A

Minor Project Report

on

**Metro Assistant For DMRC**

Submitted in partial fulfillment of the requirements for the award of degree of

**Bachelor of Technology**

in

**Computer Engineering**

by

**Ashutosh Singh**

**(17001003010)**

Under supervision of

**Dr. Lalit Mohan Goyal**



**Department of Computer Engineering**

**J. C. BOSE UNIVERSITY OF SCIENCE & TECHNOLOGY, YMCA**

**FARIDABAD-121006**

**May 2020**

**CANDIDATE’S DECLARATION**

I hereby certify that the work which is being carried out in this Minor Project titled **“Metro Assistant for DMRC”** in fulfillment of the requirement for the degree of Bachelor of Technology in Computer Engineering and submitted to “**J. C. Bose** **University of Science and Technology, YMCA, Faridabad**”*,* is an authentic record of my own work carried out under the supervision of Dr. Lalit Mohan Goyal.

The work contained in this thesis has not been submitted to any other University or Institute for the award of any other degree or diploma by me.

Ashutosh Singh

17001003010

**CERTIFICATE**

This is to certify that the work carried out in this project titled **“Metro Assistant for DMRC ”** submitted by Ashutosh Singh to “**J. C. Bose** **University of Science and Technology, YMCA, Faridabad**” for the award of the degree of Bachelor of Technology in Computer Engineering is a record of bona fide work carried out by him under my supervision. In my opinion, the submitted report has reached the standards of fulfilling the requirements of the regulations to the degree

Dr. Lalit Mohan Goyal

(Supervisor)

Assistant Professor,

Department of Computer Engineering,

J. C. Bose University of Science and Technology, YMCA, Faridabad

Dr. Komal Kumar Bhatia

Chairman,

Department of Computer Engineering,

J. C. Bose University of Science and Technology, YMCA, Faridabad

**TABLE OF CONTENTS**

CANDIDATE’S DECLARATION ii

CERTIFICATE iii

**CHAPTER 1: INTRODUCTION**  **1-5**

1.1 Introduction

1.1.1 Brief Intro to DMRC

1.1.2 Introduction to the Metro Assistant

1.2 Basic Terminology 2

1.2.1 C++

1.2.2 STL

1.3 Motivation & Problem Statement 5

1.4 Objectives and Scope 6

**CHAPTER 2: LITERATURE REVIEW 7-15**

* 1. Graphs: Adjacency matrix and lists
  2. DFS
  3. Path Finding Algorithms 9

**CHAPTER 3: DESCRIPTION & METHODOLOGY**

3.1 Overview of steps involved in producing the result

3.2 Data Gathering and Explanation/Reading of the gathered data

3.3 Getting input

3.4 Finding connections for the inputs.

3.5 Finding optimal path.

3.6 Modularity in the process.

**CHAPTER 4: HARDWARE AND SOFTWARE REQUIREMENTS**

4.1 Hardware Requirements

4.2 Software Requirements

**CHAPTER 5: RESULTS**

**CHAPTER 6: CONCLUSION**

**REFERENCES 40-41**

**BRIEF PROFILE OF STUDENT**

**CHAPTER 1: INTRODUCTION**

* + 1. **Introduction to DMRC**

DMRC refers to the Delhi Metro Railways Corporation.

The Delhi Metro has been instrumental in ushering in a new era in the sphere of mass urban transportation in India. The swanky and modern Metro system introduced comfortable, air conditioned and eco-friendly services for the first time in India and completely revolutionized the mass transportation scenario not only in the National Capital Region but the entire country.   
  
Having constructed a massive network of about 389 Km with 285 stations (including NOIDA-Greater NOIDA Corridor and Rapid Metro, Gurugram) in record time in Delhi, NCR, the DMRC today stands out as a shining example of how a mammoth technically complex infrastructure project can be completed before time and within budgeted cost by a Government agency.   
  
The Delhi Metro Rail Corporation Limited (DMRC) was registered on 3rd May 1995 under the Companies Act, 1956 with equal equity participation of the Government of the National Capital Territory of Delhi (GNCTD) and the Central Government to implement the dream of construction and operation of a world- class Mass Rapid Transport System (MRTS).

* + 1. **Introduction to Metro Assistant**

The DMRC has divided its journey into lines which connect with each other at exchanges i.e stations common between two lines the trains running on one line are exclusive to that line so if a person needs to go from a station on one line to a station on some other line, they have to change their line in the middle of journey.

These changes make it difficult to navigate the metro but it is necessary as all trains cant go to each station .with Such a vast Network of stations and so many trains and almost 10-11 lines on top of that, it is extremely difficult for people to figure out the most efficient path for their journey from station A to station B. Although Prices will stay the same no matter what path you take the time taken to travel can be reduced drastically if we choose our exchanges properly.

This tool will function as an assistant to the user providing them with the fastest and shortest path to get from one station to another. The tool will also calculate the price of the journey. The user gives their origin and destination stations as an input and the tool finds the path for them and tells them where to change between lanes how many stops to ride in a lane to get to their destination.

Our tool will give the user multiple paths with differing time taken by each path and it will be ordered according to which takes how much task but user still has a choice to select a path with a larger distance due to some other work they may have a in the id journey.

As the saying goes the customer/User is the king and deserves a choice in every decision he/she will take.

**1.2 Basic Terminology**

This section describes all the basic terminology used in the project report. These terms form the core of the project. Before moving forward to project details, these terms are to be well understood.

1.2.1 *C++*

C++ is a general-purpose object-oriented programming language developed by Bjarne Stroustrup of Bell Labs in 1979. C++ was originally called ‘C with classes,’ and was built as an extension of the C language. Its name reflects its origins; C++ literally means ‘increment C by 1.’

It was renamed C++ in 1983, but retains a strong link to C, and will compile most C programs. Compared to C, C++ added object-oriented features to C such as classes, abstraction, and inheritance.

C++ is considered a mid-level programming language, combining some elements of low-level programming languages, such as the need to learn memory management, with high-level features. Because of this, C++ is considered quite a complex language — in comparison to languages such as Python you need to know quite a bit more before you can create your first truly useful programs.

1.2.2 *STL Standard Template Library*

C++ STL is a set of data structures and algorithms that we normally encounter during coding. For example, while solving a problem you wanted to use linked list, so will you create a linked list from scratch? The answer is no, you will use list built into c++ stl library. There are a lot of similar examples which I will be giving throughout this article.

**STL has four components**

* Algorithms
* Containers
* Functors
* Iterators

**Algorithms**

The header algorithm defines a lot of commonly used algorithms that we can use out of box. They provide various operation for containers. Some well-known algorithms are

sort() — for sorting.

binary\_search() — for searching element

**Containers**

Containers or container classes store objects and data. Some container classes are given below

vector, list, forward\_list, queue, priority\_queue, stack, set, multiset, map, multimap, unordered\_set

**Iterators**

As the name suggests, iterators are used for working upon a sequence of values.

**Functors**

The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors.

Key Reasons to Use C++

1. It is extremely Flexible

C++ is a multi-paradigm coding language. This means that it supports other styles such as procedural programming, in addition to object-oriented programming. These paradigms are essentially different ways of looking at and solving a coding problem; two different C++ coders could look at and solve the same coding problem in different ways. Paradigms can also be combined to get the most efficient result. Having different ways of solving problems makes C++ more complex, also make it more powerful.

1. C++ Excels at Delivering Performance

C++ is used in many industries and can be used to write almost anything, it particularly excels in delivering performance and using system resources efficiently. The control C++ gives the user over system resources enables a skilled coder to write a program that is quicker and more powerful than a similar program written in another programming language.

* 1. **Motivation & Problem Statement**

The motivation behind this project is to facilitate the journey common man in crowded city like Delhi and understanding the complexities of its vast subway like public transportation network.

As an outsider I had to once face enormous challenges understanding the DMRC network and I had to literally sit down with its maps every time I wanted to go somewhere. DMRC is

a very aspirational project but given how it has to be fitted in an already small, crowded city it has its limitations and this tool will be helping hand to make people enjoy the journey rather than focus on figuring out the complexities of the Map of DMRC.

The DMRC has divided its journey into lines which connect with each other at exchanges i.e stations common between two lines the trains running on one line are exclusive to that line so if a person needs to go from a station on one line to a station on some other line, they have to change their line in the middle of journey.

These changes make it difficult to navigate the metro but it is necessary as all trains cant go to each station .with Such a vast Network of stations and so many trains and almost 10-11 lines on top of that, it is extremely difficult for people to figure out the most efficient path for their journey from station A to station B. Although Prices will stay the same no matter what path you take the time taken to travel can be reduced drastically if we choose our exchanges properly. For reference see the image below.



**CHAPTER 2: LITREATURE REVIEW**

In this part lets review the basic principle which were used to break the Problem into various parts to solve it. Its an example of using the Divide and conquer Strategy of problem solving

Two rather interesting fundamentals of computer programming have been used to write this tool those are as followed.

**2.1 Graphs**

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph can be defined as,

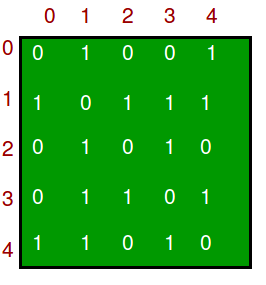


In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like LinkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender, locale etc.

Graphs in Memory are stored using adjacency matrix.

adjacency Matrix is a 2D array of size V x V where V is the number of vertices in a graph. Let the 2D array be adj[][], a slot adj[i][j] = 1 indicates that there is an edge from vertex i to vertex j. Adjacency matrix for undirected graph is always symmetric. Adjacency Matrix is also used to represent weighted graphs. If adj[i][j] = w, then there is an edge from vertex i to vertex j with weight w.



This is used to representthe actual connections in the metro lines instead of using Graph to represent the entire map we made a graph only for the connection of lines.

A graph representation is a technique to store graph into the memory of computer.

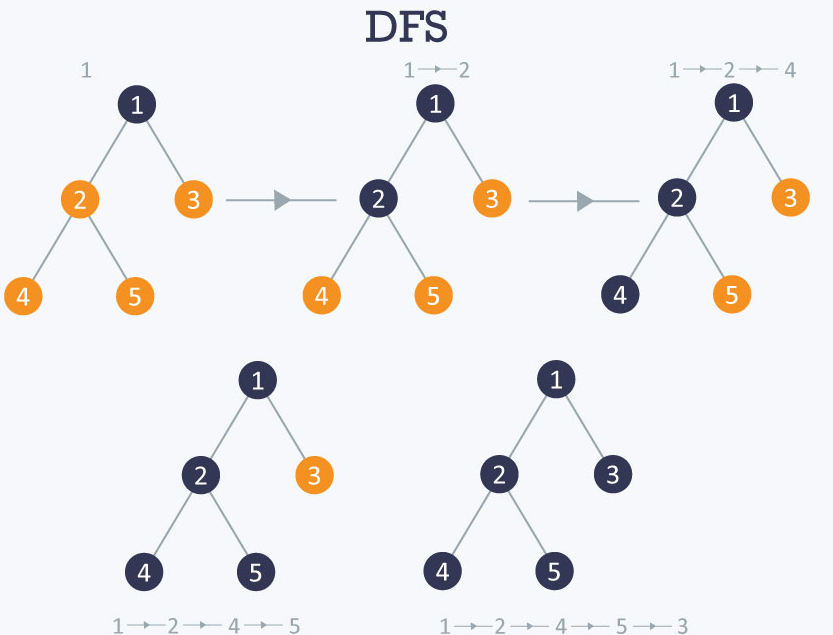
**2.2 DFS (Depth First Search)**

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected. recursive nature of DFS can be implemented using stacks.

The basic idea is as follows:  
1. Pick a starting node and push all its adjacent nodes into a stack.  
2. Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.  
3. Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

**Working of DFS to find path from 1 to 3**



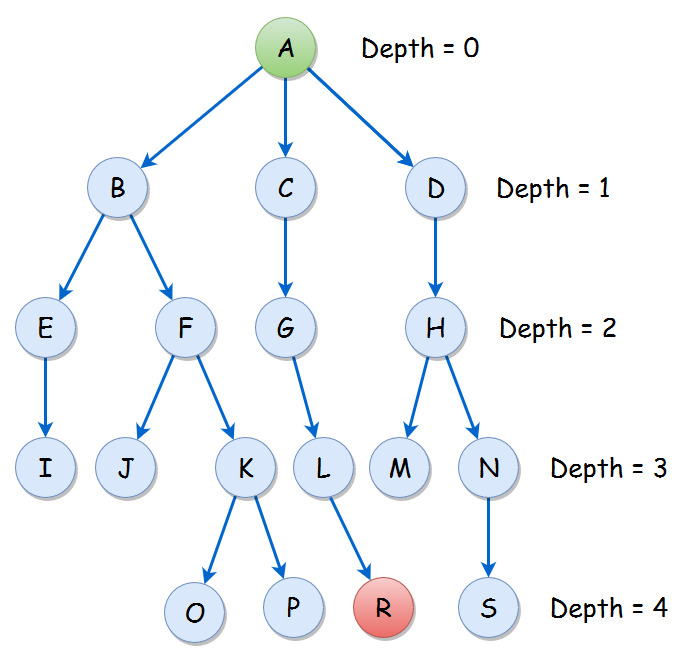
This approach by us to find link between the lines. Using an improved version of DFS called IDDFS.

Why use IDDFs.

* In DFS, you would recursively look at a node’s adjacent vertex. DFS may not end in an infinite search space. Also, DFS may not find the shortest path to the goal. DFS needs O(d) space, where d is depth of search.
* BFS consumes too much memory. BFS needs to store all the elements in the same level. In the case of a tree, the last level has N / 2 leaf nodes, the second last level has N / 4. So, BFS needs O(N) space.

Iterative deepening depth first search (IDDFS) is a hybrid of BFS and DFS. In IDDFS, we perform DFS up to a certain “limited depth,” and keep increasing this “limited depth” after every iteration.

Let us take an example to understand this –



|  |  |
| --- | --- |
| **DEPTH** | **DLS traversal** |
| 0 | A |
| 1 | A B C D |
| 2 | A B E F C G D H |
| 3 | A B E I F J K C G L D H M N |
| 4 | A B E I F J K O P C G L R D H M N S |

This approach helps us find the resulting connections with minimum line changes. That is important as it helps us save time wasted on changing lines and gives us data back already sorted according to the no of different lines which need to be used to find the connection. Which helps us minimise the resultant path length and time.

**2.3 Path finding algorithms**

Blind search algorithms such as BFS and DFS all possibilities; starting from the given node, they iterate over all possible paths until they reach the goal node. These algorithms run in O(V+E), or linear time, where V is the number of vertices, and E is the number of edges between vertices.

At its core, a pathfinding method searches a graph by starting at one vertex and exploring adjacent nodes until the destination node is reached, generally with the intent of finding the cheapest route. Although graph searching methods such as a BFS would find a route if given enough time, other methods, which "explore" the graph, would tend to reach the destination sooner. An analogy would be a person walking across a room; rather than examining every possible route in advance, the person would generally walk in the direction of the destination and only deviate from the path to avoid an obstruction, and make deviations as minor as possible.

Two primary problems of pathfinding are (1) to find a path between two nodes in a graph ; and (2) the shortest path problem—to find the optimal Shortest Path. Basic algorithms such as BFS and DFS search address the first problem by exhausting all possibilities; starting from the given node, they iterate over all potential paths until they reach the destination node. These algorithms run in O ( | V | + | E | ) {\displaystyle O(|V|+|E|)} O(V+E) , or linear time, where V is the number of vertices, and E is the number of edges between vertices

# *A\* algorithm*

A\* is a variant of Dijkstra's algorithm commonly used in games. A\* assigns a weight to each open node equal to the weight of the edge to that node plus the approximate distance between that node and the finish. This approximate distance is found by the heuristic, and represents a minimum possible distance between that node and the end. This allows it to eliminate longer paths once an initial path is found. If there is a path of length x between the start and finish, and the minimum distance between a node and the finish is greater than x, that node need not be examined.

A\* uses this heuristic to improve on the behaviour relative to Dijkstra's algorithm. When the heuristic evaluates to zero, A\* is equivalent to Dijkstra's algorithm. As the heuristic estimate increases and gets closer to the true distance, A\* continues to find optimal paths, but runs faster (by virtue of examining fewer nodes). When the value of the heuristic is exactly the true distance, A\* examines the fewest nodes. (However, it is generally impractical to write a heuristic function that always computes the true distance, as the same comparison result can often be reached using simpler calculations – for example, using Manhattan Distance over Euclidean space and 2-D space. ) As the value of the heuristic increases, A\* examines fewer nodes but no longer guarantees an optimal path. In many applications (such as video games) this is acceptable and even desirable, in order to keep the algorithm running quickly.

**CHAPTER 3: DESCRIPTION AND METHODOLOGY**

**What does the project do?**

This tool will function as an assistant to the user providing them with the fastest and shortest path to get from one station to another.

The tool will also calculate the price of the journey.

The user gives their origin and destination stations as an input and the tool finds the path for them and tells them where to change between lanes how many stops to ride in a lane to get to their destination.

**3.1 Overview of steps involved in producing the result**

* Gathering and brining the data into the codebase:

The gathering of information such as names of stations, names of lines and connection between the lines and storing them into text files.

* Reading the Data into workable graphs and HashMap’s which can used to define the metro connections and network. The code for this is attached in the below process.
* Getting the inputs in the form of starting and ending stations.
* Finding the lines associated to the stations given by the user.
* Finding connection between those lines using the maps and graphs generated in the previous steps.
* Finding the most optimal path using the stations no to calculate the distance in the fashion of A\* algorithm as it is used both the cost of reaching a given point and cost of the future journey into consideration.
* Representing the output as a set of line changes and stations at which it happens to reach from source to destination.

**3.2 Data Gathering and explanation of the gathered data**

We collected the following data items:

1. Names of all the stations.
2. Lines to which a particular station belongs to.
3. The station no that station has on the given line.

All this data was gathered from official metro website <http://www.delhimetrorail.com/> .

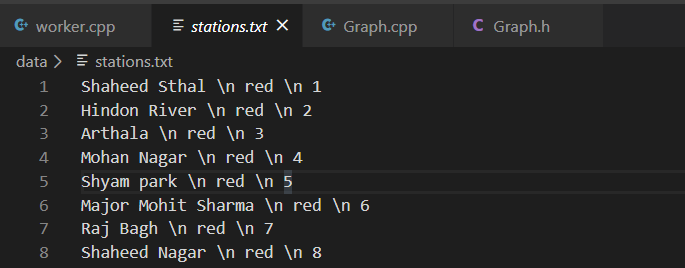
All this data was later stored into two text file which work as resource or data sources for the map making and path finding algorithms.

Name of files:

1. *Stations.txt*

This file contains the name of stations along with its line and station no.

Format: Station name \n Line name \n station no



This file is read us file handling to make a HashMap of all the stations , line name combination, with the code mentioned below.

typedef pair<string,string> pairstr;

void hashes(map< pairstr, int > &,vector<string>&);

//function to make hash map of stations with their line.

void hashes(map<pairstr, int> &result,vector<string>& station\_name) {

    ifstream myfile("data/stations.txt");

    if (myfile.is\_open())

    {

        string full;

        while (getline(myfile, full))

        {

            string word = "";

            int j = 0;

            pairstr x;

            int val=0;

            for (int i = 0; i < full.length(); i++) {

                if (full[i] == ' ' && full[i + 2] == 'n'){

                    if(j==0){

                        x.first=word;

                        station\_name.push\_back(word);

                    }

                    if (j == 1) {

                        x.second = word;

                        int a = max((int)full[full.length() - 1]-48,0);

                        int b = max((int)full[full.length() - 2]-48,0);

                        val = b \* 10 + a;

                        break;

                    }

                    j++;

                    i = i + 3;

                    word = "";

                }

                else {

                    word=word+full[i];

                }

            }

            result[x]=val;

        }

        myfile.close();

        return;

    }

    else cout << "Unable to open file";

}

How a HashMap looks like: -

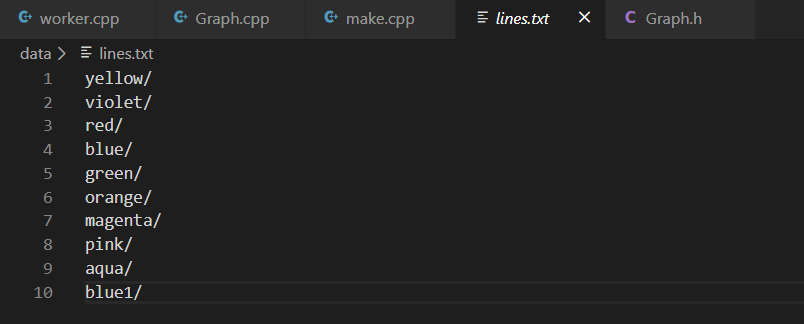
Format: {“station name”, “line”}🡪 number

● Example - ○ {“AIIMS”, “yellow”} 🡪 14

1. *Lines.txt*

This file contains the names of all the distinct lines in the metro.

Format: Line name /



This file is read to get all the line names into the code so it can be used. With the following code piece.

void load\_lines(vector <string> &lines) {

    string line;

    ifstream myfile("data/example.txt");

    if (myfile.is\_open())

    {

        while (getline(myfile, line))

        {

            string s="";

            for(int i=0;i<line.length();i++){

                   if(line[i]=='/')

                    break;

                    s+=line[i];

            }

            lines.push\_back(s);

        }

        myfile.close();

        return;

    }

    else cout  << " Unable to open file";

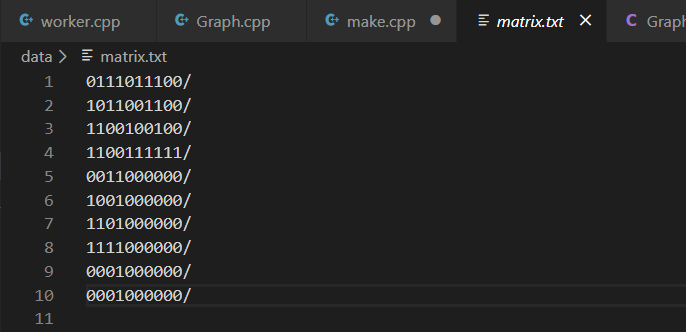
}

From here we get an array with names of lines as elements.

*3.Matrix.txt*

This file is used to store the adjacency matrix into the codebase to make a graph of all lines connection. This is one of the most important parts of the code as it helps us in the primary steps involved in the path finding algorithm.

Format: N\*N matrix to show all the row no x represents the line no x and column j represents the lines no j in the array of lines we made above. Followed by the



This matrix is used to represent the graph of all the lines. This graph in conjunction with the other modules like HashMap are used to figure out the optimal path.

The Graph is as follows:



This graph is made from the matrix.txt file using the following code.

void graph\_function(vector <vector<int>> &graph) {

    string l;

    ifstream myfile("data/matrix.txt");

    if (myfile.is\_open())

    {

        int i=0;

        while (getline(myfile, l))

        {

            for(int j=0;j<l.size();j++){

                if(l[i]=='/')break;

                if(l[j]=='1'){

                    graph[i].push\_back(j);

                }

            }

            i++;

        }

        myfile.close();

        return;

    }

    else cout  << " Unable to open file";

}

The output of the graph function has already been shown in the form of the graph which we have linked just above.

**NOTE: How to add new stations or lines or connections**

If you wish to add a new station or line you will need to manipulate the three files mentioned above that are:

1. stations.txt: Add the names of new stations to it.

2. lines.txt: Add any new lines here also updates the stations on new lines in stations.txt

3. matrix.txt: If there is any new connections in the graph update the matrix.txt file according to the rules of editing an adcancey matrix also add new stations in stations.txt and new line associated to the change in lines.txt.

A change in matrix is the highest order of change and should always be followed with corresponding changes in stations and lines. Txt

Similarly, a change in lines should be followed by a change in stations.txt

A station can be added independently.

**3.3 Getting Inputs**

**// add here**

**3.4 Finding connections for the inputs.**

This step has two Basic parts that are:

1. Finding the lines associated to the names of stations provided in the input.
2. Finding The connections between the lines from step 1

Let’s explain both paths in detail.

*3.2.1 Finding the lines associated to the names of stations provided in the input*

In this step we take name of Station from the input and send it to a function which searches for the line on which the input stations belong on. This is done using by matching the

{station name, line name} in the HashMap which was made earlier for each line using the vector of lines that we made earlier.

If there is match, we return the name of the line , which is then assigned a no according to its index in the lines in the lines vector which was made earlier so that it can be passed to the next steps.

The code to do all this is mentioned below kindly look into it. In this code the function takes three inputs which are:

1. Vector<string> &l which represents the line vector
2. Map< pairstr , int> &m which represents the HashMap
3. String name which is the name of station.

All these are passed via references to avoid copying of the entire data to save space and time.

string line1=line\_name(line,m,station1);

string line2=line\_name(line,m,station2);

string line\_name(vector < string > &l, map< pairstr, int > &m , string name ){

    for(int i=0;i<l.size();i++){

        if(m.find ( {name, l[i]} ) != m.end() ){

            return l[i];

        }

    }

    return "invalid station name";

}

*3.2.2 Finding the Connections between the lines from above code:*

In this step we take no of lines associated with the input stations and find the connection between them using the graph which we made using the matrix.txt file in the data ggathering phase

To calculate this connection, we use IDDFS as IDDFS is significantly faster and space efficient as compared to BFS and DFS.

We also limited the depth to 4 to avoid finding paths that are require too many line changes as that is just impractical. But a developer in future can change it according to the needs of his/her project or demand.

The code to do all this is mentioned below kindly look into it. In this code the function takes three inputs which are:

1. Vector<vector<int>> &Graph which represents the Graph we made earlier
2. Int &src no associated to the source station from 3.2.1
3. Int &des no associated to the destination station from 3.2.1

vector<vector<int >>dfs\_paths;

dfs\_paths=path\_function(graph,a,b);

bool vistied(int& node,vector<int >& path){

    for(auto i:path){

        if(i==node)return false;

    }

    return true;

}

vector<vector<int>>path\_function(vector<vector<int>>& graph,int& src,int& des)

{

    vector<vector<int >>ans;

    queue<vector<int >>que;

    vector<int >curr\_path;

    curr\_path.push\_back(src);

    que.push(curr\_path);

    while(!que.empty()){

        curr\_path=que.front();

        que.pop();

        int n=curr\_path.size();

        int last=curr\_path[n-1];

        if(last==des && n<=4){

            ans.push\_back(curr\_path);

            continue;

        }

        for(auto i:graph[last]){

            if(vistied(i,curr\_path)){

                vector<int >new\_path(curr\_path);

                new\_path.push\_back(i);

                que.push(new\_path);

            }

        }

    }

    return ans;

This path function gives us the Connection between the two lines in the form no associated to each line in the line hash or line vector.

But this is Just an overview of the all the connections between source and destination line path which can produce multiple paths with very different time dimensions,

The variable dfs\_paths will have multiple paths in it which will be used in future steps.

**3.5 Finding connections for the inputs.**

In this step we calculate the exact length of connections found in the last step and stored in dfs\_paths so that we can find the most optimal path possible.

This is done using a A\* like algorithm where we calculate the distance for future by calculating the path cost of dfs\_paths and use it as Heuristic value for our algorithm.

There are two parts in this:

1. Make a hash for all the connecting stations: This is done, so that we know which stations are present at which intersection and use the no’s associated with the stations to calculate the cost of the path.
2. Calculate the net path cost: We first calculate the cost for various connections and taken to reach the destinations and add our source to line exchange cost into it to get the exact value and find the most optimal result.

*3.5.1 Making a hash for all the connecting stations:*

This is needed as we don’t want to calculate the path one station at a time thus we can use this hash to reduce the calculations involved as it will reduce the tedious work of going station by station from over one line so that we can find where the other line intersect it. Rather than doing that we can just finding the connecting stations from a simple code and use the station name from its result to produce the cost of traveling from line X to line Y in one scoop rather than travelling each station.

To achieve this in our code we had to manipulate our stations.txt file a little and also our stations hash function as well so that, we can reach the desired output. A comparison between before change after change architecture of our hash function is done below along with the explanation of how it helps us reach our desired goal.

Before Change {Station name}🡪 {line name, station no}

After Change {Station name, line name}🡪 station no

Now This newer version has a hash that depends on both line and station name thus it must have all the occurrences of a stations in it if a station occurs on multiple lines.

To accommodate this, we had to store multiple occurrences of a station in the station .txt file for example. Kashmiri Gate is station that’s on three metro lines so its stored like this.

Kashmiri Gate \n red \n 16

Kashmiri Gate \n yellow \n 12

Kashmiri Gate \n violet \n 1

And its similarly present three times in the hash. Here is the piece of code that provides us with a hash of all the intersecting station so we can move between them faster and calculate their distances in O (1) time. Reducing our overall time.

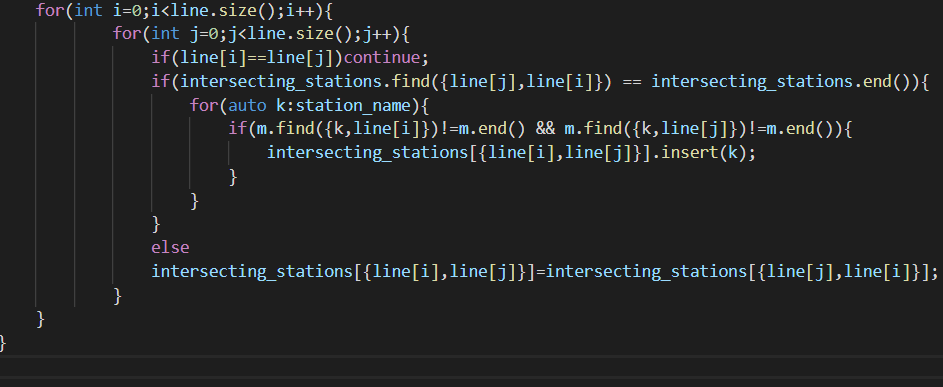
The code to do all this is mentioned below kindly look into it. In this code the function takes 4 inputs and some data structures which are:

1. map< pairstr, set< string>>&: intersecting stations this is data structure that gets all the output in this.
2. vector<string >& line: This is the hash of line which we made above.
3. map< pairstr, int > &m: This is the HashMap made in the very first step.
4. vector<string >& station\_name: name of the station.

map< pairstr ,set< string >> intersecting\_stations;

intersecting\_stations\_funtion(intersecting\_stations,line,m,station\_name);

void intersecting\_stations\_funtion(map< pairstr ,set< string >>& intersecting\_stations,vector<string >& line,map< pairstr, int > &m,vector<string >& station\_name){



The output is received in the Intersecting stations map remember this at it will be used in the future.

Also, it has structure like {line 1, line 2}🡪Station name

Where line 1, line 2 are the key and stations name is a value, example in DMRC will be

{Violet, Blue}🡪 Mandi House.

This shows that Mandi House is the connecting station for Violet and Blue line.

*3.5.2 Finding the Optimal Path.*

Every Data structure or hash map or connection set we have made so far has all been step.

This step uses the connection map we made in 3.5.1 to produce quick results for the path cost of all the line exchanges With the help of station HashMap we made in the very first step.

It goes something like this.

* We get the **input stations** let them be **a🡪 Badarpur border** and **b🡪 AIIMS**
* Find lines they are on. Let them be **Line a🡪Violet** and **Line b🡪 Yellow**
* Then we get connections/ exchanges we must use to reach from Line a to Line b , we know that there is direct connection between violet and yellow line. We just one **exchange Required** which is **Violet 🡪 Yellow** (Violet to Yellow)

Now we pass these connections to Exchange HashMap we made in 3.5.1 get the set of stations we must pass through on each line to reach to our desired line,

🡪Passing violet to yellow to exchange HashMap gives us

{violet, yellow}🡪Central Secretariat

{violet, yellow}🡪Kashmiri Gate

🡪We pass this station name to the Station HashMap to get station no:

{Central Secretariat, Violet}🡪some no X

🡪We now cal. 🡺 P = X – station no Badarpur border on violet line.

{Central Secretariat, Yellow}🡪some no Y

🡪We now cal. 🡺 Q = Y – station no AIIMS on Yellow line.

THEN, we sum P+Q to get final result.

If there are multiple exchanges, we repeat the same process for each path.

This process of calculation path cost for future and path cost for past has been derived from A\* algorithm which is a very fast and extremely efficient path finding algorithm used by various large tech. companies such as Facebook or Google and Amazon etc.

We selected this algorithm and broke all our data and processes accordingly because it seemed like the best choice to use when we know in which direction our goal lies but we did not use it completely but rather modelled our approach based on the same ideas on which this works as it needed the map for the entire metro to be coded which seemed like a very unnecessary thing to do for such a small requirement.

The code to do all this is mentioned below kindly look into it. In this code the function takes 4 inputs and some data structures which are:

1. vector<vector<int>>& dfs\_paths: It represents The connections/Exchanges we had calculated in the precious step.
2. map< pairstr, set< string >>& intersecting\_stations Contains the exchange Map we made.
3. map< pairstr, int > &m: This is the HashMap made in the very first step.
4. string>& line\_num: The line no associated to out stations.
5. string& start, string& end our starting and ending stations.

struct node{

    int val=0;

    vector<string> route;

};

Node is the data structure we will use to store the output.

The val represents the path cost.

Route is a vector representing the path taken in the form of station names string of.

Example of route can be, For the example we considered above.:

[“Badarpur Border”, “Central Secretariat”, “AIIMS”]

The full node will look like this:

{

Val= Some Value,

Route = [“Badarpur Border”, “Central Secretariat”, “AIIMS”] ,

}

The code associated is attached below please look though if you wish to.

vector<node>path\_with\_station(vector<vector<int>>& dfs\_paths , map< pairstr ,set< string >>&intersecting\_stations, map< pairstr, int > &m,map<int , string>& line\_num , string& start, string& end){

    vector<node>ans;

    for(auto i:dfs\_paths){

        if(i.size()>1){

           vector<node>curr;

            for(auto k:intersecting\_stations[{line\_num[i[0]],line\_num[i[1]]}]){

                struct node temp;

                string line=line\_num[i[0]];

                temp.val=abs(m[{k,line}]-m[{start,line}]);

                temp.route.push\_back(start);

                temp.route.push\_back(k);

                curr.push\_back(temp);

            }

            for(int j=1;j<i.size()-1;j++){

                vector<node>new\_curr;

                bool flag=false;

                for(auto k:intersecting\_stations[ {line\_num[i[j]],line\_num[i[j+1]]} ]){

                    for(auto l:curr){

                        flag=true;

                        struct node temp=l;

                        int n=temp.route.size();

                        string prev=temp.route[n-1];

                        temp.val+=abs(m[{k,line\_num[i[j]]}] - m[{prev,line\_num[i[j]]}] );

                        temp.route.push\_back(k);

                        new\_curr.push\_back(temp);

                    }

                }

                if(flag)

                    curr=new\_curr;

            }

            for(auto j:curr){

                string last\_line=line\_num[i[i.size()-1]];

                int n=j.route.size();

                string prev=j.route[n-1];

                j.val+=abs(m[{end,last\_line}] - m[{prev,last\_line}]);

                j.route.push\_back(end);

                ans.push\_back(j);

            }

        }

    }

    return ans;

}

**3.6 Modularity.**

We made sure to divide our code into small modules; we kept a separate module for each process such as

🡪a module for path finding

🡪a module for making graph

🡪a module for making hashes and reading data and so on.

🡪A separate module for input and output

🡪And break the code into header files and cpp files

How do we combine modules?

We combine the modules using the #include statements.

The STL functions we use are also modules in themselves or any library we use is also functions as a module that is why we use #include<string.> to include string

This means the code to make strings work is written in some file and we are telling the compiler that we will be using that file so it can link all the files mentioned via #include and we a get a working program.

This is done for the following reasons.

1. **Ease of Use:** This approach allows simplicity, as rather than focusing on the entire thousands and millions of lines code in one go we can access it in the form of modules. This allows ease in debugging the code and prone to less error.
2. **Reusability: It** allows the user to reuse the functionality with a different interface without typing the whole program again.
3. **Ease of Maintenance:** It helps in less collision at the time of working on modules, helping a team to work with proper collaboration while working on a large application.

C is called a structured programming language because to solve a large problem, C programming language divides the problem into smaller modules called functions or procedures each of which handles a particular responsibility. The program which solves the entire problem is a collection of such functions.  
Module is basically a set of interrelated files that share their implementation details but hide it from the outside world. How can we implement modular programming in c? Each function defined in C by default is globally accessible. This can be done by including the header file in which implementation of the function is defined.

*Example of Modular Programming in Our code*

To make the Graph for the stations we needed to declare a **Specific** data type and at the same time want to hide the implementation, including its data structure, from users. We can do this by first defining a public file called **Graph.h** which contains generic data Graph data type and the functions which are supported by the stack data type.  
In the **header file** we must include only the definitions of constants, structures, variables and functions with the name of the module, that makes easy to identify source of definition in a larger program with many modules.

Graph.h:

vector<vector<int >> void graph\_function(vector<vector<int >> &);

Now we can create a file named **Graph.cpp** that contains implementation of Graph data type:

#include "headers/Graph.h"

void graph\_function(vector <vector<int>> &graph) {

    string l;

    ifstream myfile("data/matrix.txt");

    if (myfile.is\_open())

    {

        int i=0;

        while (getline(myfile, l))

        {

            for(int j=0;j<l.size();j++){

                if(l[i]=='/')break;

                if(l[j]=='1'){

                    graph[i].push\_back(j);

                }

            }

            i++;

        }

        myfile.close();

        return;

    }

    else cout  << " Unable to open file";

}

The main file which may include module Worker.cpp

vector<vector< int >> graph(line.size());

graph\_function(graph);

//for checking the graph

for(auto i:graph){

    cout<<i.size()<<endl;

    for(auto j:i)cout<<j<<" ";

    cout<<endl;

}

Line.size() comes from the lines vector we read from the iostream in another function. code don’t worry about it.

**Chapter 4: Hardware and Software Requirements**

Hardware and software requirements of any project are must to be satisfied, so that the virtual environment can be set up on any machine to run the project. So in this section the software and the hardware requirements are discussed completely.

**4.1 Hardware Requirements**

No specific hardware requirements as C++ runs on pretty much every Modern system but make sure to have:

 1.6 GHz or faster processor with 2-4 GB RAM recommended

**4.2 Software Requirements**

A C++ program needs two things to run on a system in terms of software

1. An IDE (Integrated development environment: A simple notepad can also do this job but we used Visual studio Code along with Visual studio Community at times.
2. A C++ Compiler: There are a lot of C++ compilers in the market such as:

* [Apple C++](http://developer.apple.com/tools/macosxtools.html). XCode. It also comes with OS X on the developer tools CD.
* [Bloodshed Dev-C++](http://www.bloodshed.net/devcpp.html). A GCC-based (Mingw) IDE.
* [Clang C++](http://clang.llvm.org/). A relatively very active development associated with the analysis and code generation framework, LLVM.
* [Cygwin (GNU C++)](http://www.cygwin.com)
* [MINGW - "Minimalist GNU for Windows"](http://www.mingw.org). Another GCC version for Windows including a free (non-GPL) w32api.
* [GNU CC source](http://gcc.gnu.org/)
* [IBM C++](http://www-03.ibm.com/software/products/en/ccompfami) for IBM power, System Z, Bluegene, and Cell.
* [Intel C++ and many supporting tool](https://software.intel.com/en-us/forums/intel-system-studio)
* [Microsoft Visual C++](https://visualstudio.microsoft.com/downloads/).
* [Oracle C++](http://www.oracle.com/technetwork/server-storage/solarisstudio/overview/index.html).

All of these are free to download there are some you have to pay for but why the hell would you do that?

1. GitHub: We used GitHub to work on the project simultaneously.

**CHAPTER 5: RESULTS**

**// add here**

**CHAPTER 6: CONCLUSION**

The project was highly successful in finding the Paths between stations provided to it and it can find the optimal solution in a very small time.

Thus, it can be practical tool to navigate within the DMRC.

We tested the project at each step and phase of development with manually entering various inputs and testing their output against the most optimal path that we could find using the map and it can out accurately almost every time.

so, we can say that this is an accurate system which works quickly and efficiently to give its user the most optimal path from the DMRC on being given the source and destination as input.

**CHAPTER 7: REFRENCES**

1. To get data we used:

<https://www.mapsofindia.com/maps/delhi/delhi-metro-map.html>

2. To understand the Working and project making in C++ we used:

* Geeks for Geeks:

<https://www.geeksforgeeks.org/>

Used this site a lot to understand how and when and why to use which algorithm and which data structure.

* Medium:

<https://medium.com/swlh/pathfinding-algorithms-6c0d4febe8fd>

and Various other articles.

* Hacker Rank:

<https://www.hackerrank.com/>

to understand implementation of key concepts through Problem solving

* The cherno YouTube:

<https://www.youtube.com/playlist?list=PLlrATfBNZ98dudnM48yfGUldqGD0S4FFb>

This playlist Can teach you a lot about C++

3.Graph making Tools:

<https://graphonline.ru/en/create_graph_by_matrix>

4.Understanding Algorithms:

<https://medium.com/kredo-ai-engineering/search-algorithms-part-3-uninformed-search-algorithms-2-1bc85c0a3900>

5.Stack Overflow

<https://stackoverflow.com/>

To communicate with professionals who help me understand the errors in my code

**BRIEF PROFILE OF STUDENT**

Paste your scanned Photo

**Name** : Ashutosh Singh

**Roll No** : 17001003010

**Branch** : Computer Engineering

**Email Id** : asksingh11@gmail.com

**Brief About Project**:

This tool will function as an assistant to the user providing them with the fastest and shortest path to get from one station to another. The tool will also calculate the price of the journey. The user gives their origin and destination stations as an input and the tool finds the path for them and tells them where to change between lanes how many stops to ride in a lane to get to their destination.

This tool will also have an option to display the results in both English or Hindi languages based on the user’s specification.

**Future Scope**

The project can be extended to multiple more dimensions if there are public repositories of words to expose on the algorithms. The involvements can include.

1. Speech to text and Text to speech for input output facility.

2. Multiple language choices to help the uneducated people use the tool more effectively

3. A way to find the path from source to destination going through a particular station. Just break the journey into two sets and find the optimal path for both separately and then combine them into a single result.

4. GPS to find the entry station itself.