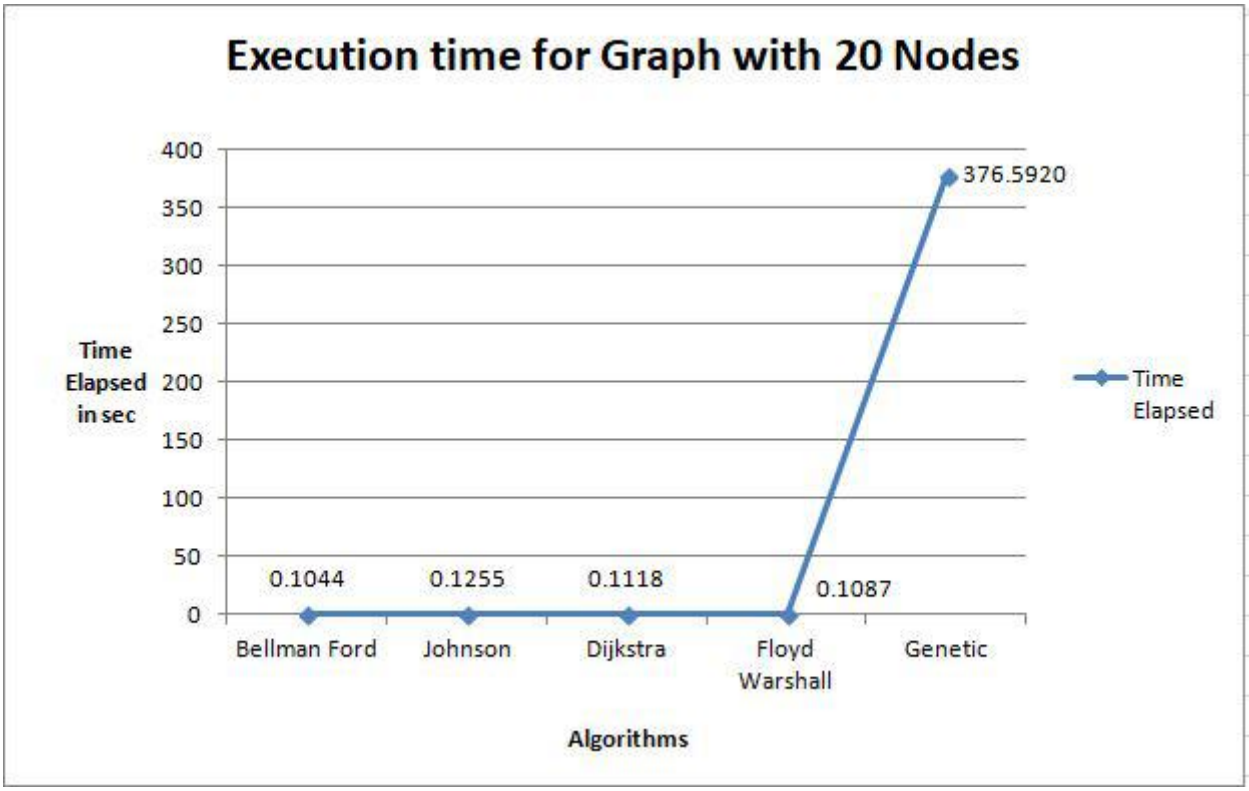


Figure 30: Time comparison graph among all the algorithm for 20 nodes

4.5 Final Result Analysis

From the overall experiment and result, except the one which we implemented by using genetic algorithm all the algorithm shows same optimal cost and take almost same elapsed time. We did not consider the factor of edges since there is no chance of big impact by the variable number of edges because from our analysis we saw genetic algorithm took a bit more time but the rest of algorithms almost took same time.

Chapter 5

Conclusion

5.1 Conclusion

We implemented some existing shortest path algorithms which are Dijkstra's Algorithm, Bellman Ford Algorithm, Floyd Warshall Algorithm, Johnson Algorithm in this paper. We also proposed a new way of solving this kind of search problem using basic genetic algorithm technique with some minor changes. We tested these algorithms along with our proposed one using input graphs with four (5,9,15,20) different number of nodes to analyze how much these algorithms take time as well as which algorithm gives better path with optimal cost. Finally, this paper concludes the following:

1. Our proposed algorithm(GA) takes the longest time to execute.
2. Genetic algorithm is the slowest among all algorithm.
3. The higher number of nodes genetic algorithm take, the more time it takes to execute.
4. The more the number of iterations in proposed approach, the more the chance of getting optimal route from source to destination.
5. Except genetic algorithm, the time differences to execute between all other algorithms are negligible.

5.2 Future Work

The different algorithms that we studied are as varied as the problems that they solve. By developing a good understanding of a large range of algorithms, we will be able to minimize the execution time of genetic algorithm. We are trying to find out a way to make the genetic algorithm faster. For this purpose, we need to have clear understanding on some more related algorithms. We are also trying to develop efficient formula to get the approximate number of iterations for our proposed approach rather than manual inspection or guessing.

References

[1] Pan Liu , Miao Huai Kou, Yin Liang and Yu Guo Ping, "A new practical algorithm for the shortest path problem," 25-27 June 2008.

[2] Kairanbay Magzhan, Hajar Mat Jani," A Review And Evaluations Of Shortest Path Algorithms," INTERNATIONAL JOURNAL OF SCIENTIFIC & TECHNOLOGY RESEARCH VOLUME 2, ISSUE 6, JUNE 2013

[3] B.S. Hasan, M.A. Khamees, and A.S.H. Mahmoud, - A Heuristic Genetic Algorithm for the Single Source Shortest Path Problem, ‖ Proc. of International Conference on Computer Systems and Applications, pp. 187-194, 2007

[4] M.Thorup. Undirected single source shortest path in linear time. In Proceeding softhe 38th Annual Symposium on Foundations of Computer Science (FOCS-97), pages 12–21, Los Alamitos, October 20–22 1997. IEEE Computer Society Press.

[5] J. Chamero, - Dijkstra's Algorithm‖ Discrete Structures & Algorithms, 2006

[6] James B. Orlin, Kamesh Madduri, K. Subramani, and M. Williamson. 2010. "A faster algorithm for the single source shortest path problem with few distinct positive lengths. J. of Discrete Algorithms 8," 2nd June 2010

[7] Kramer, O. "**Genetic Algorithm Essentials**," 2nd June 2017

[8] D. Lozano, L.A. Galindo, and L. Garcia, "WIMS 2.0: Converging IMS and Web 2.0. Designing REST APIs for the exposure of session-based IMS capabilities," in Proc. of The Second International Conference on Next Generation Mobile Applications, Services, and Technologies, 2008, pp. 18–24.

Appendix

The actual codes that were implemented have been listed below:

```
function bellman(no_of_vertices,src,dest,edges,edge_start, edge_connected, edge_weight,G)
figure('name','Bellman Ford Algorithm');
 dg=plot(G,'EdgeLabel',G.Edges.Weight);
 hold on;

[d,p]=initdp(no_of_vertices);

d(src)=0;

for i=1:(no_of_vertices-1)
    [nc,d,p]=bellman_algo(edges,edge_start,edge_connected,edge_weight,d,p);
end
if nc==1
    disp('negative cycle exists');
else
    for i=1:no_of_vertices
        if(i==dest)
            fprintf('\nminimal_cost = \n\n\t\t%d',d(i));
            fprintf('\n\nPath: \n\t');
            graph_print(i,p,edge_weight,src,dg);
            fprintf('\n');
        end
    end
end
end
```

```
function [value,d,p]=bellman_algo(edges,edge_start,edge_connected,edge_weight,d,p)
    value=0;
    for i=1:edges
        [d,p]=relax(edge_start(i),edge_connected(i),edge_weight(i),d,p);
    end
    for i=1:edges
        if d(edge_connected(i))>(d(edge_start(i))+edge_weight(i))
            value=1;
            break;
        end
    end
end
```

```
function compare_algo()
prompt = 'Directed or UnDirected Graph? [d=Directed/ANY KEY=UnDirected]: ';
strg = input(prompt,'s');
prompt = 'How Many Vertices?\n ';
no_of_vertices = input(prompt);
nov=no_of_vertices;
for i=1:nov
    for j=1:no_of_vertices
        D(i,j)=0;
    end
end
edge_start=[];
s=[];
t=[];
psize=20;
graph_edge=[];
edge_connected=[];
edge_weight=[];
edges=0;
prompt = 'create edges ';
while edges<(no_of_vertices*(no_of_vertices-1))
    prompt = 'enter first node';
    first_node=input(prompt);
    prompt = 'enter connected node';
    second_node=input(prompt);
    prompt = 'enter weight';
    weight=input(prompt);
    if first_node<1 || first_node>no_of_vertices || second_node<1 || second_node>no_of_vertices || first_node==second_node
        disp( 'enter valid edges')
        continue;
    end
    edge_start=[edge_start;first_node];
    s=[s first_node];
    edge_connected=[edge_connected;second_node];
    t=[t second_node];
    edge_weight=[edge_weight;weight];
    graph_edge=[graph_edge weight];
    edges=edges+1;
```

```
        D(first_node,second_node)=weight;
        prompt = 'Do you want more edges? YES/NO [y/ANY KEY]: ';
        str = input(prompt,'s');
        if str=='y'
            continue;
        else
            break;
        end
    end
    for i=1:nov
        D(i,i)=0;
    end
    if strg=='d'
        G=digraph(s, t,graph_edge);
    else
        G=graph(D);
    end
    plot(G,'EdgeLabel',G.Edges.Weight);
    prompt = 'enter source vertex?\n ';
    src = input(prompt);
    prompt = 'enter destination vertex?\n ';
    dest = input(prompt);

    prompt = 'enter Max iteration for genetic algo?\n ';
    max = input(prompt);

    tic
    modified_ga(no_of_vertices,psize,src,dest,edge_start, edge_connected, edge_weight,edges,G,max);
    timeElapsed_GA = toc

    tic
    bellman(no_of_vertices,src,dest,edges,edge_start, edge_connected, edge_weight,G);
    timeElapsed_bellman = toc

    tic
    Dijkstra(no_of_vertices,src,dest,G,D);
```

```
timeElapsed_dijkstra = toc


tic
johnson(no_of_vertices,src,dest,edges,edge_start, edge_connected, edge_weight,G,D);
timeElapsed_johnson = toc


tic
fw(no_of_vertices,src,dest,G,D);
timeElapsed_floydWarshall = toc

end
```

```
function c=crossOver(q,row)
    c=[];
    j=ceil(row/2);
    c1=[];
    c2=[];
    d=1;
    i=0;
    while(i<j)
        r1=randi(row);
        r2=randi(row);
        while(1)
            if r1~=r2
                break;
            else
                r2=randi(row);
            end
        end
        a=cell2mat(q(r1));
        b=cell2mat(q(r2));
        lena=size(a,2);
        lenb=size(b,2);
        if lena<lenb
            len=lena;
        else
            len=lenb;
        end
        if len>0
            point=randi(len);
            count=0;
            while(1)
                check=intersect(a(1:point),b(point+1:end));
                check1=intersect(b(1:point),a(point+1:end));
                if isempty(check)==1 && isempty(check1)==1
                    c1=[a(1:point) b(point+1:end)];
                    c2=[b(1:point) a(point+1:end)];
                    c{d}=c1;
                    c{d+1}=c2;
                    d=d+2;
```

```
                    i=i+1;
                    break;
                else
                    point=randi(len);
                    count=count+1;
                    if count==100
                        break;
                    end
                end
            end
        end
    end
end
```

```
function Dijkstra(no_of_vertices,source,dest,G,D)
 figure('name','Dijkstra Algorithm');
 dg=plot(G,'EdgeLabel',G.Edges.Weight);
 hold on;
    n=size(D,1);
    o=n;

    for i=1:n
        dist(i)=intmax;
        prev(i)=-1;
        selected(i)=0;
    end

    selected(source)=1;
    dist(source)=0;
    start=source;
     extracted=[];
   while sum(selected)~=n
        m=source;
        for i=1:n
            if D(start,i)~=0
                d=dist(start) + D(start,i);
                if(d<dist(i)&& selected(i)==0)
                    dist(i)=d;
                    prev(i)=start;
                    extracted=[extracted d];
                end
            end
        end
        minval=min(extracted);
        m=find(minval==dist);
        if length(m)>1
            for i=1:n
                if selected(i)~=1
                    m=i;
                end
            end
        end
```

```
39          midx=find(extracted==min(extracted));
40          selected(start)=1;
41          start=m;
42          extracted(midx)=[];
43      end
44      start=dest;
45      j=1;
46      minimal_cost=0;
47      while true
48          path(j)=start;
49          j=j+1;
50          start=prev(start);
51
52          if(start==-1)
53              break;
54          end
55
56      end
57      path=fliplr(path);
58
59      for i=1:j-2
60          minimal_cost=minimal_cost+D(path(i),path(i+1));
61          highlight(dg,[path(i) path(i+1)],'EdgeColor',[1 0 0]);
62      end
63     minimal_cost
64     disp('Path:');
65     disp(path);
66  end
```

```
1   function Dijkstra_johnson(no_of_vertices,source,dest,G,D,E)
2    figure('name','Johnson Algorithm');
3    dg=plot(G,'EdgeLabel',G.Edges.Weight);
4    hold on;
5       n=size(D,1);
6       o=n;
7
8       for i=1:n
9           dist(i)=intmax;
10          prev(i)=-1;
11          selected(i)=0;
12      end
13
14      selected(source)=1;
15      dist(source)=0;
16      start=source;
17     extracted=[];
18     while sum(selected)~=n
19          m=source;
20          for i=1:n
21              if D(start,i)~=-intmax
22                  d=dist(start) + D(start,i);
23                  if(d<dist(i)&& selected(i)==0)
24                      dist(i)=d;
25                      prev(i)=start;
26                      extracted=[extracted d];
27                  end
28              end
29
30          end
31         minval=min(extracted);
32          m=find(minval==dist);
33          if length(m)>1
34              for i=1:n
35                  if selected(i)~=1
36                      m=i;
37                  end
38              end
```

```
38              end
39          end
40          midx=find(extracted==min(extracted));
41          selected(start)=1;
42          start=m;
43          extracted(midx)=[];
44
45      end
46      start=dest;
47      j=1;
48      minimal_cost=0;
49      while true
50          path(j)=start;
51          j=j+1;
52          start=prev(start);
53
54          if(start==-1)
55              break;
56          end
57
58      end
59      path=fliplr(path);
60
61      for i=1:j-2
62          minimal_cost=minimal_cost+E(path(i),path(i+1));
63          highlight(dg,[path(i) path(i+1)],'EdgeColor',[1 0 0]);
64      end
65      minimal_cost
66      disp('Path:');
67      disp(path);
68
69  end
```

```
function fw(no_of_vertices,source,dest,G,D)
figure('name','Floyd Warshall Algorithm');
 dg=plot(G,'EdgeLabel',G.Edges.Weight);
 hold on;

    M=D;
    for i=1:size(D,1)
        for j=1:size(D,1)
            if i~=j && M(i,j)==0
                M(i,j)=intmax;
            end

        end
    end

    n=size(D,1);
    for i=1:n
        for j=1:n
            if(i==j)
                path(i,j)=i;
            else
                path(i,j)=j;
            end


        end
    end

    n=size(D,1);
    for k=1:n

            for i=1:n
                for j=1:n
                        if(M(i,j)>M(i,k)+M(k,j))
                        M(i,j)=min( M(i,k)+M(k,j));
                        path(i,j)=path(i,k);
                        end
                    end

                end
    end

    index=2;
    Npath(1)= source;
    minimal_cost=0;
    while 1==1
        Npath(index)= path(source,dest);
        if(Npath(index)~=dest)
            source=Npath(index);
            index=index+1;
        else
            break;
        end

    end

    n=size(Npath,2);
    for i=1:n-1
        minimal_cost=minimal_cost+D(Npath(i),Npath(i+1));
        highlight(dg,[Npath(i) Npath(i+1)],'EdgeColor',[1 0 0]);
    end
    minimal_cost
    disp('Path:');
    disp(Npath);
end
```

```
function value=getQuality(a,edge_start, edge_connected, edge_weight,src,dest)
    value=0;
    ba=[1 3 8 14 20];
    a=[src,a(1:end),dest];
     if isequal(a,ba)==1
         disp('hiiiiiiiiiiii')


     end
    i=1;
    while(i<=size(a,2)-1)
        j=i+1;
        match=find(a(i)==edge_start);
        match2=find(a(j)==edge_connected);
        if isempty(match)~=1 && isempty(match2)~=1
          if isempty(intersect(match,match2))==1
            value=value+intmax;
            break;
          else
              idx=intersect(match,match2);
              value=value+edge_weight(idx);
          end
        else
            value=value+intmax;
        end

       i=i+1;
    end

end
```

```
function graph_print(i,p,edge,s,r)
    if p(i)==-1
         fprintf('%d\t\t',i);
        return;
    else
          graph_print(p(i),p,edge,s,r)
          highlight(r,[p(i) i],'EdgeColor',[1 0 0]);
          fprintf('%d\t\t',i);
    end


end
```

```
function [d,p] = initdp(vertices)
    d=[];
    p=[];
    for i=1:vertices
        d=[d;intmax];
        p=[p;-1];
    end

end
```

```
function johnson(no_of_vertices,src,dest,edges,edge_start, edge_connected, edge_weight,G,D)
E=D;
nov=no_of_vertices;
for i=1:nov
    for j=1:no_of_vertices
       if D(i,j)==0
          D(i,j)=-intmax;
       end
    end
end
new_edge_start=edge_start;
new_edge_connected=edge_connected;
new_edge_weight=edge_weight;
new_edges=edges;
for i=1:nov
    new_edge_start=[new_edge_start;nov+1];
    new_edge_connected=[new_edge_connected;i];
    new_edge_weight=[new_edge_weight;0];
    new_edges=new_edges+1;
end
newsrc=nov+1;
[d,p]=initdp(no_of_vertices+1);
d(newsrc)=0;
for i=1:(no_of_vertices)
    [nc,d,p]=bellman_algo(new_edges,new_edge_start,new_edge_connected,new_edge_weight,d,p);
end
if nc==1
    disp('negative cycle exists');
else
    for i=1:(no_of_vertices)
        h(i)=d(i);
    end

    for i=1:nov
        for j=1:no_of_vertices
           if D(i,j)~=-intmax
              D(i,j)=D(i,j)+h(i)-h(j);
           end
        end
    end
```

```
41
42    Dijkstra_johnson(no_of_vertices,src,dest,G,D,B);
43  end
44  end
45
```

```
1   function modified_ga(no_of_vertices,psize,src,dest,edge_start, edge_connected, edge_weight,edges,G,max)
2   figure('name','Genetic Algorithm');
3   dg=plot(G,'EdgeLabel',G.Edges.Weight);
4   hold on;
5   p=[];
6   for i=1:psize
7       pnodes=randi(no_of_vertices-2);
8           p{i} =[randsample(setdiff(1:no_of_vertices, [src,dest]), pnodes)];
9
10  end
11
12    t=0;
13    minimal_cost=intmax;
14    arr1=[];
15    arr2=[];
16    while(t<max)
17        pc=crossOver(p,psize);%%%crossover done
18        pm=mutation(p,psize,src,dest,no_of_vertices);
19        [p,mincost]=Select(p,pc,pm,psize,edge_start, edge_connected, edge_weight,src,dest,no_of_vertices);
20        [j k]=min(mincost);
21        minpath=cell2mat(p(k));
22        minpath=[src,minpath,dest];
23        if j<minimal_cost
24            minimal_cost=j;
25            arr1=[j];
26            arr2=[minpath];
27         end
28
29        t=t+1;
30    end
31      minimal_cost
32      disp('Path:');
33      disp(minpath);
34
35      n=size(minpath,2);
36      for i=1:n-1
37          highlight(dg,[minpath(i) minpath(i+1)],'EdgeColor',[1 0 0]);
38      end
```

```
function m=mutation(q,row,src,dest,no_of_vertices)
    c_over=crossOver(q,row);
    m=[];
    for i=1:row
        a=cell2mat(c_over(i));
        if size(a,2)~=0
            if length(a(1,:))~=no_of_vertices-2
                len=length(a(1,:));
                flip_pos=randi(len);
                flip_value=randsample(setdiff(1:no_of_vertices, [src,dest]), 1);
                count=0;
                while(1)
                    check=intersect(a(1:end),flip_value);
                    if a(flip_pos)~=flip_value && isempty(check)==1
                        a(flip_pos)=flip_value;
                        break;
                    else
                        count=count+1;
                        flip_value=randsample(setdiff(1:no_of_vertices, [src,dest]), 1);
                        if count == 200
                            break;
                        end
                    end
                end
            end
            m{i}=a;
        end
    end

end
```

```
function print_path(i,p)
    if p(i)==-1
        disp(i);
        return;
    else
        print_path(p(i),p);
        disp(i);
    end

end
```

```
function [d,p]=relax(edge_start,edge_connected,edge_weight,d,p)
    if d(edge_connected)>(d(edge_start)+edge_weight)
        d(edge_connected)=d(edge_start)+edge_weight;
        p(edge_connected)= edge_start;
    end


end
```

```
function [s,mincost]=Select(p,pc,pm,psize,edge_start, edge_connected, edge_weight,src,dest,no_of_vertices)
    s={};
    mincost=[];
    visited={};
    r=[];
    k=0;
    r=[p;pc;pm];
    for i=1:psize
        minval=intmax;
        for j=1:size(r,1)
            for k=1:size(r,2)
                if isempty(r(j,k))~=1
                    a=cell2mat(r(j,k));
                    kl=[2 3];
                    quality=getQuality(a,edge_start, edge_connected, edge_weight,src,dest);
                    if quality<minval
                        minval=quality;
                        rem1=j;
                        rem2=k;
                    end
                end
            end

        end

        if minval==intmax
            s{i} =[randsample(setdiff(1:no_of_vertices, [src,dest]), randi(no_of_vertices-2))];
            mincost=[mincost;intmax];
        else
            cvtomat=cell2mat(r(rem1,rem2));
            result=0;
            if size(visited,2)~=0
                for j=1:size(visited,2)
                    cvtmat=cell2mat(visited(j));
                    if isequal(cvtmat,cvtomat)==1
                        result=1;
                        break;
                    end

                end
            end
            if result==0
                s{i}=cvtomat;
                mincost=[mincost;minval];
                visited{k}=cvtomat;
                k=k+1;
                r{rem1,rem2}=[no_of_vertices+1 no_of_vertices-2];
            else
                s{i} =[randsample(setdiff(1:no_of_vertices, [src,dest]), randi(no_of_vertices-2))];
                mincost=[mincost;getQuality(cell2mat(s(i)),edge_start, edge_connected, edge_weight,src,dest)];
            end
        end


    end
end
```