**Modified Shortest Path Algorithms for Multiple Dynamically Changing Scenarios**

*submitted in partial fulfillment of the requirements for the award of the degree of*

# BACHELOR OF TECHNOLOGY

**in COMPUTER SCIENCE**

**Specialization in**

**CCVT**

By:

|  |  |
| --- | --- |
| **Name** | **Roll No** |
| Shubham Bhatnagar | R110218151 |
| Yashvardhan Singh Nathawat | R110218192 |
| Harsh Upparwal | R110218198 |

***Under the guidance of***

# Mr. Amrendra Tripathi

# Assistant Professor Department of Virtualization



**Department of Virtualization**

**School of Computer Science**

### UNIVERSITY OF PETROLEUM AND ENERGY STUDIES

**Bidholi, Via Prem Nagar, Dehradun, Uttarakhand 2020-21**



**CANDIDATES DECLARATION**

We hereby certify that the project work entitled **Modified Shortest Path Algorithms for Multiple Dynamically Changing Scenarios** in partial fulfilment of the requirements for the award of the Degree of Bachelor of Technology in Computer Science And Engineering with Specialization in Cloud Computing and Virtualization Technology and submitted to the Department of Virtualization at School of Computer Science, University of Petroleum And Energy Studies, Dehradun, is an authentic record of our work carried out during a period from **August, 2020** to **December, 2020** under the supervision of **Mr. Amrendra Tripathi, Assistant Professor and Affiliation**.

The matter presented in this project has not been submitted by us for the award of any other degree of this or any other University.

### Shubham Bhatnagar, Roll No. R110218151

### Yashvardhan Singh Nathawat, Roll No. R110218192

### Harsh Upparwal, Roll No. R110218198

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

(Date: 07 December 2020) **Mr. Amrendra Tripathi**

Project Guide

### Dr. Deepshikha Bhargava

Head Department of Virtualization

School of Computer Science

University of Petroleum and Energy Studies Dehradun - 248 001 (Uttarakhand)

**ACKNOWLEDGEMENT**

We wish to express our deep gratitude to our guide **Mr. Amrendra Tripathi**, for all advice, encouragement and constant support he has given us throughout our project work. This work would not have been possible without his support and valuable suggestions.

### We sincerely thank to our Head of the Department, Dr. Deepshikha Bhargava, for her great support in doing our Modified Shortest Path Algorithms for Multiple Dynamically Changing Scenarios at SoCS.

We are also grateful to **Dr. Manish Prateek Professor and Director SoCS** UPES for giving us the necessary facilities to carry out our project work successfully.

We would like to thank all our **friends** for their help and constructive criticism during our project work. Finally, we have no words to express our sincere gratitude to our **parents** who have shown us this world and for every support they have given us.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Name** | **Shubham Bhatnagar** | **Yashvardhan Singh Nathawat** | **Harsh Upparwal** |  |
| **Roll No.** | **R110218151** | **R110218192** | **R110218198** |  |

# ABSTRACT

Finding optimal shortest path in a dynamically changing scenarios by using multiple modified shortest path algorithms is what this paper/project is about. Generally, the regular shortest path problems and algorithms do not consider the time based updation and changes in edge weights as well as number of nodes while calculating shortest paths in a graph, which is actually the situation in a real time scenario, for example like in a Traffic based system for real time and continuous route planning of vehicles. There are few dynamic shortest path algorithms but not many because studying these types of algorithms is theoretically interesting yet computational wise challenging. The modified algorithms will successfully provide us with **shortest path between a source and a destination node while continuously adapting to the dynamically changing conditions of nodes and edge weights**. At the end, by the results of all three algorithms we’ll be able to compare them on the basis of different performance parameters and successfully conclude which one works better. In this paper we work on three of the most widely used algorithms used for shortest path problems **namely A\*, Dijkstra’s and Bellman ford** and execute them to observe how they work in a Dynamic scenario.

***KEYWORDS:***Dijkstra’s, A\*, Bellman-ford, Dynamic Shortest paths, graphs etc.

# TABLE OF CONTENTS

**Contents**

|  |  |  |
| --- | --- | --- |
| **1** | **Introduction** |  |
| **2** | **Literature Review** |  |
| **3** | **Problem Statement** |  |
| **4** | **Objective** |  |
| **5** | **Design Methodology** |  |
| **6** | **Implementation** |  |
|  | 6.1 Pseudocode  6.2 Flowchart |  |
|  | 6.2 Output Screen |  |
|  | 6.3 Result Analysis |  |
| **7** | **Conclusion and Future Scope** |  |
| **A** | **APPENDIX I PROJECT CODE** |  |

**LIST OF TABLES**

**List of Tables**

1 TIME COMPARISION TABLE

# LIST OF FIGURES

**List of Figures**

1 Pert Chart

2 Research Methodology

3 Flow Chart

4 Time Comparison Chart

# Introduction

**DYNAMIC GRAPH PROBLEMS**

Static and Dynamic are the two types of **Graph problems**, in which Dynamic is the one where there is no restriction in terms of updation of different parameters such as edge weight and number of nodes. Several combinatorial optimization problems on graphs may be deﬁned on dynamic graphs, and in most cases this fact signiﬁcantly alters the problem deﬁnition, so that the employed methods are diﬀerent in the static and dynamic case.

**Dynamic shortest path problems** are now one of the most widely studied dynamic graph problems. Which are further classified as Fully and Semi Dynamic the difference being that fully can have both incremental and decremental update operations available whereas the latter having only one of them. **Dynamic single source shortest path problem** is a type of dynamic shortest path problem which gives shortest paths from a source vertex to all the other vertices of a graph in dynamic scenario and is the one we will be solving and through our all three modified algorithms.

**WHY DYNAMIC SCENARIO?**

In usual cases the shortest path problems were implemented to solve the transportation related problems, where transportation cost was the main compensation expendable to run a route between locations. These were all static cases, since few decades back there weren’t any dynamic factors affecting the transport cost, and some established systems are still using the same measurement such as freight forwarders, packet services, etc.

However, when computer games come to a rise, the case are expanded to solve transportation problems in games, which in turn represents today’s real condition on a micro scale. They are more complex as they involve just-in-time decision to change the next route when additional problems come up, such as unexpected monsters, disasters, and so on. Which are not too far away from today’s traffic problems. **Traffic congestion** often came out of nowhere and is not expected before. Gladly the automatic traffic control has the data which shows on what route the congestion occurs and how bad the traffic is, so it is feasible to model the graph and cost using these data in order to find the most possible new route.

**BACKGROUND OF ALGORITHMS**

The calculation of shortest path is a very widely used topic of research inArtificial Intelligence, Operations Research, and Transportation for many years. Dijkstra, A\* and Bellman Ford are most widely used algorithms for shortest path computation. But the existing approaches of these algorithms was mainly devised for static networks and therefore are not effective when applied on dynamic shortest path planning in a real-time Dynamic environment in their original form.

* **DIJKSTRA ALGORITHM**

Single source shortest path algorithm for both directed and undirected graph (not for negative weighted graph). In this algorithm each node is relaxed based on min(d[u]+c[u,v], d[v]) and added to a set of relaxed vertices set where,

d[u]🡪weight of vertex u from source

c[u,v]🡪Weight from vertex u to v

d[v]🡪weight of vertex from source to vertex v.

This always gives optimal solution because it relaxes all the vertices and this is also the reason this algorithm is slow because it calculates and modify adjacency list for all vertices whether or not that path is relevant or not.





Its worst-case time complexity is O(v^2) and O (elog v) if adjacency list is used.

* **A\* ALGORITHM**

It is a path finding algorithm which is similar to Dijkstra, comes under heuristic/informed search.

To select a vertex to expand, we calculate

F(n)=G(n)+H(n)

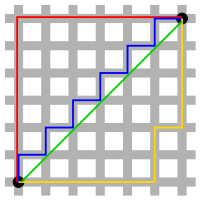
G(n): Actual cost from start node to ‘n’.

H(n): Estimation cost from ‘n’ to goal node.

It is better than Dijkstra in terms of time complexity O(|E|) but its accuracy is dependent on how we calculate the heuristic.

Different ways to calculate heuristics:

1. Euclidian distance: Simple straight-line distance between current node and destination node.
2. Manhattan distance: distance between two points is equal to lines along right angle



Green line: Euclidian distance

Blue line: Manhattan distance

* **BELLMAN FORD ALGORITHM**

The Bellman-Ford algorithm is a classical algorithm for the single-source shortest paths problems.

Bellman-Ford works fine for graphs that contains negative edge weights unlike Dijkstra, hence it is more versatile. Bellman-Ford is also simpler than Dijkstra but the time complexity of Bellman-Ford is O(v.e), which is more than Dijkstra. Although, it contributes in handling two main issues:

1. Detection of negative cycles

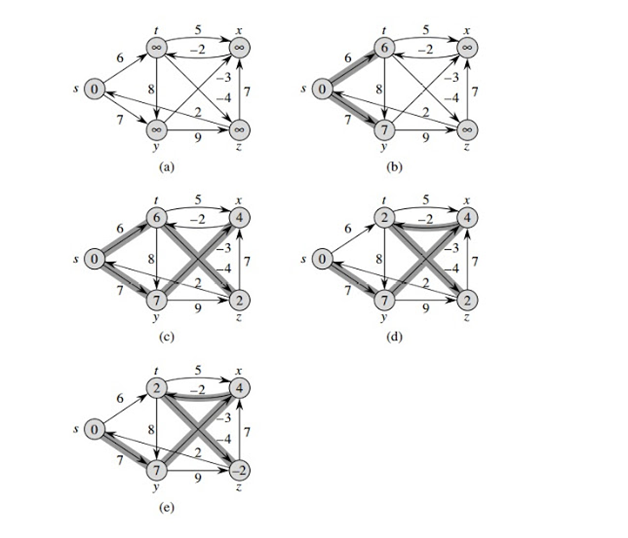
this is very important because it hinders shortest-path finding altogether because going through a negative weight cycle will reduce the path length and we will get an incorrect result.

1. Proper ordering of relaxations

Dijkstra’s algorithm selects the nearest vertex that has not been visited yet. Bellman-Ford on the other hand, goes through all the edges in every iteration which increases the accuracy of the distance to any given vertex.

**dk[u] = min(dk-1[u], min(dk-1[i] + cost[i,u])) i ≠ u**

Above recurrence relation is used in reaching an optimum solution.



Time complexity:

Worst case: O (V\*E), Best case: O (E)

Space complexity: O (V)

# Literature Review

Reference paper Number [1]:

In this paper the main focus is on finding shortest path in real time depending upon the current location of the user and current weight of the graph, it has focused on the greedy approach by applying algorithm to whole graph over and over again. **Loop Optimization Method (LOM**) has been developed to produce a sub graph to reduce excessive re planning operations. It has proved to be useful for a large graph.

Reference Paper Number [2]:

In this paper they are dynamizing the Dijkstra’s algorithm by using retroactive priority queue data structure. **Retroactive data structure** identifies the set of affected vertices step by step and thus help to accommodate changes in least number of steps.

**Scope:** There is a scope of processing the update operations as a single one which can further reduce the time bound of the problem

Reference Paper Number [3]:

This study applied the case of dynamic path using Dijkstra’s algorithm on a graph representing street routes with two possible digraphs: one-way and two- way. experiments showed that there are no additional computation stresses in re-calculating the shortest path while going halfway to reach the goal.

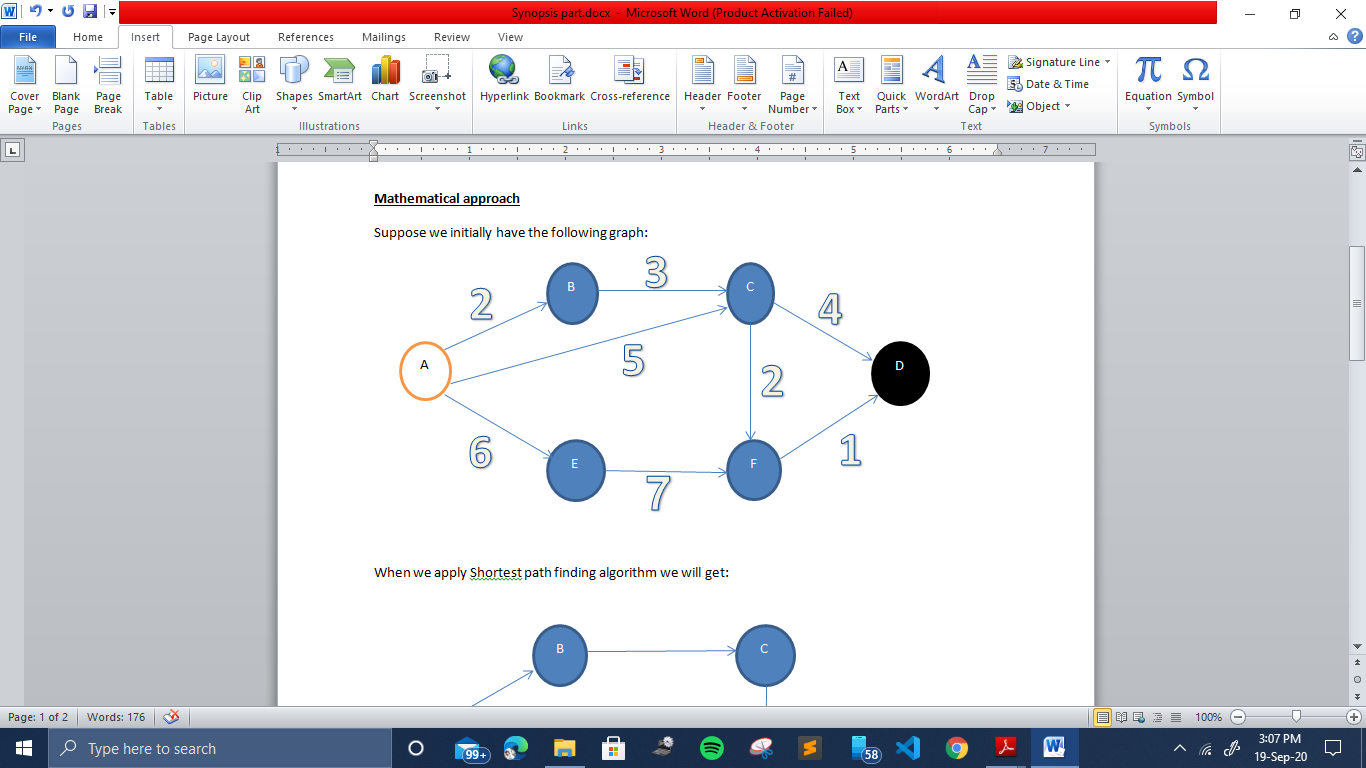
**Scope:** It is not feasible to use in a two-way digraph because a large “loop” may occur after the recalculation. We can eliminate this by programming that parallel edges between two nodes are not to be selected as sequence in path. Also, its effect on wider graph i.e. more than 30 nodes are unknown.

# Problem Statement

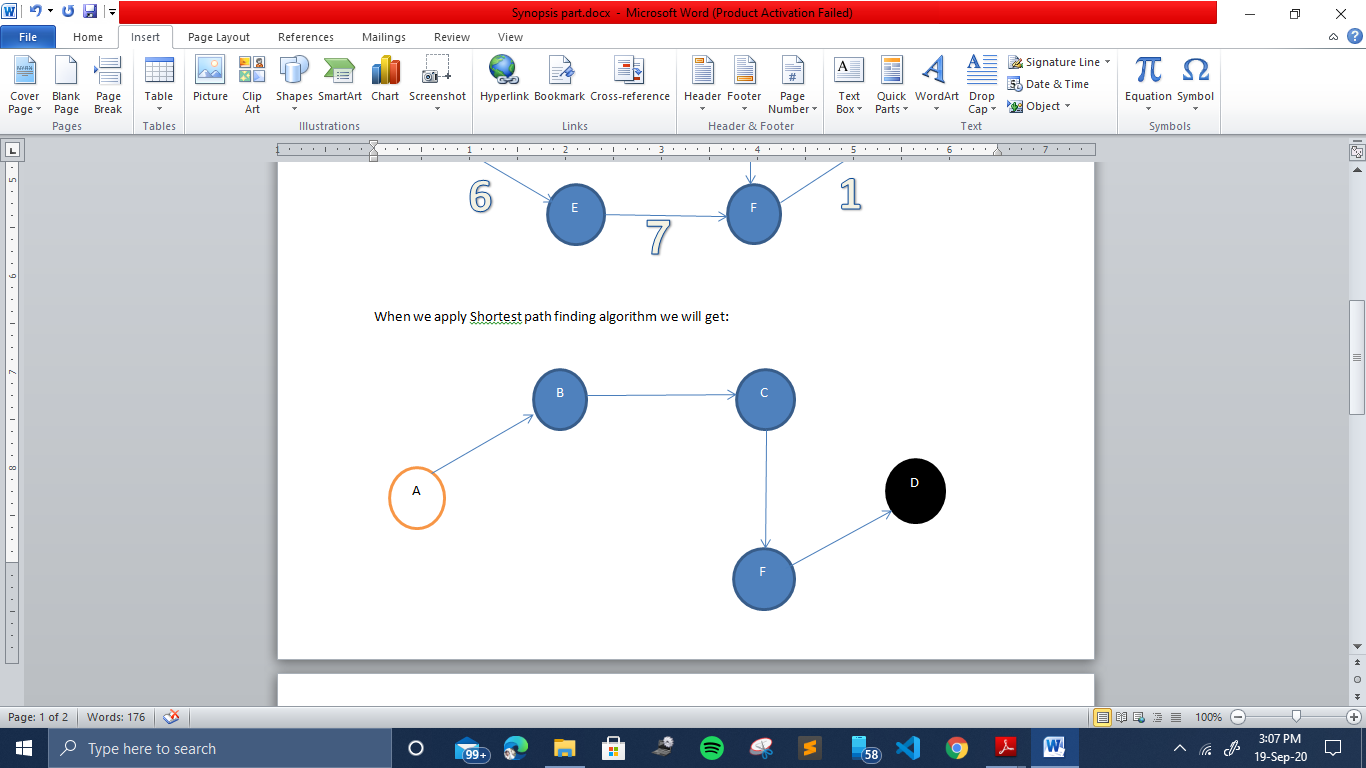
* **MATHEMATICAL PROBLEM STATEMENT**

A challenging issue is to develop efficient and less computationally intensive algorithms that can be used to compute dynamic shortest paths between moving objects and given destinations in dynamic changing networks.

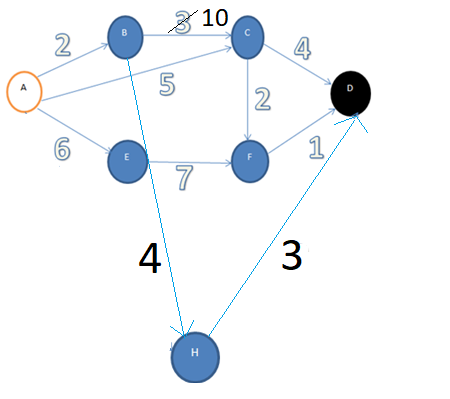
1. Suppose we initially have the following graph (Entered by user or chosen from a set of different graphs)



1. After applying the shortest path algorithms on this initial graph, we get



1. Now if dynamically another node is added or even the weight of any edge is changed during the traversal of the path then the shortest path will also get altered. For example if a new node H is added as shown below and the weight of edge from B to C is changed from 3 to 10 then the shortest path will change from B which we need our algorithms to find out



* **REAL WORLD PROBLEM STATEMENT**

Essentially any combinatorial optimization problem can be formulated as shortest path problem and this is the reason why shortest path algorithms are used so widely and can be applied to solve various problem statements.

Route planning in traffic-based Road Networks, Internet Routing, Communication Networks, Operation Research, Management Systems, Computer Science and Artificial Intelligence: The Dynamic shortest path algorithms are implemented to solve most of the problems associated with these fields in real world scenarios.

The scenario of **Dynamic real-time transportation networks** provides the most appropriate problem statement for the Dynamic Shortest path algorithms: -

If we talk about a Dynamic Traffic environment then there the traffic conditions are mainly time dependent. Like if we take a scenario of a widely taken route in our city Dehradun i.e. from ISBT to UPES, which is a common route for lot of people every day and thus has a lot of congestion in traffic. Now at the start of journey we can plan a shortest path according to the initial traffic and other conditions, but we may have to adjust that route en-route because traffic conditions change all the time. Moreover, in some cases we have to modify our route regularly after reaching some particular in between locations i.e. multiple times we might need to change the route according to real time traffic environment. we need an algorithm that can take into account both the changing locations of the travelers and changing traffic conditions in the area. To the best of our knowledge, little has been reported on the computation of this type of dynamic shortest paths. Currently, most existing dynamic algorithms only consider changes in traffic conditions and compute a dynamic shortest path between a pair of or all pairs of fixed nodes on a network, or consider changes in the position of the start node but re-compute the shortest path from scratch when traffic condition changes.

The current problems faced by traffic information providers is that they have to offer the drivers with GPS enabled terminals a source-destination path subject to the following constraints**: (a) the path should be fast in terms of travelling time; (b) the travelling times (weights on the edges) vary according to traﬃc information being available on part of the road network; (d) traﬃc information data are updated at regular time intervals; (e) answers to path queries should be computed in real time**. Also, it may take a long time for some mobile terminals, such as in-car computers and personal digital assistants, to re-plan a new route from scratch using a static approach while en-route because of the limited computational power of these devices.

**Our project develops efficient and less computationally intensive algorithms that can be used to compute dynamic shortest paths between moving objects and given destinations in dynamic changing networks. Which gives the quickest path in terms of travelling time even if the weights are dynamically changing. Also, in our algorithms there is no need to calculate the entire path from scratch.**

# Objective

**Objective 1**

Studying and grasping the background knowledge related to the Algorithms: Graph Theory, understanding heuristic approach, Previous works related to dynamic shortest path problem.

**Objective 2**

Static implementation of Shortest path algorithms (A\* algorithm, Dijkstra, Bellman-Ford)

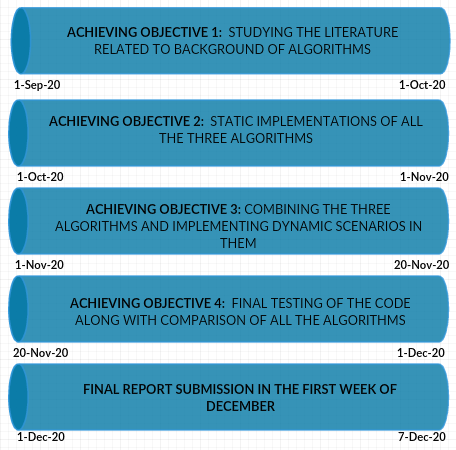
**Objective 3**

Implementation of the Dynamic Shortest path algorithms.

**Objective 4**

Comparison of Algorithms on basis of different performance parameters

1. **PERT CHART**



# Design Methodology

**Research and analysis**

We will gain information about graph theory, Shortest path finding algorithms (A\*, Dijkstra, Bellman Ford), different data structures to search, retreive data and how we should input data to make our graph dymnamic and that how size of graph affects the speed of algorithms.

**Design**

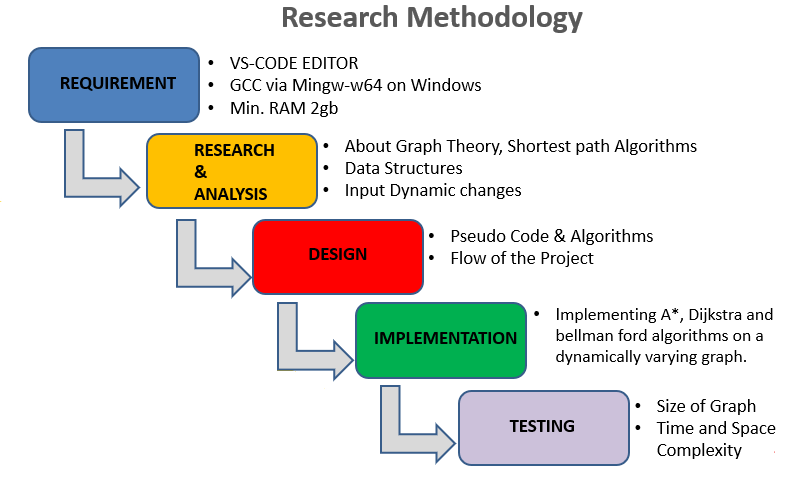
Here we will create pseudocode and understand the data flow of the code we will implement.

**Implementation**

In this phase we will implement A\*, Dijkstra and bellman ford algorithms on a dynamically varying graph.

**Testing:**

Test the algorithms against different sizes of graphs and different parameters and finally compare their complexities.



* **SOFTWARE REQUIREMENTS**
  + OPERATING SYSTEM: Windows 7 or higher / Linux Based / Mac OS
  + EDITOR : VS-CODE Editor
  + LANGUAGE : C language
  + COMPILER : GCC via Mingw-w64 on Windows
* **HARDWARE REQUIREMENTS**
  + PROCESSOR : Intel® Core™ i5 Processor or higher
  + DISK DRIVE : Hard Disk Drive
* RAM : 2Gb or Higher

# Implementation

# Pseudocode

This Program implements the Dijkstra, Bellman ford & A\* Algorithms in a Dynamic Scenario where the weights of the edges and no. of nodes in the graph are continuously changing.

***THE int main FUNCTION***

***{***

*Allocate Memory to the graph Dynamically by using Calloc*

*Enter the initial Graph*

*NOTE: We have not mentioned the graph functions in this*

*Pseudo code*

*Enter the Initial Source and the constant Destination Vertex*

*Choose 1 Algorithm to proceed with among the 3 Algorithms*

1. *dijkstra(Source)*
2. *Bellmanford(Source)*
3. *Astar(Source)*

*return 0*

***}***

***THE FUNCTIONS FOR THE 3 PATH ALGORITHMS***

*function* ***Dijkstra****(Argument one)*

***{***

***//****initializing-single-source(G,S)*

*for(each vertex)*

*d[v] 🡨 Infinity;*

*π[v] 🡨 NULL;*

*d[source] 🡨 0;*

***//****initializing set list and queue*

*S 🡨 Φ;*

*Queue 🡨 All vertices;*

***//****Extracting minimum distance vertex and relaxing them*

*clock\_gettime (CLOCK\_MONOTONIC, &start)*

*while(Queue is not empty)*

*U 🡨 minimum(weights, number of vertices);*

*S 🡨 S U {U};*

***//****find number of vertices adjacent to u*

*for(each adjacent vertex)*

***//****Relaxing the nodes and updating their weights*

*Relax(U, adjacent vertex, Weight);*

*if[ d[adjacent] > d[U] + weight ];*

*d[adjacent]🡨d[U] + weight;*

*π[adjacent] 🡨 U*

*clock\_gettime (CLOCK\_MONOTONIC, &end);*

*//Calculate time in nano Seconds*

*time\_taken = (end.tv\_sec - start.tv\_sec) \* 1e9;*

*time\_taken = (time\_taken + (end.tv\_nsec - start.tv\_nsec));*

*Display path Calculated by dijkstra algorithm*

*path();*

*Display time taken by dijkstra algorithm*

*Display path by other two algorithms as well*

*Ask user to enter if he/she wants to start traversing the path (0/1)*

*if yes, he wants to traverse*

*traverse ();*

*if no he doesn’t want to*

*exit (0);*

***}***

*function* ***Bellmanford****(Argument one)*

***{***

***//*** *OVERESTIMATION, initialize a null parent and distance of every vertex to Infinity except for source, we assign source distance to zero*

*for each vertex v in vertices  
 distance[v] = INFINITY  
 parent[v] = NULL*

*distance[source] = 0*

*// RELAXATION, relax edges for (V-1) times here V is No. of vertices*

*clock\_gettime (CLOCK\_MONOTONIC, &start)*

*for i = 1 to V-1       
        for each edge (u, v) with weight w  
             if (distance[u] + w) is less than distance[v]  
                distance[v] = distance[u] + w  
                parent[v] = u****//*** *check for negative-weight cycles*

*for each edge (u, v) with weight w  
    if (distance[u] + w) is less than distance[v]  
        Return (“Graph contains a negative-weight cycle”)*

*clock\_gettime (CLOCK\_MONOTONIC, &end);*

*//Calculate time in nano Seconds*

*time\_taken = (end.tv\_sec - start.tv\_sec) \* 1e9;*

*time\_taken = (time\_taken + (end.tv\_nsec - start.tv\_nsec));*

*Display path Calculated by Bellmanford algorithm*

*path()*

*Display time taken by Bellmanford algorithm*

*Display path by other two algorithms as well*

*Ask user to enter if he/she wants to start traversing the path (0/1)*

*if yes, he wants to traverse*

*traverse ();*

*if no he doesn’t want to*

*exit (0);*

***}***

*function* ***Astar****(Argument1)****{***

***/\*****adding source to final list****\*/***

*final= (struct nodeForAStar \*)malloc(1\*sizeof(struct nodeForAStar));*

*formoving=final;*

***/\*****initializing PQ(priority queue array****) \*/***

*PQ= (struct nodeForAStar \*) malloc (1\*sizeof(struct nodeForAStar));*

***/\*****calling function to create PQ using heap and passing source node from graph, created min heap using f(H)=D+H \*/*

*createArrayForPQ(graph[0],graph[0].noOfNodeConnected);*

*/\* now initial PQ is ready, now we need to add more nodes to PQ and extract elements until destination node is reached\*/*

*clock\_gettime (CLOCK\_MONOTONIC, &start)*

*while (!(PQ[0].vertex==destination))*

*{*

*deleteFromHeap();*

*createArrayForPQ(graph[(formoving->vertex)-1],graph[(formoving->vertex)-1].noOfNodeConnected);*

*}*

*clock\_gettime (CLOCK\_MONOTONIC, &end);*

*//Calculate time in nano Seconds*

*time\_taken = (end.tv\_sec - start.tv\_sec) \* 1e9;*

*time\_taken = (time\_taken + (end.tv\_nsec - start.tv\_nsec));*

*Display path Calculated by Astar algorithm*

*Display time taken by Astar algorithm*

*Display path by other two algorithms as well*

*Ask user to enter if he/she wants to start traversing the path (0/1)*

*if yes, he wants to traverse*

*traverse ();*

*if no he doesn’t want to*

*exit (0);*

***}***

*Function* ***path****(Argument 1, Argument 2, Argument 3)****{***

*Specify counter as 0*

*if(destination equals source)*

*print value of destination*

*return 0;*

*if(destination not equals source)*

*print value of destination*

*Recursively call the path function again by changing the destination*

*return(path(parent[destination-1],parent,source));*

*}*

***THE FUNCTIONS TO DYNAMIZE AND TRAVERSE THE 3 ALGORITHMS AND PATH RESPECTIVELY***

*Function* ***traverse****(Argument 1, Argument 2)****{***

*Initialize a temp[] array size of no. of vertices with value 0*

*Insert the path calculated by algorithms in the temp[] array*

*while(R2 which if turn 0 we have to call dynamic function is equal to 1){*

*if(counter variable is not at the last node){*

*Display the node you are currently at i.e temp[counter]*

*Display the next node in path i.e temp[counter+1] and ask if user want to keep*

*going or bring dynamic change*

*change R2 to 0 if user wants dynamic change*

*if(R2 is still 1)*

*counter++*

*}*

*else if(counter variable is at last node){*

*Display user is at the last node i.e temp[counter]*

*exit (0)*

*}*

*}*

*if(R2 value is 0)*

*{*

*Call Dynamicchange() function*

*Call the static Algorithm by passing the node user is currently at in place of source*

*if(Value of Algorithm is 1)*

*dijkstra(temp[counter])*

*if(Value of Algorithm is 2)*

*Bellmanford(temp[counter])*

*if(Value of Algorithm is 3)*

*Astar(temp[counter])*

*}*

***}***

*Function* ***Dynamicchange****()****{***

*Display the two options of Dynamic change that a user can pick from*

1. *Add Nodes*
2. *Change weight of Edges*

*if(the value of R3 is 0)*

*addnodes() function is called*

*if(the value of R3 is 1)*

*weightupdate() function is called*

***}***

*Function* ***Addnodes****()****{***

*Ask User how many nodes they want to add*

*Update the total number of vertices in graph*

*noVer=noVer+n*

*Use realloc to assign the memory to new nodes in the graph*

*Call createEdge() function to add the respective node details*

*Createedge(Argument 1)*

*Call getheuristic() function to add the heuristic for new nodes*

*get\_heuristic(Argument 1)*

*Ask User about the connections of the new node with existing Nodes*

*Update those Node details in the graph*

*updateedge()*

***}***

*Function* ***Weightupdate****()*

***{***

*Ask user how many edges has to be updated*

*for(play the loop the number of times equal to edges to be updated)*

*{*

*Ask user to enter Source vertex of edge*

*Ask user to enter the destination vertex of edge*

*Ask user to enter the updated weight of this edge*

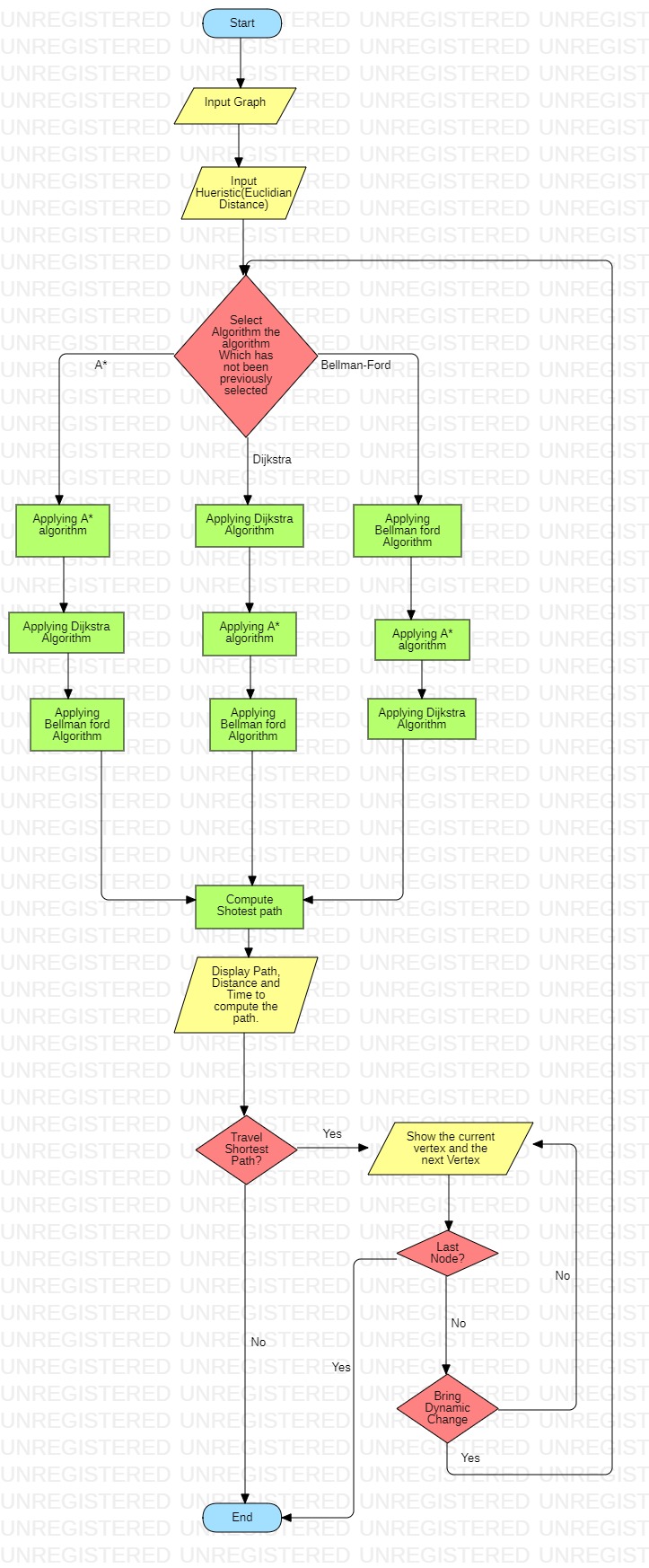
*Call changeweight() function*

*changeweight(Argument 1, Argument 2, Argument 3)*

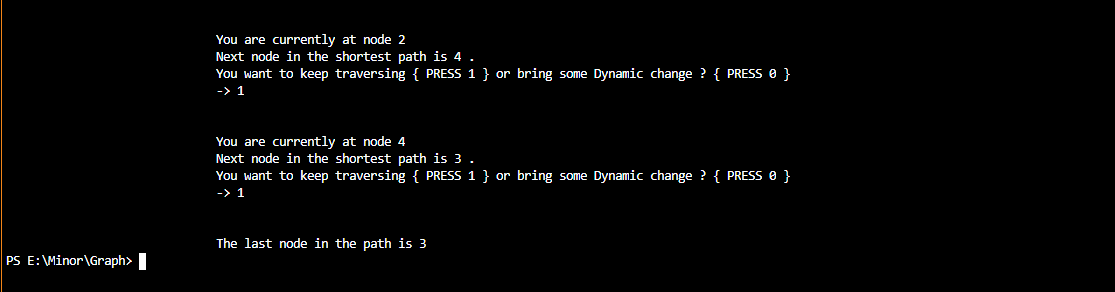
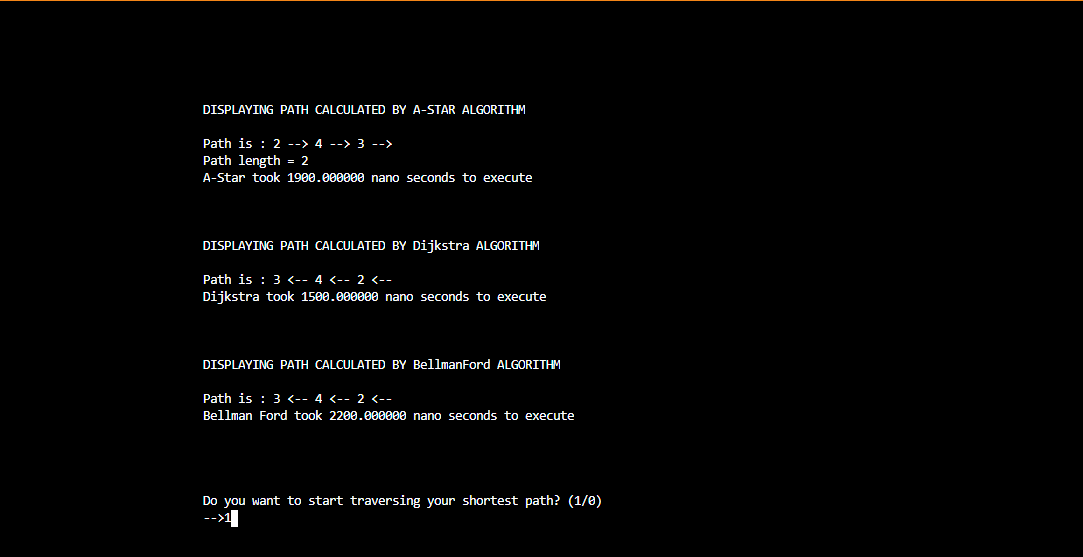
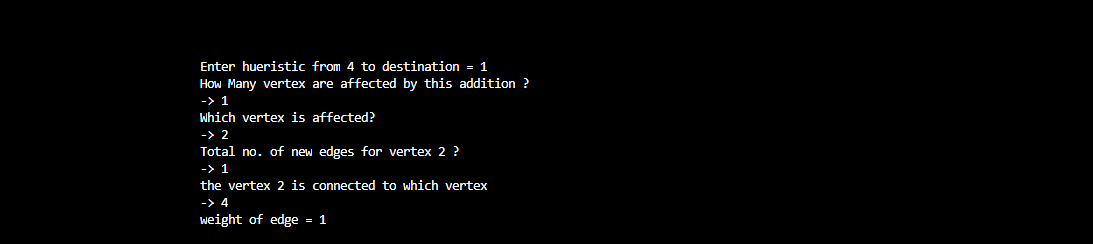
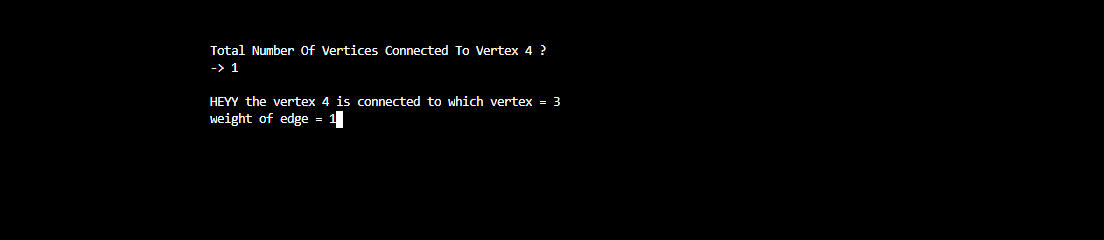
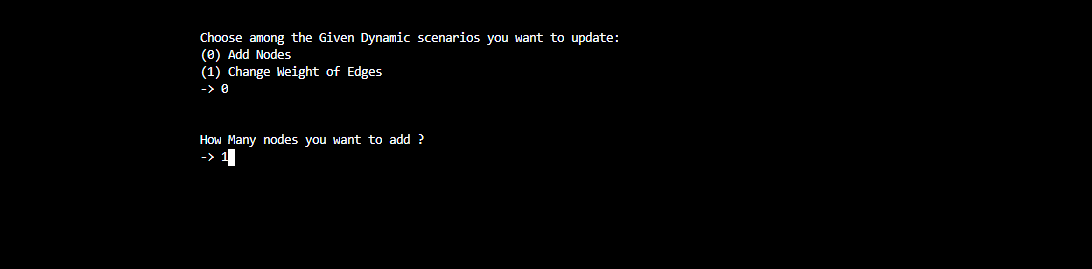
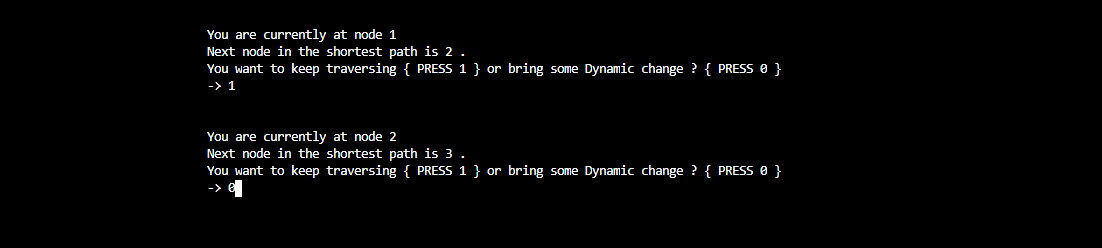
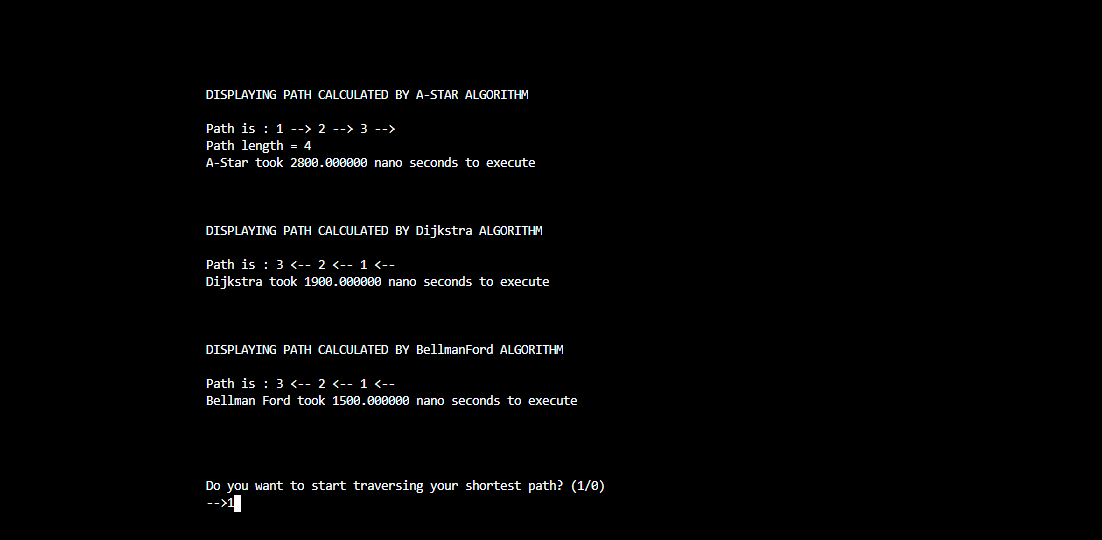
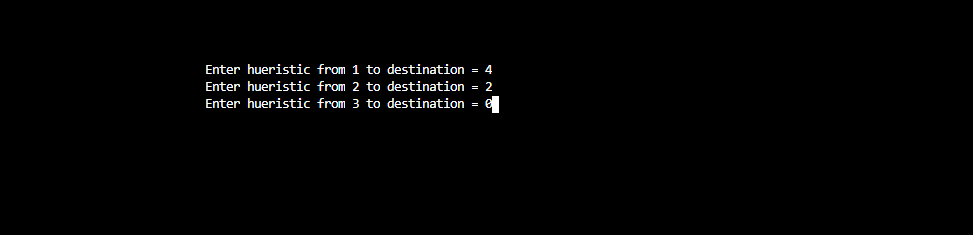
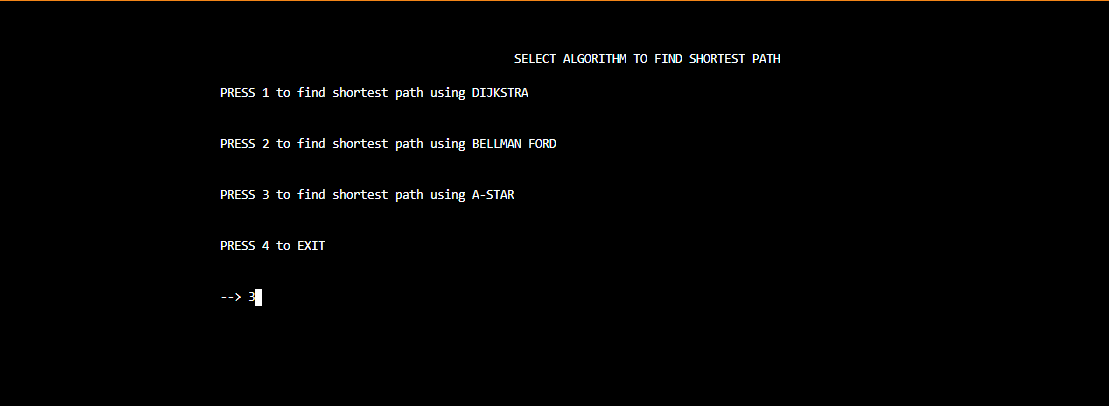
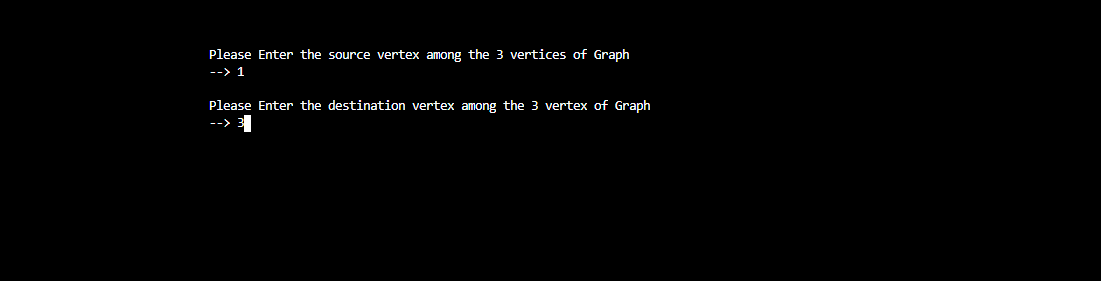
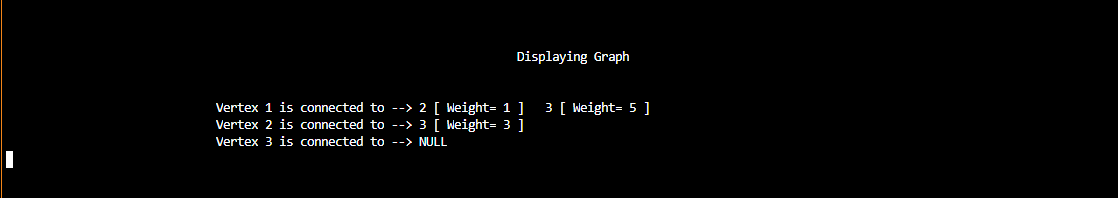
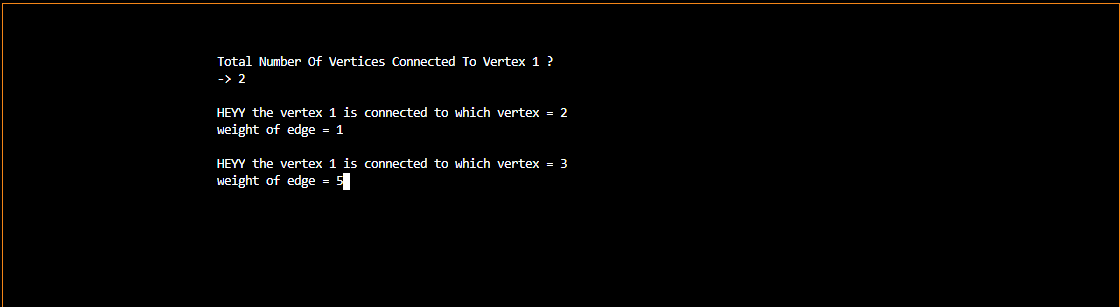
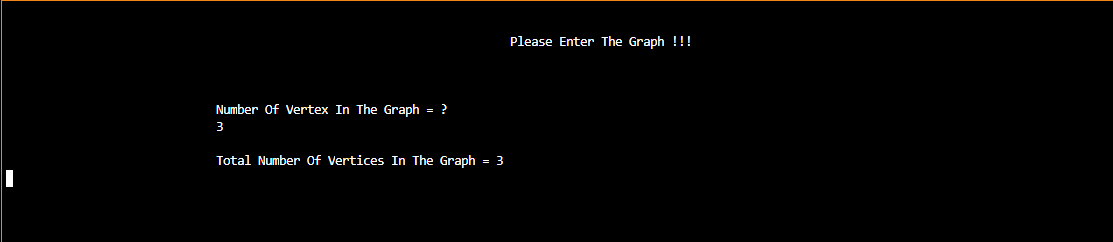
*}*

*}*

**FLOW CHART:**



1. **OUTPUT SCREEN**

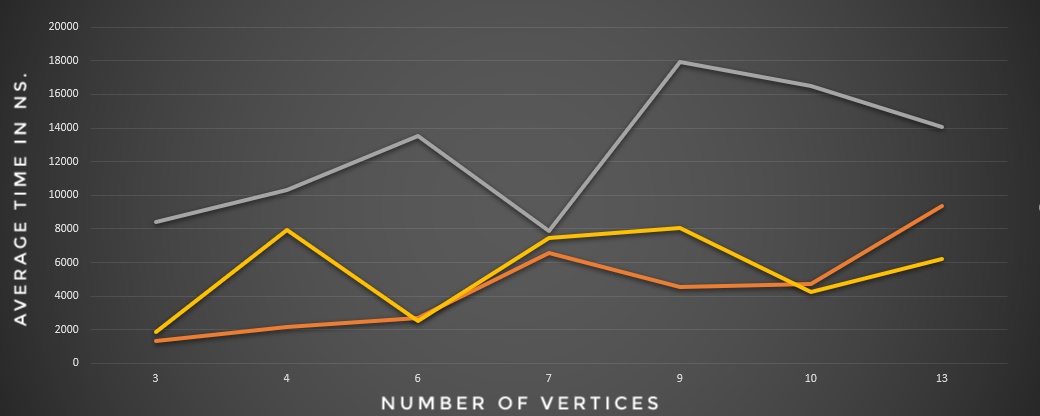
****

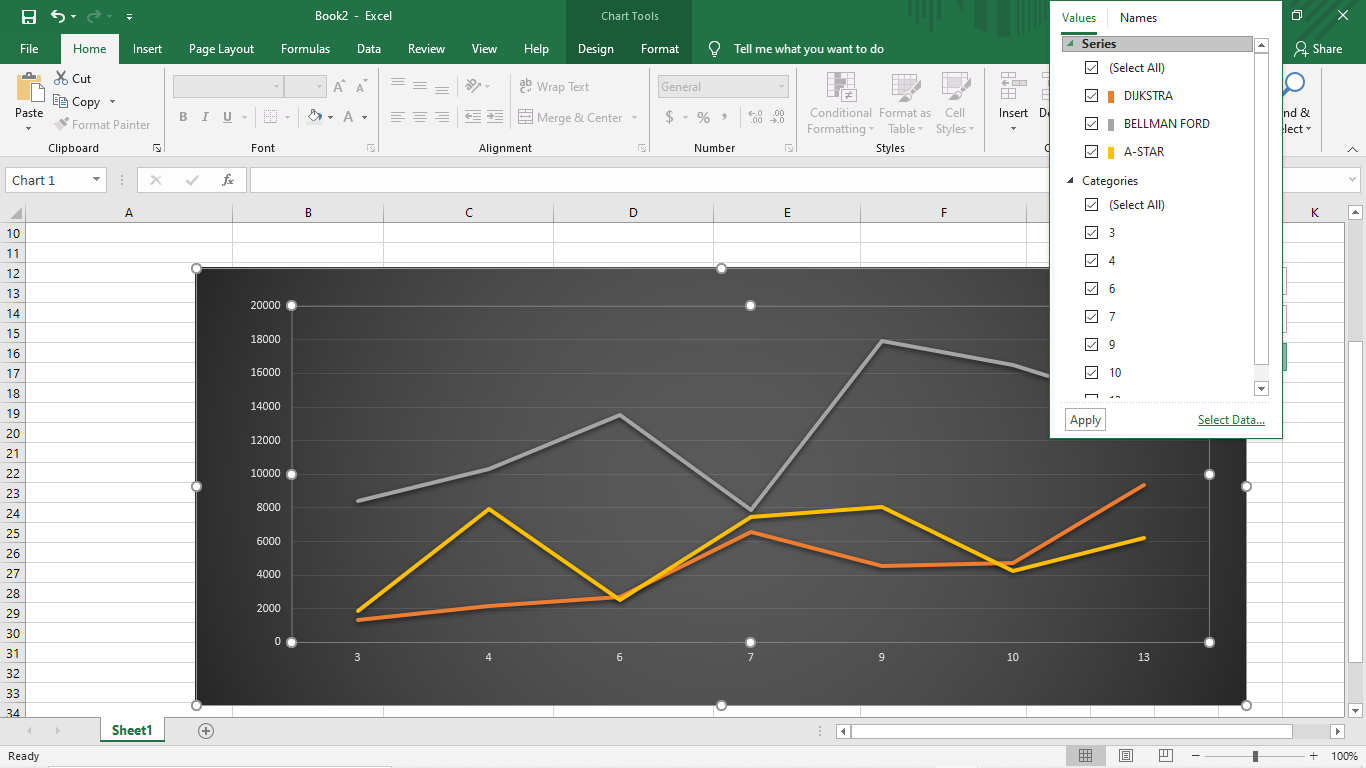
1. **RESULT ANALYSIS**

We have calculated the result of our algorithm by running 10 different graphs in our program. We have run each graph thrice and then taken the average of time encountered in all three attempts.

NOTE: Due to reason of Context Switching in our PC it is not possible to get a constant time in every Run case. Due to which we have taken the average of multiple test cases

Final plotting of the Result table is done on a line graph given in the next page.

****



**TIME COMPARISION TABLE**

|  |  |  |  |
| --- | --- | --- | --- |
| **Number of Vertex** | **Dijkstra (ns)** | **Bellman Ford (ns)** | **A-Star (ns)** |
| |  | | --- | | 3 | | 4 | | 6 | | 7 | | 9 | | 10 | | 13 | | |  | | --- | | 1333.33 | | 2166.67 | | 2700.00 | | 6566.67 | | 4566.67 | | 4700.00 | | 9366.67 | | |  | | --- | | 8400.00 | | 10300.00 | | 13533.33 | | 7866.67 | | 17900.00 | | 16500.00 | | 14033.33 | | |  | | --- | | 1900.00 | | 7966. 667 | | 2533.33 | | 7433.33 | | 8066.67 | | 4233.33 | | 6233.33 | |

* We can clearly see that in the cases of less number of vertices the Dijkstra algorithm is working better then A\* but when the no. of vertices increases the A\* algorithm takes lesser time due to its special ability to not travel all the vertices of a graph

# Conclusion and Future Scope

All the four objectives of the project have been achieved before the due date. All The 3 shortest path algorithms have been successfully modified for dynamically Changing scenarios and compared on the basis of time taken.

Future Scopes:

* We can improve the A\* by not asking for heuristic from user itself instead using other methods like longitude and latitude of places to calculate heuristic
* The UI and the front-end part of the project has a lot of scope for updation
* Line of code can be reduced by making the code compact.
* We can change some specific variables to reduce the Time complexity

# References

[1] Continuous Route Planning over a Dynamic Graph in Real-Time: Tianlun Daia, Wenchao Zhenga, Jiayue Suna, Cun Jib, Tao Zhoua, Mingtong Lia, Wei Hua, Ziqiang Yua

LINK: https://www.sciencedirect.com/science/article/pii/S1877050920315799

* [2] Dynamizing Dijkstra: A Solution to Dynamic Shortest Path Problem through Retroactive Priority Queue: Sunita, Deepak Garg

LINK : <https://www.sciencedirect.com/science/article/pii/S1319157817303828>

* [3] SHORTEST PATH WITH DYNAMIC WEIGHT IMPLEMENTATION USING DIJKSTRA’S ALGORITHM:

Elizabeth Nurmiyati Tamatjita1; Aditya Wikan Mahastama2

LINK:<https://www.researchgate.net/publication/320201529_Shortest_Path_with_Dynamic_Weight_Implementation_using_Dijkstra's_Algorithm>

* [4] A Novel Hybrid Algorithm for the Dynamic Shortest Path Problem: Yafei Guo, Zheng Qin,Yang Chang

LINK: <https://ieeexplore.ieee.org/document/5583241>

* [5] A shortest path algorithm with novel heuristics for dynamic transportation networks:

B. HUANG, Q. WU and F. B. ZHA

LINK:<https://www.researchgate.net/publication/262407565_A_shortest_path_algorithm_with_novel_heuristics_for_dynamic_transportation_networks>

# A APPENDIX I PROJECT CODE

# <https://github.com/shubhambhatnagar12/minor1/blob/master/Final-Minor-Code.c>