**JAGAN INSTITUTE OF MANAGEMENT STUDIES**   
3, institutional area, sector-5, Rohini, Delhi-110085   
(AFFILIATED TO GGSIPU-NEW DELHI) 

**Object Oriented Programming with C++**  
**(BCA 271)**

Assignment-1

Shortest Distance Using Dijkstra Algorithm

**Submitted by:**

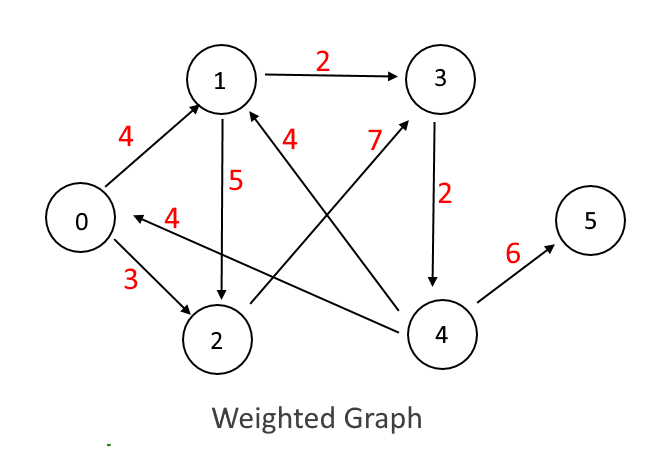
Pragya                                                          Himanshi Tomer   
Enrolment no.- 02014002023                           Enrolment no.- 05214002023  
BCA, Shift-1, Semester-3 BCA, Shift-1, Semester-3

# Table of Contents

|  |  |  |
| --- | --- | --- |
|  | **Content** | **Page No.** |
| 1. | Introduction | 2 |
| 2. | Case Description | 3 |
| 3. | Contributions of Team Members | 5 |
| 4. | Objectives | 6 |
| 5. | Design | 7 |
| 6. | Algorithm | 8 |
| 7. | Data Description | 13 |
| 8. | File Descriptions | 15 |
| 9. | Screenshots | 17 |
| 10. | Applications | 23 |
| 11. | Conclusion | 25 |
| 12. | References Used | 26 |

# Introduction

Graphs are discrete data structures consisting of vertices and edges that connect these vertices.

They are a powerful and widely used data structure. While we may not realize it, the implementations of this hierarchical data structure are rooted in every other common day-to-day technology. Ranging from finding the most convenient route on *Google Maps* or your car’s in-built navigation system to communicating over a network, graphs play a crucial role in various applications.

Graphs are expressed as a set of vertices or points connected by edges, which represent the relationships or paths between the points.

Finding the shortest path between these vertices is a common problem encountered in many real-world applications. This challenge spans multiple fields and industries, highlighting the versatility and importance of graph algorithms.

In the context of this assignment, we will focus on implementing a solution for the shortest path problem using an undirected graph. In order to achieve this, we will define a Point class to represent vertices and an Arc class to represent the connections or edges between these vertices.

The assignment involves using Dijkstra’s Algorithm, a well-known and efficient method for finding the shortest path in graphs, to compute the best path between any two vertices.

Additionally, this project explores the practical significance of the graph algorithm.

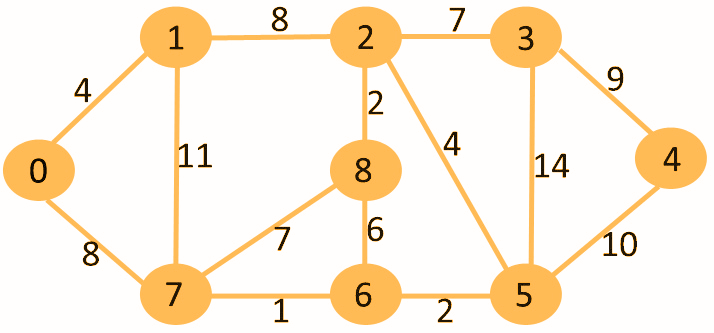
The ability to compute shortest paths efficiently can lead to improvements in various fields, such as transportation, networking, and logistics. By modelling the graph and implementing the algorithm, we gain valuable insights into the fundamental principles of graph theory and their practical utility.

# Case Description

**Problem Description:**

Define Point class and an Arc class. Define a graph class which represents graph as a collection of Point Objects and collection of Arc objects.   
Find the shortest distance between any two points (use Dijkstra’s algorithm).   
You may also design your own algorithm.

**Proposed Solution:**

Consider the graph in the figure.

The given is an undirected graph, with each node going to itself (source and destination), and edges between various nodes, such that a path is formed.

The circles with numbers enclosed represents nodes, the lines between them, connecting them to each other- edges and the numbers above the edges (lines), are known as cost or weight. Now, let’s try to travel from the node 0 to the other nodes.

The most optimal solution to find the shortest path from any one of these nodes (say, source), to another node (say, destination) can be found out using Dijkstra’s Algorithm.

Let,   
 Source Vertex = 0  
 Using, Dijkstra’s algorithm:

d (u) + c (u, v) < d (v)

where, d (u) is the distance from source (assume)  
 d (u, v) is the distance between *u* (source) and *v* (destination)  
 d (v) is the distance to the source  
  
Implementing the above formula repeatedly until we get the path with least weight possible:

|  |  |  |
| --- | --- | --- |
| **Vertex** | **Path** | **Weight** |
| 0 | 0 | 0 |
| 1 | 1 | 4 |
| 2 | 1->2->2 | 12 |
| 3 | 0->1->2->3 | 19 |
| 4 | 0->7->6->5->4 | 21 |
| 5 | 0 -> 7 -> 6 -> 5 | 11 |
| 6 | 0 -> 7 -> 6 | 9 |
| 7 | 0 -> 7 | 8 |
| 8 | 0 -> 1 -> 2 -> 8 | 14 |

The formula given afore, is known as *Relaxation.*It is known as the heart of the Dijkstra algorithm.

The code for the relaxation is as follows:

*if ((beg==min\_node) && (visit\_status[last]=='N') && (cost[beg]+weight<cost[last])) {*

*cost[last] = cost[beg] + weight;*

*}*

*if ((last==min\_node) && (visit\_status[beg] == 'N') && ((cost[last]+weight)<cost[beg])) {*

*cost[beg] = cost[last] + weight;*

*}*

It is a concept of graph theory that represents the process of updating the shortest path known between any two vertices (points) with an even shorter path by checking untraversed paths of the graph.

It is also used in other algorithms of shortest distance, like Bellman-Ford.

In the provided solution, Dijkstra Algorithm works by progressively exploring the shortest paths to all the vertices from a given starting point, ensuring that each vertex is visited only once.

The following are the steps involved in the successfully implementation:

1. Initialization of source, or initial point to infinity (we treat the weight of arc as unknown; this is usually done by a number representing infinity) or a null pointer.
2. Initialization of visit status of each node as visited or not. This is so as to not repeat any path.
3. Finding unvisited nodes by checking status.
4. For each un-visited node, the distance is calculated, and if it is found to be shorter than the last known distance, or the variable is infinity or a null ptr, the value is initialized.
5. The node once checked, is marked as visited.
6. The above is repeated until no un-visited nodes exist.

The given is a simple, Adjacency Matrix implementation of the algorithm.

# Contributions of Team Members

The crux of the work was accomplished together working simultaneously.

Tasks that were performed together:

* Selection of the most suitable implementation of Dijkstra Algorithm.
* Theoretical study of Dijkstra Algorithm- applications, optimizations, etc.
* Terminologies related to and the various types of graphs.

The following is the broader division of work performed by the team member:

Pragya  
(02014002023)

* + Coding:
    1. Implementation of Point and Arc Classes
    2. Implementation of basic structure of Graph Class   
       (dynamic alloc, pointers, etc)
    3. Alphabet-Index Conversion and Operations
    4. Constructors and Destructors
  + Documentation:
    1. Objectives
    2. Design
    3. File Description
    4. Algorithms
  + Data-Sets:
    1. Ran tests with various data sets.

Himanshi Tomer  
(05214002023)

* + Coding:
    1. Shortest-Path implementation
    2. Integrating algorithm with Classes
    3. Implemented File Handling with *.txt* file
  + Documentation:
    1. Introduction
    2. Case Description
    3. Applications
    4. Conclusion
  + Data-Sets:
    1. Provided with samples in .txt

# Objectives

The objective of this assignment is to:

1. Represent Graphs using Classes and Objects:   
   The Point and Arc classes define the fundamental elements of a graph.   
   The Graph class provides a container for these elements, allowing you to create and manipulate graphs.
2. Calculate Shortest Distance:   
   To find the shortest possible path between two given points by implementing Dijkstra’s algorithm to efficiently compute the shortest distance between any two points in the graph. Dijkstra’s algorithm is a valuable tool for solving various graph-related problems, such as finding optimal routes in transportation networks or network analysis.
3. Implementing Dijkstra’s Algorithm: Dijkstra’s algorithm is a powerful tool for finding the shortest path between nodes in a graph where edges have non-negative weights.   
   *GPS (Global Positioning System)* and *Google Maps* also make use of this algorithm to provide users with the best possible routing options.
4. Containership:  
   To show the usage of data encapsulation and abstraction using containership providing controlled access through member functions. It promotes data integrity, modularity, and code reusability. Key concepts include classes, objects, member variables, member functions, and access modifiers.

# Design

Data Structures

* **Point:**
  + Attributes: **x, y** coordinates (integer)
  + Operations: distance\_to (calculates distance between two points)
* **Arc:**
  + Attributes: start, end points (references to *Point* objects), weight (integer)
  + Operations: get\_start, get\_end, get\_weight
* **Graph:**
  + Attributes: points (collection of *Point* objects), arcs (collection of *Arc* objects)
  + Operations: **insert\_point, insert\_arc, find\_shortest\_path**

Algorithms

* **Dijkstra's Algorithm:**
  + Data structures: distances (map of distances from the starting node to each node), visited (set of visited nodes)
  + Implementation: Adjacency Matrix
  + Steps:
    1. Initialize distances and visited sets.
    2. Select the unvisited node with the smallest distance.
    3. Update distances for neighbours.
    4. Mark the current node as visited.
    5. Repeat until all nodes are visited.

Design Decisions

* **Graph Representation:**
  + **Adjacency Matrix:** Adjacency Matrix has been used as it’s efficient for dense graphs
* **Point Representation:**
  + Character
* **Arc Weight:**
  + Integer

Efficiency Considerations

* **Time complexity:**   
  Dijkstra's algorithm has a time complexity of O((|V| + |E|) log |V|), where |V| is the number of vertices and |E| is the number of edges. Choose appropriate data structures and algorithms to optimize performance.
* **Space complexity:**   
  The algorithm requires O(|V|) space to store distances and visited nodes. Consider memory constraints when designing your graph representation.

# Algorithm

1. **Point Class:**

*CLASS Point*

*CHAR id*

*METHOD Point()*

*id = '-'*

*END METHOD*

*METHOD Point(INTEGER identifier)*

*id = identifier*

*END METHOD*

*METHOD ~Point()*

*END METHOD*

*END CLASS*

1. **Arc Class:**

*CLASS Arc*

*INT weight*

*Point \*start, \*end*

*METHOD Arc()*

*weight = 0*

*start = end = 0*

*END METHOD*

*METHOD Arc(Point \*beg, Point \*last, INT weigh)*

*weight = weigh*

*start = beg*

*end = last*

*END METHOD*

*METHOD ~Arc()*

*END METHOD*

*END CLASS*

1. **Graph Class:**

*CLASS Graph*

*Point \*points*

*Arc \*arc*

*INT p\_count, a\_count*

*METHOD Graph(INT point\_count, INT arc\_count)*

*p\_count = point\_count*

*a\_count = arc\_count*

*points = new Point[p\_count]*

*arc = new Arc[a\_count]*

*END METHOD*

*METHOD insert\_point(INT index, CHAR id)*

*END METHOD*

*METHOD insert\_arc(INT index, CHAR startId, CHAR endId, INT weight)*

*END METHOD*

*METHOD adjacency\_matrix()*

*END METHOD*

*METHOD dijkstra\_algorithm(CHAR param)*

*END METHOD*

*METHOD ~Graph()*

*delete[] points*

*delete[] arc*

*END METHOD*

*END CLASS*

1. **Graph::insert\_point Function:**

*METHOD insert\_point(INT index, CHAR id)*

*IF index < p\_count*

*points[index] = Point(id)*

*END IF*

*END METHOD*

1. **Graph::insert\_arc Function:**

*METHOD insert\_arc(INT index, CHAR startId, CHAR endId, INT weight)*

*Point \*start = nullptr*

*Point \*end = nullptr*

*FOR INT i = 0 TO p\_count - 1*

*IF points[i].id == startId*

*start = &points[i]*

*END IF*

*IF points[i].id == endId*

*end = &points[i]*

*END IF*

*END FOR*

*IF start AND end*

*IF index < a\_count*

*arc[index] = Arc(start, end, weight)*

*END IF*

*END IF*

*END METHOD*

1. **Graph::adjacency\_matrix Function:**

*METHOD adjacency\_matrix()*

*INT adj[p\_count][p\_count]*

*FOR INT i = 0 TO p\_count - 1*

*FOR INT j = 0 TO p\_count - 1*

*adj[i][j] = 0*

*END FOR*

*END FOR*

*FOR INT i = 0 TO a\_count - 1*

*adj[arc[i].start->id - 'A'][arc[i].end->id - 'A'] = arc[i].weight*

*adj[arc[i].end->id - 'A'][arc[i].start->id - 'A'] = arc[i].weight*

*END FOR*

*PRINT "Adjacency Matrix:"*

*FOR INT i = 0 TO p\_count - 1*

*FOR INT j = 0 TO p\_count - 1*

*PRINT adj[i][j] + "\t"*

*END FOR*

*PRINT ""*

*END FOR*

*END METHOD*

1. **Graph::dijkstra\_algorithm Function:**

*METHOD dijkstra\_algorithm(CHAR param)*

*INT index = param - 'A'*

*INT \*cost = new INT[p\_count]*

*CHAR \*visit\_status = new CHAR[p\_count]*

*FOR INT i = 0 TO p\_count - 1*

*cost[i] = 999999*

*visit\_status[i] = 'N'*

*END FOR*

*cost[index] = 0*

*FOR INT i = 0 TO p\_count - 1*

*INT min\_cost = 999999, min\_node*

*FOR INT j = 0 TO p\_count - 1*

*IF visit\_status[j] == 'N' AND cost[j] < min\_cost*

*min\_cost = cost[j]*

*min\_node = j*

*END IF*

*END FOR*

*visit\_status[min\_node] = 'Y'*

*FOR INT i = 0 TO a\_count - 1*

*INT beg = arc[i].start->id - 'A'*

*INT last = arc[i].end->id - 'A'*

*INT weight = arc[i].weight*

*IF beg == min\_node AND visit\_status[last] == 'N' AND cost[beg] + weight < cost[last]*

*cost[last] = cost[beg] + weight*

*END IF*

*IF last == min\_node AND visit\_status[beg] == 'N' AND cost[last] + weight < cost[beg]*

*cost[beg] = cost[last] + weight*

*END IF*

*END FOR*

*END FOR*

*PRINT "\nShortest distances from " + index + ":"*

*FOR INT i = 0 TO p\_count - 1*

*PRINT "\tTo Point " + (CHAR)('A' + i) + ": " + cost[i]*

*END FOR*

*delete[] cost*

*delete[] visit\_status*

*END METHOD*

1. **main():**

*METHOD main()*

*IFSTREAM infile("graph\_data.txt")*

*IF NOT infile*

*RETURN 1*

*INT total\_graphs*

*infile >> total\_graphs*

*FOR INT g = 0 TO total\_graphs - 1*

*INT point, arc*

*infile >> point >> arc*

*Graph g1(point, arc)*

*FOR INT i = 0 TO point – 1*

*CHAR pointId*

*infile >> pointId*

*g1.insert\_point(i, pointId)*

*END FOR*

*FOR INT i = 0 TO arc – 1*

*CHAR startId, endId*

*INT weight*

*infile >> startId >> endId >> weight*

*g1.insert\_arc(i, startId, endId, weight)*

*END FOR*

*PRINT "\n-------------------------------------------"*

*PRINT "Processing Graph " + (g + 1) + ":"*

*PRINT "-------------------------------------------"*

*g1.adjacency\_matrix()*

*g1.dijkstra\_algorithm(g1.points[0].id)*

*END FOR*

*infile.close()*

*RETURN 0*

*END METHOD*

# Data Description

The following is a vivid description pf the various data types used in the assignment- per the requirement of problem statement and other necessary attributes.

Data Types and Structures:

The program primarily relies on dynamic memory allocation so as to preserve resources and allocate the ~exact memory required.

**Classes:**

*Point Class:*

* + 1. Contains unique identifier to identify each node.
    2. Attributes: *char id;*
    3. Both of these have been declared under *public* access modifier for free access by the Graph class.
    4. The attributes are used to distinguish one point from another and represent the nodes (points) in the graph.
    5. The class has two constructors:
       1. A default constructor to initialize data member as follows:

*id = ‘-’;*

* + - 1. A parameterized constructor to initialize data member with the parameter provided as follows:

*Point (‘A’);*

* + 1. A destructor has been shown to depict deallocation. The definition is left empty.
    2. The objects of the class have been created dynamically in the form of an array of objects in Graph class.

*Arc Class:*

1. Contains start node, end node and the weight between those two nodes.
2. Attributes: *int weight;  
    Point \*start;  
    Point \*end;*
3. Both of these have been declared under *public* access modifier for free access by the Graph class.
4. The attributes are used to represent the source and the destination nodes (points) in the graph. Weight attribute holds the distance between the two nodes of Point class.
5. The class has one constructor:
   * + 1. A parameterized constructor to initialize data member with the parameter provided as follows:

*Arc (‘A’, ‘B’, 10);*

* + 1. A destructor has been shown to depict deallocation. The definition is left empty.
    2. The objects of the class have been created dynamically in the form of an array of objects in Graph class.

*Graph Class:*

1. Represents the entire graph structure and manages collections of Point and Arc Objects.
2. Attributes:  
    *Point \*points;   
    Arc \*arc;  
    int p\_count;  
    int a\_count;*
3. The points and arc attributes are dynamically allocated arrays of Point and Arc objects, respectively.
4. The p\_count and a\_count attributes keep track of the number of points and arcs in the graph.
5. The Graph class includes methods for inserting points and arcs, displaying the adjacency matrix, and implementing Dijkstra’s algorithm for finding the shortest path.
6. The class constructors and destructors handle dynamic memory allocation and deallocation for the arrays of Point and Arc objects.

**Matrices:**

*Adjacency Matrix:*

1. The adjacency matrix is a square matrix with   
     *rows = number of vertices  
    columns = number of vertices*
2. Declaration, definiton:  
    *adj[rows][columns] = adj[i][j]*

# File Descriptions

The following are the files that have been used to successfully compile the assignment and get the desired output:

1. *main.cpp* File
2. *main.exe* File
3. *graph\_data.txt* File

The format of the files is that of a text file and a c++ file with the respective extensions.

While *main.cpp* contains our code, on compilation, it creates a *.exe* file of the same name. The *.exe* file represents the executable file of the code so written.

The *.txt* holds the data of the graphs that are taken as sample data sets to run though the assignment code, in the algorithm and check whether the output delivers desired results.

Description of Files:

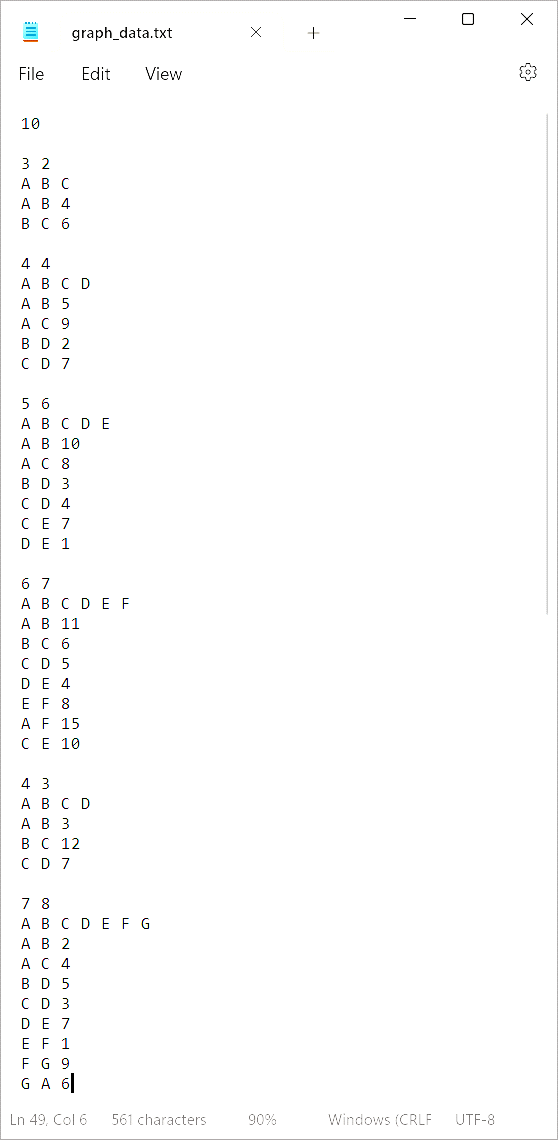
1. **main.cpp File:**
   1. This file contains the C++ code for the assignment.
   2. It defines the classes (Point, Arc, and Graph), implements the necessary algorithms (including Dijkstra's algorithm), and handles input/output operations.
   3. The file is essential for compiling and running the program.
   4. Format:   
       .cpp (C++ source code file)
   5. Purpose:   
       This file implements the solution to the problem, reading graph data, processing it, and outputting the results, such as the adjacency matrix and the shortest path calculations.
2. **main.exe File:**
   1. This is the executable file generated after compiling the main.cpp file. It can be directly run to execute the program without needing to recompile the source code.
   2. Format:   
       .exe (Windows executable file)
   3. Purpose:   
       This file represents the compiled version of the program, which can be executed to solve the problem without re-running the compilation process.
3. **graph\_data.txt:**
   1. Contains the input data for the program.
   2. It stores the sample graph data sets that the program reads and processes.
   3. Format:   
       .txt (text file)
   4. Content Format:  
       The file begins with an integer specifying the number of graphs.  
       For each graph, it contains:  
       Integer representing the number of vertices (points) in the graph.  
       Integer representing the number of edges (arcs) in the graph.  
       Characters representing the vertex identifiers.  
       Edges, where each edge is specified by the starting vertex, ending vertex, and the weight (distance) between them.
4. Purpose:  
    This file provides the data necessary for the program to create and analyse graphs. It is used to test the implementation of the graph structures and algorithm

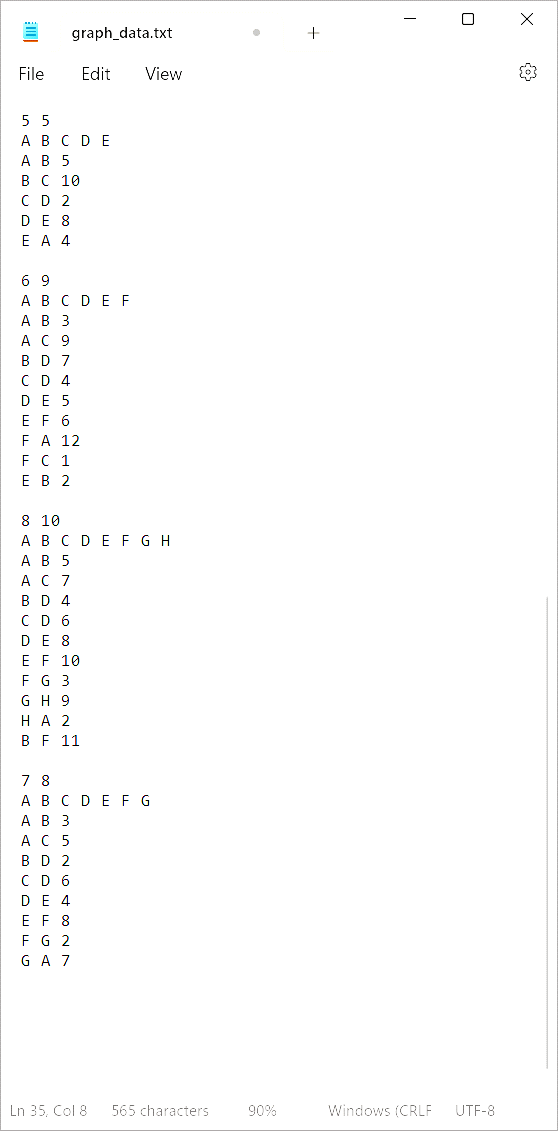
Use of File Handling:

1. Input Operations:   
   The graph\_data.txt file is read by the program, which uses the information to dynamically construct the graph by creating instances of the Point and Arc classes.
2. Output Operations:   
   The program outputs results to the terminal, such as the adjacency matrix and the shortest paths computed using Dijkstra's algorithm.

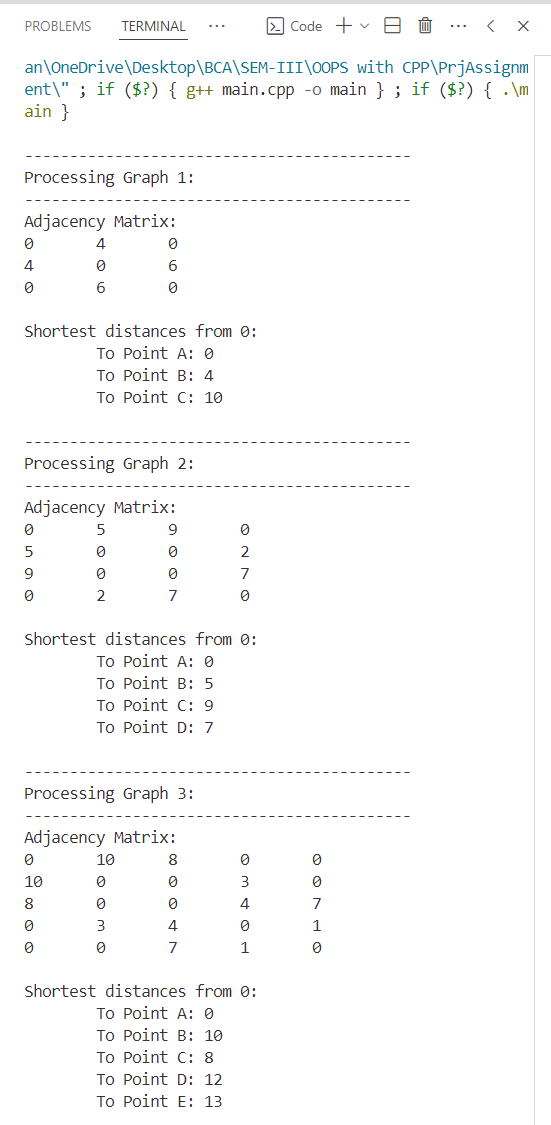
# Screenshots

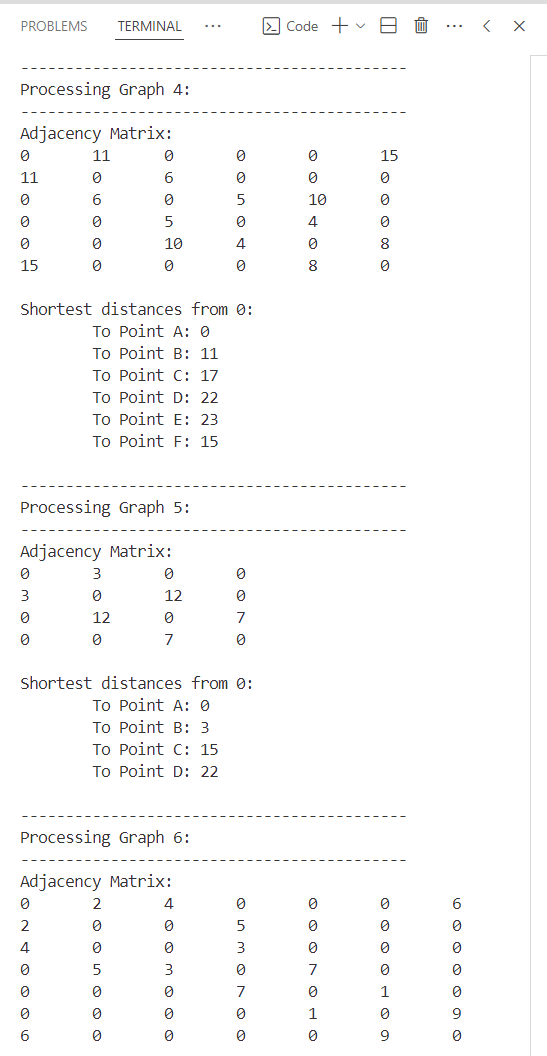
Data Set:

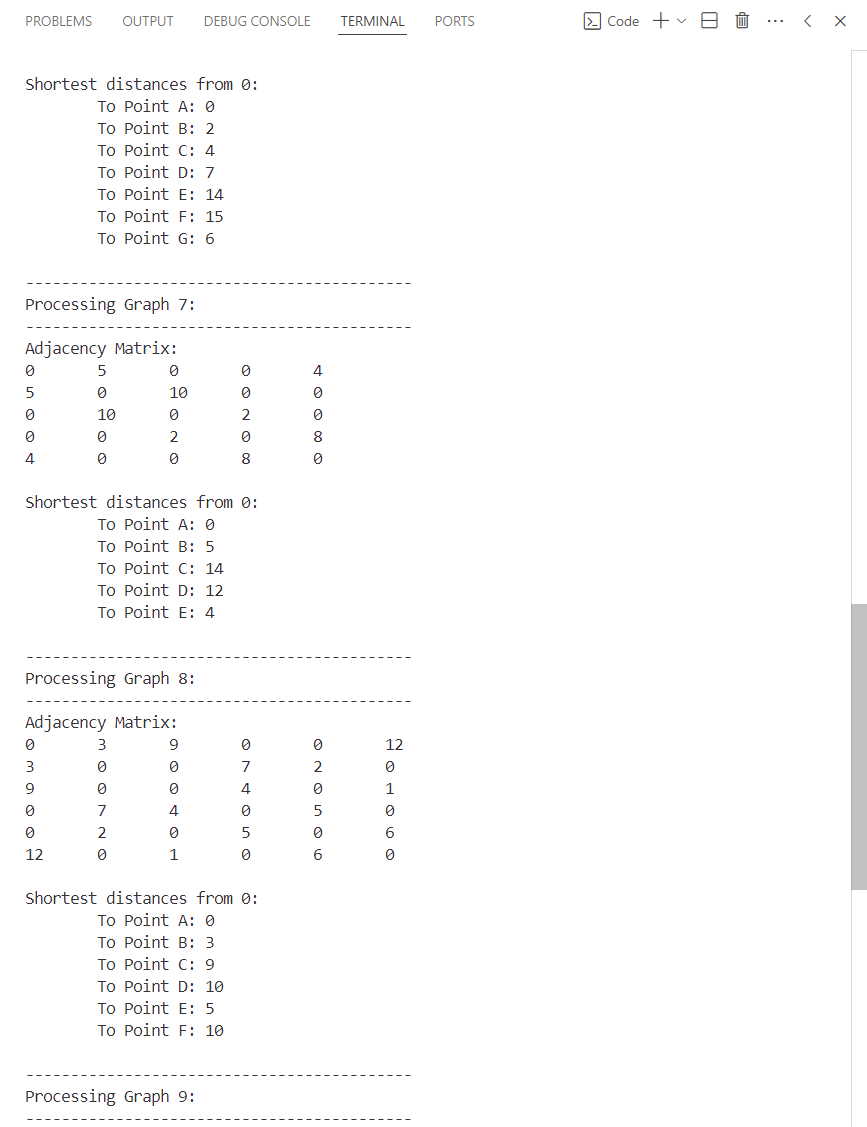


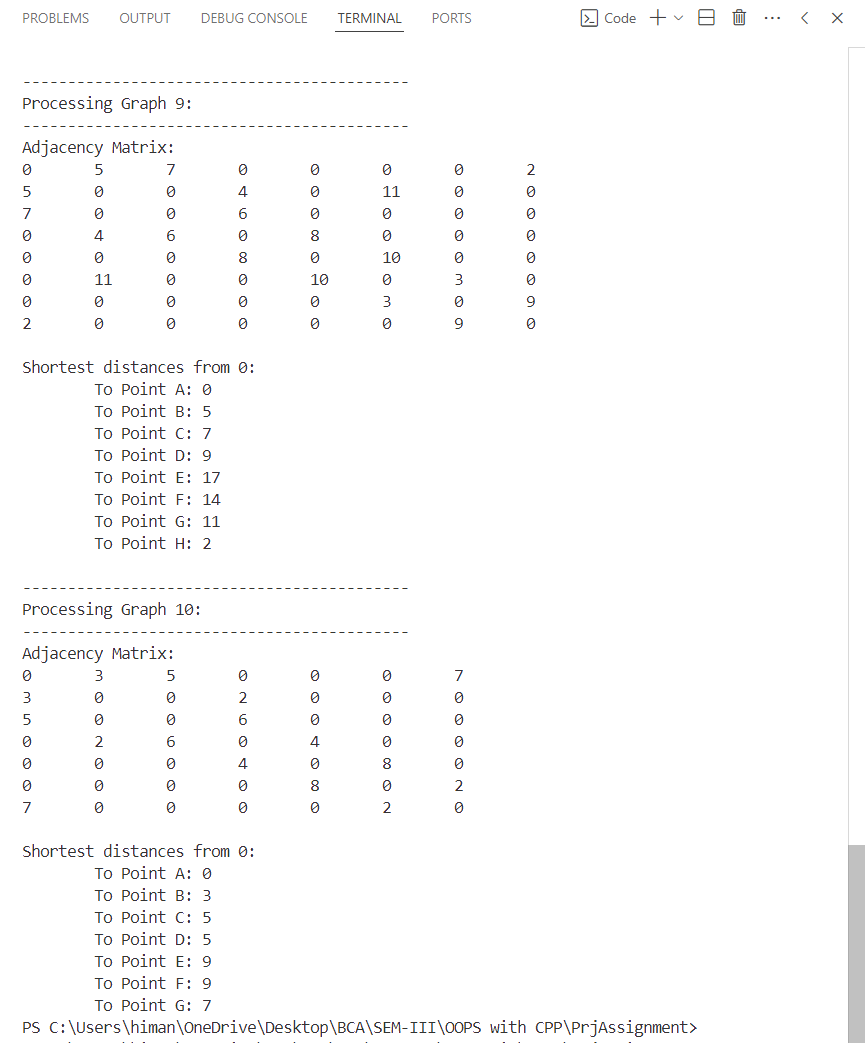


Output:









# Applications

The concept of graphs has proven to be incredibly useful cross various fields. Given the wide application of graphs, we are always attempting to find a more optimized algorithm to find the shortest distance to further enhance the accessibility in the implementable fields.

Following the same, we so far have various Shortest Distance Algorithm, this includes- MultiSource BFS, Dijkstra algorithm, Bellman-Ford algorithm, Topological Sort, etc,. Since the given assignment discusses and shows an implementation of Dijkstra algorithm.

The following are some of the most successful and popular implementations of the said algorithm:

1. GPS Navigation Systems  
   *(e.g., Google Maps, In-built car navigation systems, etc)*
   * Uses optimized versions of the algorithm to find the optimal route from a source to destination. Unlike our solution, it factors in real world situations and problems a driver, or traveller may face on the road.
   * Recently, Google Maps, has started to factor in and return the amount of fuel a user saves by choosing to take the suggested path. This is in terms of fuel efficiency and green emissions.
2. Telecommunications and Network Routing  
   *(e.g., well the Internet)*
   * Use the shortest path algorithm to find the shortest path for quick, and effective delivery.
   * Each point represents a router or switch, and each Arc, the connection between the points.
3. Public Transportation  
   *(e.g., buses, trains, etc)*
   * Stations are represented as points and the distance between them as arcs.
   * Popular in delivery apps like Zomato, Swiggy, BlinkIt, BigBasket, Amazon, etc.
   * Also implementable in Airline and Flight Routing in finding the most efficient flight paths between destinations, optimizing for distance, time, cost, etc.
4. Social Networks  
   *(e.g., Facebook, Twitter, Instagram, etc)  
   ~~not TikTok, that’s banned in India~~*
   * Graphs are all about connections between nodes, now nodes are people and the distance between them and their mode of connection, out arcs.
   * Every relationship on social media- friendships, follows, likes, shares,… everything is depicted in the form of graphs.
   * This allows mapping between them and is also how influencers are separated from regular users.

The aforementioned, represent the most popular uses of Dijkstra we encounter but never realise. More such include Supply Chains, Distribution Networks, and even Game Development.

With the right implementations, it has proven to be an ideal model to plan out emergency response scenarios. Cities are modelled as graphs, intersections become points, roads the arcs.

The ability of computing shortest paths between two points is more than just a theoretical method to depict data in a dynamic way. It has been ingrained in to our day-to-day lives to the point of no comebacks.

As technology evolves, more algorithms emerge and evolve further enhancing our lifestyle and increasing our dependency on them.

# Conclusion

The assignment demonstrated the application of Dijkstra’s Algorithm for finding the shortest path in an undirected graph through the implementation of the Point, Arc, and Graph classes.   
We modelled a graph where vertices represent nodes and edges represent connections with the relevant weights.

The primary objective of this assignment was to compute the shortest distance between any two vertices in the graph, which was effectively achieved using Dijkstra’s Algorithm.

Our approach involved defining a Graph class that manages collections of Point and Arc objects, and implementing Dijkstra’s Algorithm to calculate the shortest path. The results validate the efficiency and effectiveness of the algorithm, providing shortest path distances from a source vertex to all other vertices in the graph.

The project also offered insights into the various design decisions, including the choice of adjacency list representation for sparse graphs and the handling of dynamic memory allocation. It provided a hands-on experience in coding, debugging, and optimizing algorithms, reinforcing the theoretical knowledge gained through the study of graph theory and algorithms.

# References

Books:

1. Varsha H. Patil (2012), *Data Structures Using C and C++*. Oxford.
2. Singh, J. P. (2013). *Discrete mathematics for undergraduates* (revised ed.). Ane Books.
3. Yedidyah Langsam, Moshe J. Augenstein, Aaron M. Tenenbaum (2015), *Data Structures Using C and C++* (second edition). Pearson.

Research Papers:

1. Dreyfus, S. E. (1969). An appraisal of some shortest path algorithms. *Operations Research*
2. Javaid, A. (2012). Understanding Dijkstra algorithm. SSRN Electronic Journal.

Online Resources:

1. YouTube:
   1. Striver’s Take U forward
   2. Gate Smashers
   3. Abdul Bari
2. Reading:
   1. GeeksforGeeks
   2. Medium Articles

Our ✨favourite AI✨ Chatbots:

1. ChatGPT Auto